

Nachrichtenbasierte Informationssysteme

Mit dem MS Message Queue Server ein nachrichtenbasiertes Informationssystem entwickeln.

In zwei Szenarien werde ich Anwendungsgebiete für nachrichtenbasierte Informationssysteme aufzeigen. Anschließend wird ein aus mehreren Komponenten bestehendes Framework vorgestellt, mit dem ein solches Informationssystem einfach realisiert werden kann. Der Microsoft Message Queue Server bietet für den Transport solcher Nachrichten eine komfortable Infrastruktur.

Was sind die MS Message Queue Services?

Sie verwalten eine Menge von Nachrichtenwarteschlangen (Queues) mit deren Nachrichten und übernehmen den Transport von Nachrichten zwischen den Nachrichtenwarteschlangen, die auf beliebigen Computern im Netz existieren können. (Abbildung 1)

Dabei erfolgt die Kommunikation zwischen den Queues asynchron. Der Sender kann sich also nach dem Versenden anderen Aufgaben zuwenden und der Empfänger muß nicht online sein. Er kann sich seine Nachrichten abholen, wenn er dazu bereit ist.

Die MS Message Queue Services stellen dem Anwendungsentwickler eine Infrastruktur zur asynchronen Interprozeßkommunikation zur Verfügung, eine Art E-Mail zwischen Computerprogrammen. Eine Einführung finden Sie unter [1], [2]. Im folgenden werde ich zwei Szenarien vorstellen, bei denen die MS Message Queue Services zum Einsatz kommen.

Szenario 1: Aktive Informationsverteilung zwischen Systemkomponenten

In einem klassischem Client / Server System rufen die einzelnen Applikationen ihre Daten von einem Datenbankserver ab und präsentieren diese dem Anwender. Änderungen werden wieder in die Datenbank zurückgeschrieben, ohne andere Anwender darüber zu informieren. Diese „sehen“ die Änderungen erst, wenn sie erneut auf die Daten zugreifen.

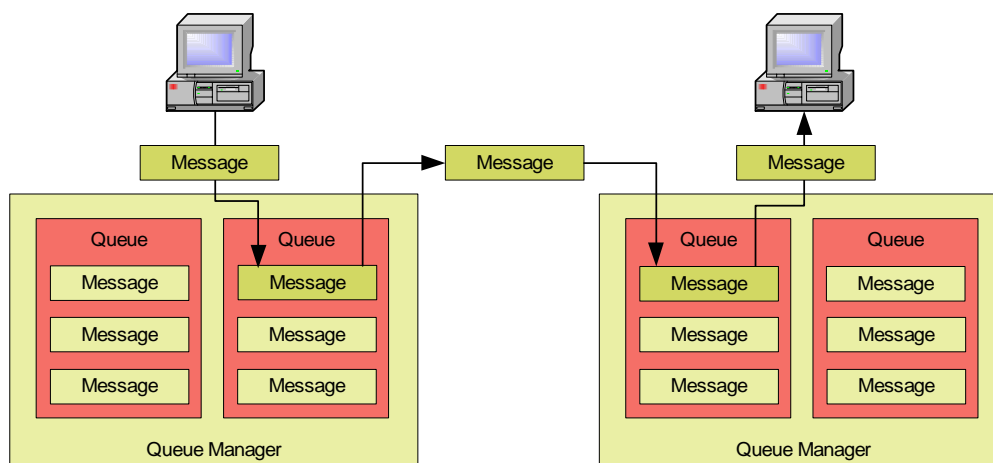


Abbildung 1: Die Queue Manager transportieren die Nachrichten zwischen den Queues der Computer

Überblick

Thema

Die Anforderungen an Informationssysteme werden immer höher. In vielen Anwendungsbereichen ist es notwendig, Systemänderungen an alle Betroffenen unverzüglich weiterzuleiten. Dabei kann nicht immer davon ausgegangen werden, daß alle Systemkomponenten ansprechbar sind. Um Blockaden zu vermeiden, werden Nachrichten asynchron ausgetauscht.

Bei modernen Informationssystemen interagieren die einzelnen Komponenten mit Hilfe von Nachrichten. Durch den Einsatz offener Standards wie z.B. XML für die Nachrichtenübermittlung, wird eine hohe Entkopplung der Systeme erreicht. Das ist vorteilhaft für die Integration eines Systems in eine „gewachsene“ IT-Landschaft und für die Erweiterbarkeit eines Systems.

Technik

MSMQ, XML

Voraussetzungen

NT4 Server / Win2000

VB Versionen: VB5, VB6

Zielgruppe

Systemarchitekten, Softwareentwickler

Solche Anwendungen bestehen aus passiven Komponenten. Alle Aktionen gehen vom Anwender aus, er stößt den Thread an (löst ein *Refresh* aus). Aktive Komponenten dagegen verfügen über einen „own thread of control“, sie schreiben nicht nur die Änderungen in die Datenbank, sondern informieren auch andere aktive Teile des Systems über Änderungen. Diese wiederum können sofort dem Anwender wichtige Neuigkeiten präsentieren.

Dieses Verhalten ist vor allem bei Informationssystemen gewünscht, die Anwender immer auf dem aktuellen Stand halten sollen. Stellen Sie sich z.B. ein Terminverwaltungssystem in einem Krankenhaus vor. Patienten der Stationen haben Termine im Diagnostischen Bereich (Funktionsbereich) des Krankenhauses. Jetzt trifft in der Rettungsstelle ein Notfall ein und der Diagnostische Bereich packt seine Geräte zusammen und geht zur Rettungsstelle. So ein Einsatz dauert ca. 2h. In dieser Zeit können keine anderen Patienten untersucht werden und alle Termine müßten abgesagt werden. Keiner hat die Zeit alle betroffenen Stationen herauszusuchen und diese anzurufen. Die Patienten gehen dann umsonst zum Diagnostischen Bereich (oft müssen sie von einer Schwester begleitet werden).

Würde aber am Informationssystem des Funktionsbereiches noch schnell auf den Notfallknopf gedrückt, der den Diagnostischen Bereich für einige Stunden sperrt, dann könnte ein aktives System alle anderen Bereiche des Krankenhauses benachrichtigen. Das Programm sucht jetzt alle Patienten heraus, die in dem Zeitraum einen Termin haben und informiert die betroffenen Stationscomputer. Die Stationschwester sieht jetzt auf dem Bildschirm, daß sich Patiententermine verschoben haben und keiner wird umsonst nach unten geschickt (*Abbildung 2*). Solche Benachrichtigungen werden bei allen die Kalender einer Abteilung betreffenden Ereignissen im System ausgetauscht. Dadurch erfahren alle Abteilungen der Klinik immer den aktuellen Stand der Patiententermine [5].

Wie können sich nun zwei Arbeitsplatzcomputer über solche Änderungen informieren? Sie senden sich über das Netzwerk Nachrichten zu. Das könnte jetzt alles mühsam von Hand selbst programmiert werden, aber der MS Message Queue Server stellt einem die notwendige Technologie dafür zur Verfügung. Er empfängt, versendet und verwaltet Nachrichten. Diese Nachrichten können auch Bestandteil einer MTS-Transaktion sein.

Der Nachrichtentransport mit MSMQ erfolgt asynchron. Schickt eine Komponente eine Nachricht an eine andere, so kann es vorkommen, daß diese gerade andere Aufgaben erledigt, oder der Computer nicht hochgefahren ist, so daß die Nachricht eine Weile unbeantwortet bleibt. In dieser Zeit kann die erste Komponente anderen Aufgaben nachgehen (z.B. weitere Nachrichten senden), sie blockiert nicht, wie dies bei einer synchronen Kommunikation der Fall wäre.

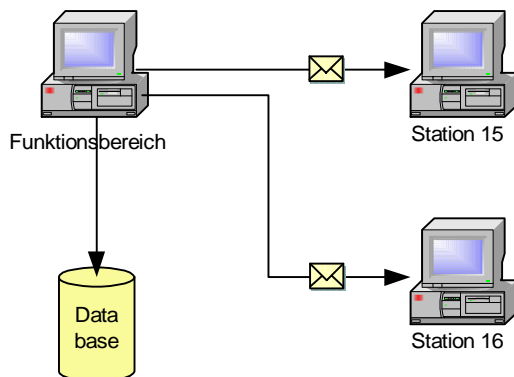


Abbildung 2: Aktive Informationsweiterleitung

Szenario 2: Lastverteilung durch Job-Queues

In einem Labor müssen täglich Tausende von Meßdaten in eine Datenbank importiert werden, die anschließend durch rechenintensive Vergleiche ca. 10.000 Mustern zugeordnet werden müssen. Um diese Arbeit zu beschleunigen wäre der Einsatz mehrerer Computer denkbar. Die zusätzlichen Maschinen könnten zum Beispiel die ganz normalen Arbeitsplatzrechner der Sekretärin oder anderer Mitarbeiter sein.

Wie könnte ein solches System aufgebaut sein? Ein oder mehrere Computer übernehmen den Import der Meßdaten (je nachdem wie zeitaufwendig der Import ist). Nun können die Meßdaten in die Datenbank eingetragen werden und in einer Job Queue wird eine Nachricht mit der DBID des importierten Datensatzes abgelegt. Alternativ dazu könnte mit dem Eintragen in die Datenbank auch bis nach dem Einsortieren gewartet werden: Dadurch werden Zugriffe auf die Datenbank gespart. Für den importierten Meßdatensatz kann zum Beispiel ein COM-Objekt instanziiert werden, welches dann mit der Nachricht in der Job Queue abgelegt wird (Objektpersistenz [4], [6]). Denkbar wären auch mehrere Job-Queues, auf welche die Arbeit verteilt wird. (*Abbildung 3*)

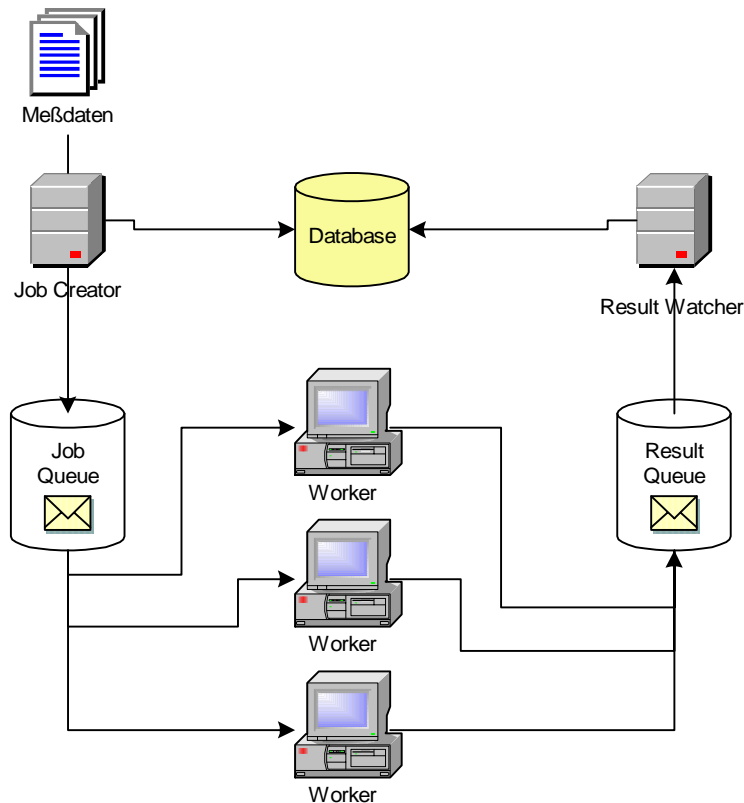


Abbildung 3: Architektur-Job Scheduling

Mehrere Instanzen eines Worker-Programms auf verschiedenen Rechnern holen sich aus der Job-Queue die einzelnen Aufträge, übernehmen die zeitaufwendige Einsortierung und senden die Ergebnisse an eine Ergebnis-Queue. Diese Ergebnis-Queue wird von einem weiteren Programm (ResultWatcher) überwacht, welches dann die Datenbank aktualisiert und eine aktuelle Statistik des Importprozesses ausgeben kann.

Die einzelnen Systemkomponenten sind bei diesem Ansatz sehr schön entkoppelt. Bei Bedarf können sehr einfach weitere Worker, JobCreators oder ResultWatcher hinzugefügt werden. Dieses Szenario eignet sich gut, wenn die einzelnen Ergebnisse der Worker nicht voneinander abhängen, also zwischen den einzelnen Workern keine oder nur geringe Abstimmung erfolgen muß (Worker muß nicht auf Antworten anderer Komponenten warten).

Nachricht (Typ)	Beschreibung	Beteiligte
NewImport (Request/Done)	Meldet dem ResultWatcher den Start eines neuen Imports	JobCreator, ResultWatcher
Job (Inform)	Auftrag in der Job-Queue ablegen	JobCreator, Worker
Result (Inform)	Ergebnis in Result-Queue ablegen	Worker, ResultWatcher
JobCount (Inform)	JobCreator informiert ResultWatcher über die Anzahl der erzeugten Jobs	JobCreator, ResultWatcher
AllJobsDone (Inform)	ResultWatcher informiert JobCreator, daß alle Jobs abgearbeitet wurden	ResultWatcher, JobCreator

Tabelle 1: Nachrichten, die zwischen den Systemkomponenten ausgetauscht werden

Die Kommunikation der einzelnen Komponenten erfolgt über Nachrichten, die mit dem MSMQ Server versendet werden. In Tabelle 1 werden die erforderlichen Nachrichten zusammengefaßt. Nachrichten, die eine Bestätigung erwarten, sind vom Typ Request und werden mit einem Done oder einem Refuse beantwortet. Inform-Nachrichten brauchen nicht bestätigt zu werden.

Installation NT4 / 9x

Einen kurzen Überblick über die Architektur des MSMQ finden Sie im *Kasten 1* oder in [2].

Die Installation des MSMQ ist unnötigerweise aufwendig. Als erstes muß auf einem NT4 Server ein PEC installiert werden. Voraussetzung für einen PEC ist ein lokaler SQL Server 6.5 oder höher, auf dem das MQIS gespeichert wird. Das Setup ist Bestandteil des NT4 Option Packs (siehe Visual Studio 6, CD 2). Achtung: Bei meiner deutschen Visual-Studio Version erhalte ich beim ersten MSMQ Setup auf NT Server die Fehlermeldung, daß die Datei „MQKeyhlp“ fehlt. Mit der englischen Version des Option-Packs funktioniert es, die weiteren Setups können mit der deutschen Version durchgeführt werden.

Anschließend können auf weiteren Computern Clients installiert werden, ein Site Controller ist nicht notwendig. Der PEC kann auch als Independent Client benutzt werden. Als Betriebssystem für die Clients kann Win9x, NT4 Workstation / Server und Windows 2000 eingesetzt werden. Während der Installation des Clients versucht das Setup Kontakt mit dem PEC aufzunehmen, um den Computer in das MQIS einzutragen. Achtung! Dafür benötigt der Setup-Prozeß administrative Rechte auf dem PEC. Sind Client und PEC nicht in der gleichen Domäne, dann hilft ein Mirror Account (gleicher Username und gleiches Passwort). Diese Erkenntnis hat mich einen Tag gekostet.

Sind zwei Independent Clients erst mal installiert, können sie auch ohne PEC wunderbar miteinander kommunizieren, solange sie Zugriffe auf das MQIS vermeiden (nur private Warteschlangen, siehe weiter unten).

Ein zwingender Zugriff während der Installation auf einen PEC erscheint daher nur lästig. Es gibt auch keine Möglichkeit, den PEC zu wechseln. Der Client muß deinstalliert werden (wobei er versucht sich aus dem MQIS auszutragen) und dann wieder neu installiert werden (Eintrag in das neue MQIS). Ein Bekannter teilte mir mit, daß er es geschafft hätte, einen Independent Client unter NT4 Workstation ohne Zugriff auf einen PEC zu installieren, konnte mir aber nicht mehr sagen wie, und ich konnte das auch nicht reproduzieren. Sollte es jemand noch einmal schaffen, dann würde ich gerne erfahren, wie es geht.

Nach dem Setup muß das aktuelle Service Pack von Windows NT (zur Zeit 6.0a) noch einmal installiert werden. Unbedingt auch auf das „a“ achten, da in dieser Version einige MSMQ Probleme behoben wurden. Alle Bestandteile des NT Option Pack zählen zum Betriebssystem und werden über die NT Service Packs aktualisiert.

Im Startmenü unter NT Option Pack befindet sich dann der Shortcut zum MSMQ Explorer (*Abbildung 4*) und zu einigen Beispielprogrammen.

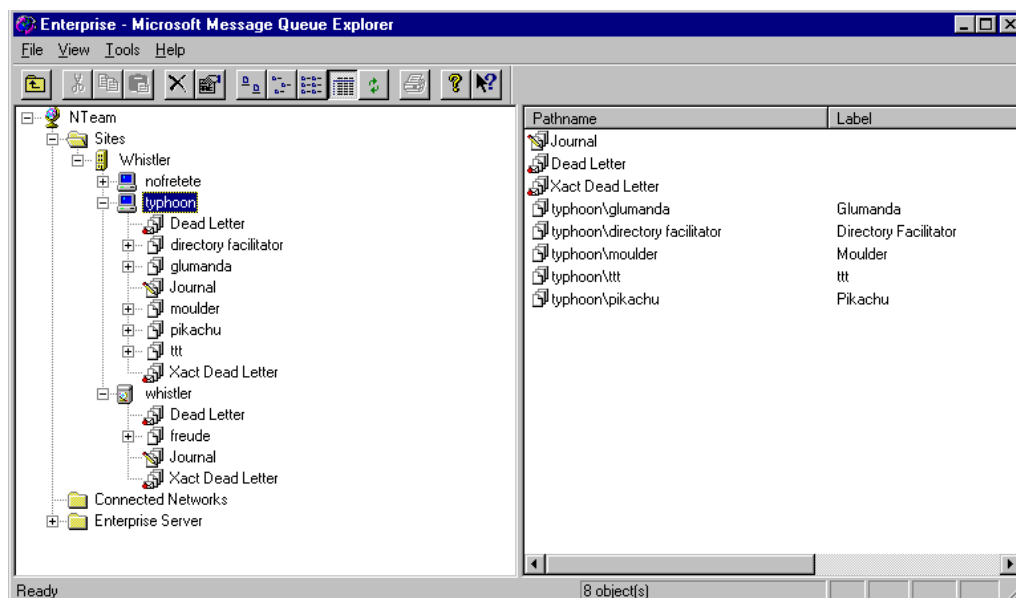


Abbildung 4: MSMQ Explorer unter NT4

Installation Win2000

Unter Windows 2000 kann man endlich auch ohne PEC und MQIS auskommen. Wahrscheinlich haben wir das COM+ zu verdanken, das auch ohne Active Directory Queued Components unterstützen soll. Unter Systemsteuerung → Software wird das Windows Setup aufgerufen (Windows Komponenten hinzufügen) und dort die MS Message Queue Services auswählen.

Im nächsten Dialog „Zugriff für das Active Directory manuell auswählen“ anklicken. Dadurch versucht das Setup nicht automatisch, einen PEC oder einen Active Directory Controller zu finden. Unter Windows 2000 Professionell können Sie zwischen unabhängigen und abhängigen Clients wählen.

Danach entscheiden Sie, ob Sie mit einem MQIS arbeiten wollen. Wenn ein NT4-PEC oder ein Active Directory-Controller vorhanden ist, dann kann dieser hier angegeben werden, sonst „Message Queuing greift nicht auf das Active Directory zu“ auswählen. Diese Option wird dann als *Workgroup Mode* (Arbeitsgruppenmodus) bezeichnet. Da kein MQIS existiert, sind keine Public Queues möglich und alle Operationen, die auf das MQIS zugreifen, führen zu einem Laufzeitfehler. Trotzdem kann man im Arbeitsgruppenmodus wunderschön arbeiten.

Leider kann auch unter Windows 2000 die Konfiguration nicht einfach geändert werden, deshalb muß der MSMQ immer deinstalliert (Neustart) und neu installiert (kein Neustart) werden, was sowohl unter NT4 als auch Windows 2000 prima funktioniert.

Die Verwaltung unter Windows 2000 ist jetzt in die Computerverwaltung (Abbildung 5) integriert (Start→Programme→Verwaltung→Computerverwaltung).

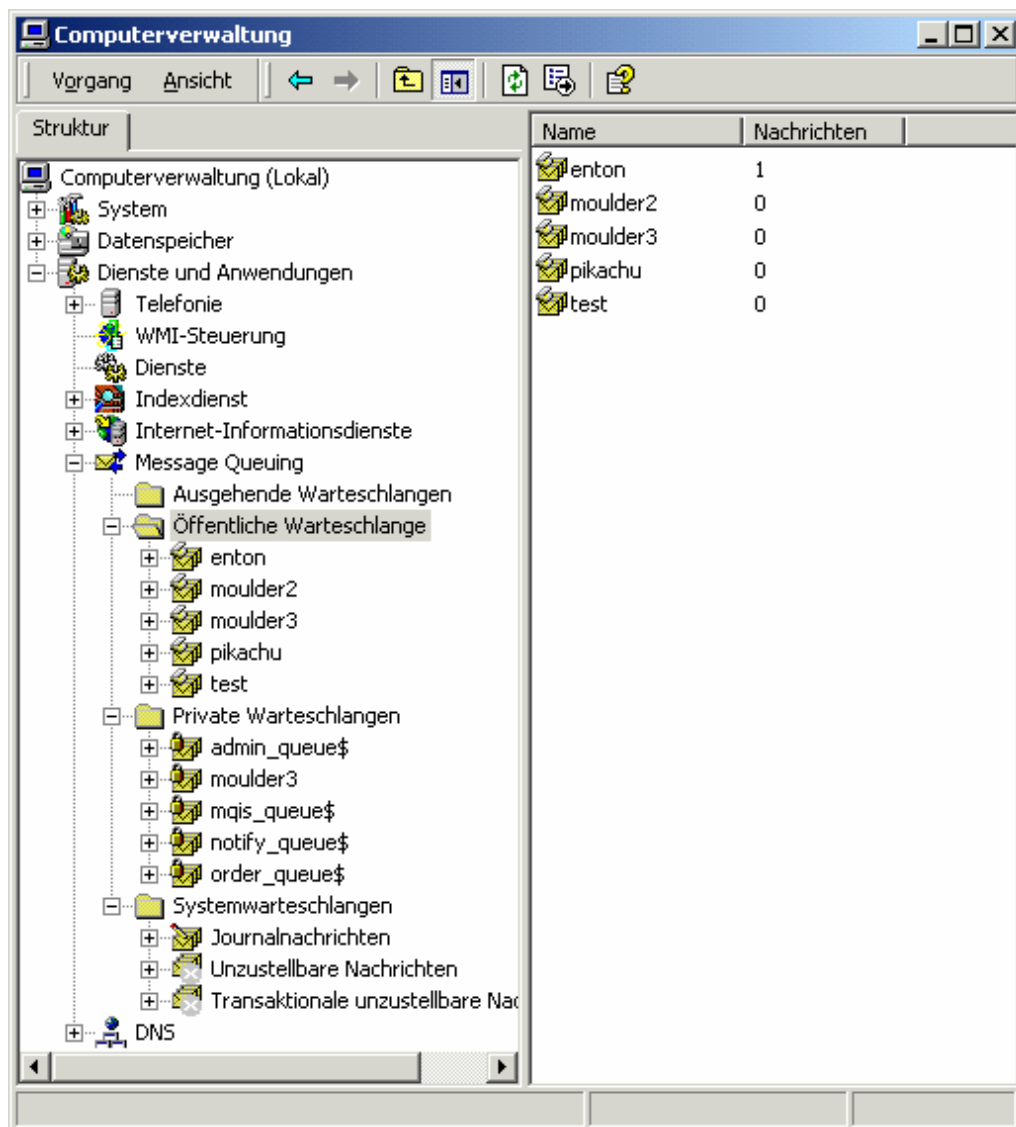


Abbildung 5: Computerverwaltung unter Windows 2000

Programmierung

Für die Programmierung der MS Message Queue Services stehen eine COM- und eine C-API zur Verfügung. Die vier wichtigsten Objekte der COM-API mit ihren Besonderheiten möchte ich hier kurz vorstellen. Eine ausführliche Erläuterung der MSMQ-APIs finden Sie in der MSDN. Im folgenden beziehe ich mich auf das MSMQ 1.0 API. MSMQ 2.0 läuft nur unter Windows 2000. Es wurde um einige Objekte und Methoden erweitert.

MSMQQueueInfo

Das QueueInfo-Objekt wird verwendet, um auf eine Queue zuzugreifen. Vergleichbar mit dem ADODB.Connection Objekt, stellt es eine Verbindung zu einer Queue her. Um eine Queue zu öffnen, muß entweder die Eigenschaft *PathName* oder die Eigenschaft *FormatName* des *QueueInfo*-Objektes spezifiziert werden. In *Tabelle 2* finden Sie die gebräuchlichsten Adressierungen. Darüber hinaus können Queues auch über ihre GUID geöffnet werden. Im WorkGroup Mode können Sie nur private Warteschlangen öffnen. In *Listing 1* wird eine Funktion der Klasse *CMSMQTransfer* gezeigt. Diese Funktion ist für das Setzen des *MSMQInfo*-Objektes verantwortlich. Sie bekommt den gewünschten Mode zum Öffnen der Queue und ein von mir entworfenes *CAddress*-Objekt (siehe weiter unten) übergeben. Mit diesen Daten initialisiert sie das *QueueInfo*-Objekt. Anschließend kann die *Open*-Methode aufgerufen werden, die eine Referenz auf ein *MSMQQueue*-Objekt zurückgibt:

```
Set q = qi.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
```

Pathname
theQueueInfo.PathName = "HostName\QueueName"
theQueueInfo.PathName = "HostName\Private\QueueName"
FormatName
theQueueInfo.FormatName = "DIRECT=OS:HostName\QueueName"
theQueueInfo.FormatName = "DIRECT=OS:HostName\PRIVATE\QueueName"
theQueueInfo.FormatName = "DIRECT=TCP:1.1.1.1\QueueName"
theQueueInfo.FormatName = "DIRECT=TCP:1.1.1.1\PRIVATE\QueueName"

Tabelle 2: Spezifizieren des QueueInfo-Objektes zum Lokalisieren einer Queue

```
Private Function SetQueueInfoByAddress(ByVal theQueueInfo As MSMQ.MSMQQueueInfo, _
    ByVal xAdr As CAddress, ByVal theOpenMode As QueueOpenModeEnum)
    If theOpenMode = QOMDirectTCP Then
        'FormatName="DIRECT=TCP:157.18.3.1\PRIVATE$MyQueue"
        If xAdr.QueueIsPrivate Then
            theQueueInfo.FormatName = "DIRECT=TCP:" & xAdr.HostIP & _
                "\PRIVATE$" & xAdr.Queue
        Else
            theQueueInfo.FormatName = "DIRECT=TCP:" & xAdr.HostIP & _
                "\" & xAdr.Queue
        End If
    ElseIf theOpenMode = QOMPathName Then
        'PathName=Server+Queue
        If xAdr.QueueIsPrivate Then
            theQueueInfo.PathName = xAdr.HostName & "\Private$" & xAdr.Queue
        Else
            theQueueInfo.PathName = xAdr.HostName & "\" & xAdr.Queue
        End If
    ElseIf theOpenMode = QOMDirectOS Then
        'DIRECT=OS:MachineName\QueueName
        If xAdr.QueueIsPrivate Then
            theQueueInfo.FormatName = "DIRECT=OS:" & _
                xAdr.HostName & "\PRIVATE$" & xAdr.Queue
        Else
            theQueueInfo.FormatName = "DIRECT=OS:" & _
                xAdr.HostName & "\" & xAdr.Queue
        End If
    Else
        Err.Raise vbObjectError + 2, , "Ungültiger QueueMode!"
    End If
End Function
```

Listing 1: Initialisieren des QueueInfo-Objektes

MSMQQueue

Nach dem Öffnen einer Queue können über das *MSMQQueue*-Objekt Nachrichten der Queue eingesehen (*Peek*) und entnommen (*Receive*) werden. Eine Queue ist dabei so etwas ähnliches wie ein Recordset. Die Nachrichten sind in der Reihenfolge ihres Eintreffens gespeichert.

MSMQMessage

Das *MSMQMessage*-Objekt repräsentiert eine Nachricht. Über verschiedene Optionen kann der Transport zwischen den Queues beeinflusst werden. Die *Send*-Methode erwartet ein *QueueInfo*-Objekt als Zieladresse.

MSMQEvent

Um herauszufinden, ob neue Nachrichten für eine Queue eingetroffen sind, könnte man in regelmäßigen Intervallen das *MSMQQueue*-Objekt überprüfen. Einfacher allerdings ist es, das *MSMQEvent*-Objekt zu nutzen:

```
Private WithEvents m_MSMQEvent As MSMQ.MSMQEvent
Set m_MSMQEvent = New MSMQ.MSMQEvent
m_Queue.EnableNotification m_MSMQEvent
```

Einem *MSMQQueue*-Objekt wird das *MSMQEvent*-Objekt übergeben (mit *WithEvents* deklariert), welches dann beim Eintreffen neuer Nachrichten ein Event auslöst. Dadurch braucht man nicht regelmäßig nachzusehen, ob eine neue Nachricht eingetroffen ist. Nach jedem Öffnen einer Queue und nach jedem *Arrived*-Event des *MSMQEvent*-Objektes muß mit *EnableNotification* die Eventbehandlung neu initialisiert werden.

Soviel zur Theorie! In der Praxis bin ich (und auch andere laut MS-Newsgroup) von der Zuverlässigkeit des *MSMQEvent*-Objektes sehr enttäuscht. Wenn Sie mit dem Beispielprojekt herumspielen, werden Sie seltsame Erfahrungen machen.

Während unserer Tests fand ich folgende Probleme. Wenn über einen längeren Zeitraum keine Nachricht in einer Queue ankommt, dann wird beim Eintreffen einer neuen Nachricht in der Queue kein *MSMQEvent* ausgelöst, obwohl sich dann eine neue Nachricht in der Queue befindet. Wird dann eine Nachricht vom Rechner der Queue abgesendet, feuerte kurz danach das Event welches über das Eintreffen der neuen Nachricht informiert. Dieser Effekt tritt in unregelmäßigen Abständen auf.

Ein anderes Problem ist, daß beim Öffnen einer Queue, in der sich Nachrichten befinden, nicht immer das Event ausgelöst wird. Lange Rede, kurzer Sinn: Man muß wohl doch einen Timer benutzen, um sicherzustellen, daß man keine neuen Nachrichten verpasst. Schade! Ich habe noch einen anderen Workaround gefunden. Unsere Komponenten senden alle paar Minuten eine Ping-Nachricht, daß sie noch am Leben sind. Dadurch scheint das obengenannte Problem umgangen worden zu sein, aber sehr vertrauenerweckend finde ich das nicht. In den News meldeten sich einige andere Entwickler, die einfach einen Timer einsetzen. Falls jemand einen besseren Workaround findet, dann bitte Mail an mich.

Besonderheiten

Im Workgroup-Mode (kein MQIS, kein ActiveDirectory, kein PEC) werden nur *Private Queues* unterstützt. Nachrichten lokaler Queues können empfangen und gesendet werden. Unter NT4 können Nachrichten von privaten *Remote Queues* nur empfangen, aber nicht gesendet werden. Werden Zugriffe auf das MQIS vermieden, dann kann auch ein bißchen Performance herausgeholt werden.

Für den Offline-Modus gelten eigene Einschränkungen. So hat ein Laptop, der offline ist, keinen Zugriff auf das MQIS (Active Directory oder PEC). Nachrichten werden lokal zwischengespeichert und sollten immer *recoverable* gesendet werden, da die Nachrichten sonst bei einem Neustart verloren gehen. Will man im Offline-Mode Nachrichten nicht nur lokal verschicken, muß man das MQIS vermeiden, also die *FormatName*-Eigenschaft des *MSMQQueueInfo*-Objektes verwenden.

Architektur eines MSMQ – basierten Informationssystems

Beispielanwendung

Die Beispielanwendung auf der CD hat zwei Aufgaben. Zum einen kann man mit ihr die verschiedenen Zugriffsmöglichkeiten (Technologie) auf den MSMQ testen. Dafür sind zwei einfache Nachrichten gedacht. Sendet ein Programm eine *Ping*-Nachricht an ein anderes Programm, dann antwortet dieses mit einer *Pong*-Nachricht.

Die zweite Aufgabe besteht darin ein kleines nachrichtenbasiertes Informationssystem zu demonstrieren. Die Station des Krankenhauses fordert einen Termin vom Funktionsbereich an. Das System besteht aus mehreren Komponenten, die verschiedene Aufgabe haben. Die einzelnen Komponenten werden in den folgenden Abschnitten erläutert.

Komponentenentwurf

In *Abbildung 6* wird der Entwurf für die einzelnen Komponenten des Systems dargestellt. Jede Clientanwendung verfügt über ein sogenanntes *Postoffice*, das Methoden zum Versenden von anwendungsspezifischen Nachrichten zur Verfügung stellt, und durch COM-Events den Empfang eingehender Nachrichten signalisiert. Die Aufgabe dieser Komponente besteht vor allem darin, die Komplexität des Nachrichtenverpackens, -versendens, -transports und -empfangs zu verbergen. Der Entwickler der Clientanwendung benutzt Methoden wie z.B.: *VerschiebeTermin(PatID)* und schreibt Code in Ereignisprozeduren wie z.B.: *myPostOffice_NewAppointment(PatID, StartTime, WorkplaceID)*.

Die Komponente *UtilPostOffice* verpackt die Methodenaufrufe in XML-Strings und versendet diese an die betreffenden *Postoffices* der Empfänger. Bekommt ein solches *PostOffice* eine neue Nachricht, dann prüft es um welche Nachricht es sich handelt und löst dann ein COM-Event aus.

Sowohl Client-Anwendung als auch *PostOffice* arbeiten dabei mit Objekten des Objektmodells der Anwendung also z.B. *CPatient*, *CStation*, *CService*-Objekten. Unterhalb der Komponente *UtilPostOffice* ist dieses Domänenwissen nicht bekannt. *UtilPostOffice* versendet und empfängt nur *UtilMessage.CMessage*-Objekte und hat keine Ahnung daß der MS Message Queue Server als Transportmechanismus im Spiel ist. Diese Objekte enthalten in der *Content*-Property den generierten XML-String. Verschiedene Kommunikationspartner können unterschiedliche *PostOffices* implementieren, je nach gewünschtem Wortschatz.

Die Komponente *UtilTransferMsg* ist für den Transport der *UtilMessage.CMessage*-Objekte verantwortlich. Ihre Aufgabe besteht darin, die Komplexität des MSMQ Zugriffs vor dem System zu kapseln. Ein zweiter Vorteil besteht darin, daß auch andere Transportmechanismen wie z.B. TCP benutzt werden können, ohne den Rest des Systems zu beeinflussen. Das *UtilMessage.CAddress*-Objekt legt über die *AddressType*-Property den Transportmechanismus fest, so daß auch zur Laufzeit die Transportmechanismen gewechselt werden können. Dadurch wäre es auch möglich mit Unix Systemen zu kommunizieren. *UtilTransferMsg* persistiert das *UtilMessage.CMessage*-Objekt in einen XML-String und versendet diesen mit der *Body*-Property des *MSMQMessage*-Objektes.

Die Komponente *UtilMessage* stellt Objekte für die Nachrichten und Adressverwaltung sowie ein *Content*-Objekt zur Verfügung. Das *Content*-Objekt ist dafür zuständig verschiedene Arten von Inhalten aufzunehmen (XML-Strings, persistierbare Objekte, Recordsets, Binärdaten).

Bei komplexeren Aufgaben sind oft mehrere Nachrichten an verschiedene Komponenten notwendig um eine Aktivität auszuführen, die alle beantwortet werden müssen, bevor die Aktivität abgeschlossen ist. Dafür habe ich weitere Objekte in *UtilMessage* eingeführt. *Operations* bilden eine solche Aktivität ab, *Conversations* ist eine Frage / Antwort Kommunikation zwischen zwei Systemkomponenten. Treffen neue Nachrichten ein, dann werden diese automatisch der richtigen *Operation* und *Conversation* zugeordnet. Ist eine *Conversation* oder eine *Operation* abgeschlossen, werden wieder COM-Events ausgelöst, die dem Postoffice mitteilen, das eine Aktivität beendet ist. In der Beispielanwendung auf der CD sind diese beiden Objekte jedoch aus Gründen der Vereinfachung nicht enthalten.

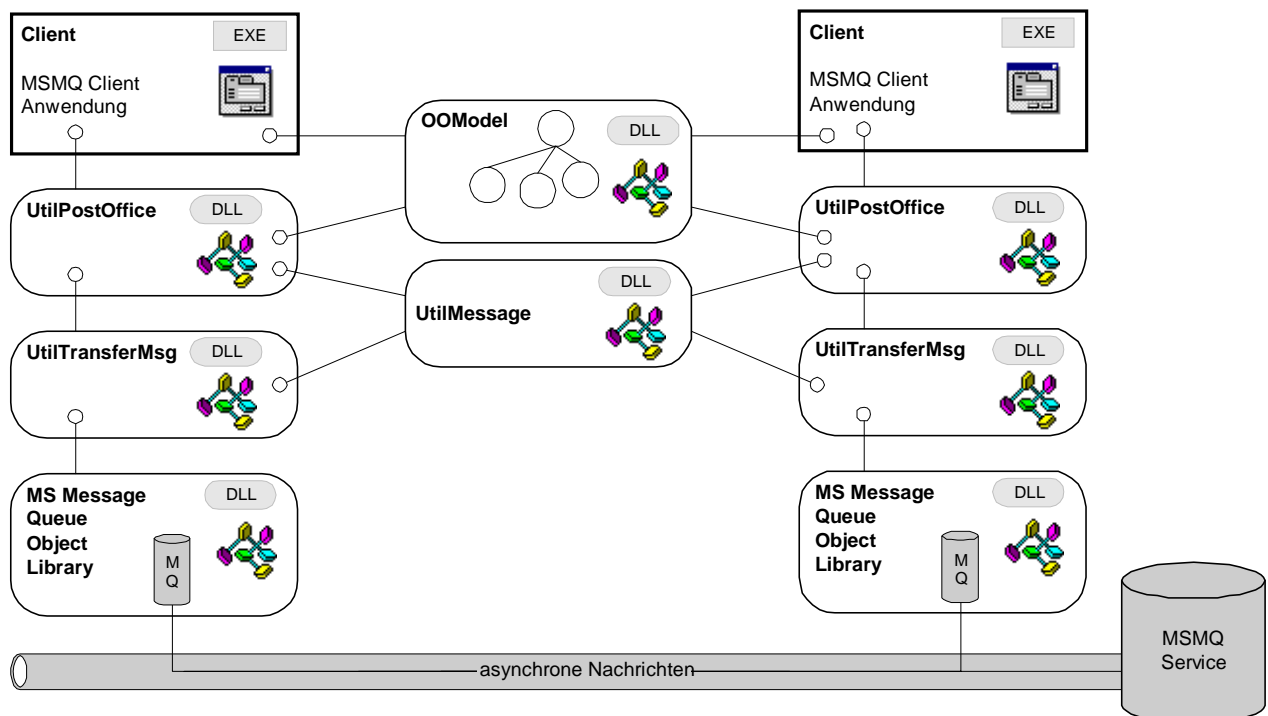


Abbildung 6: Architektur eines nachrichtenbasierten Systems

Realisierung des Komponententwurfes in den Beispielprojekten

Für diesen Artikel habe ich den Entwurf vereinfacht. Der Beispielcode besteht aus drei Projekten (Abbildung 7). Die Komponente *UtilPostOffice* habe ich mit in das Client-Projekt (Klasse *CPostOffice*) eingefügt. Idealerweise würden Objekte der *UtilMessage*-Komponente nur in der Klasse *CPostOffice* benutzt werden, da aber der Client an

der Oberfläche die ein- und ausgehenden Nachrichten zu Testzwecken anzeigen soll, werden in *frmMain* ebenfalls *UtilMessage*-Objekte benutzt.

Das Projekt *OOModel* enthält die Klasse *CExamination*, die eine medizinische Untersuchung repräsentiert. Normalerweise würde das Projekt auch Patienten, Funktionsbereichs Objekte und viele weitere enthalten. Das Klassenmodul *IXMLObject* enthält einige Interfaces, die sowohl von den *OOModel*- als auch von den *UtilMessage*-Objekten implementiert werden. In der Praxis empfiehlt sich die Verwendung einer separaten *TypeLibrary*. *CGlobalCode* ist eine *GlobalMultiUse*-Klasse, die einige projektübergreifende Methoden enthält. Der *Client* und auch *UtilMessage* referenzieren die Komponente *OOModel*.

In dem Projekt *UtilMessage* sind die Komponenten *UtilMessage* und *UtilTransferMsg* zusammengefaßt. Nur die Klasse *CMSMQTransfer* gehört zu *UtilTransferMsg*, alle anderen Klassen gehören zu *UtilMessage*. In den folgenden Abschnitten werden die einzelnen Projekte ausführlich erläutert.

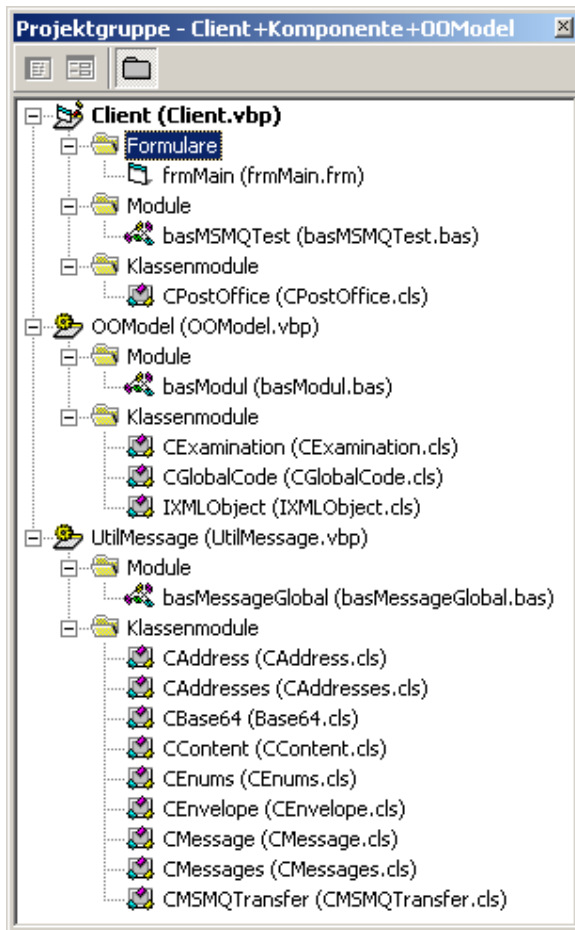


Abbildung 7: Dateien der Beispielprojekte

Projekt „Client“

Clientanwendung

Die Clientanwendung (Abbildung 8) hat zwei Aufgaben. Zum einen soll sie verschiedene Optionen des Zugriffs auf den MSMQ ermöglichen (*Ping*, *Pong* Nachrichten), zum anderen die Informationsübermittlung in einem nachrichtenbasierten Informationssystem (*GetExaminationDate*-Nachricht) demonstrieren. Die Client-Anwendung ist sowohl Sender als auch Empfänger. Am besten Sie starten jeweils eine Instanz auf zwei Rechnern. Es funktioniert aber auch, wenn Sie zwei Instanzen auf einem Rechner starten. Sie können sogar mit nur einer Instanz Ihre Tests durchführen. Sie öffnen die erste Queue, senden an eine zweite Queue und öffnen dann die zweite und antworten der ersten Queue. Hier wird sehr schön die Asynchronität deutlich.

Die *Ping*-Nachricht wird mit einem *Pong* beantwortet. Erhält ein Client die *GetExaminationDate*-Nachricht, dann entscheidet ein Zufallsgenerator ob ein Termin vergeben werden kann und die Nachricht wird entweder mit einem *ctMTDone* oder mit einem *ctMTRefuse* beantwortet. Das Sequenzdiagramm in Abbildung 9 zeigt den Informationsfluß zwischen den Komponenten der bei einer Terminanforderung stattfindet.

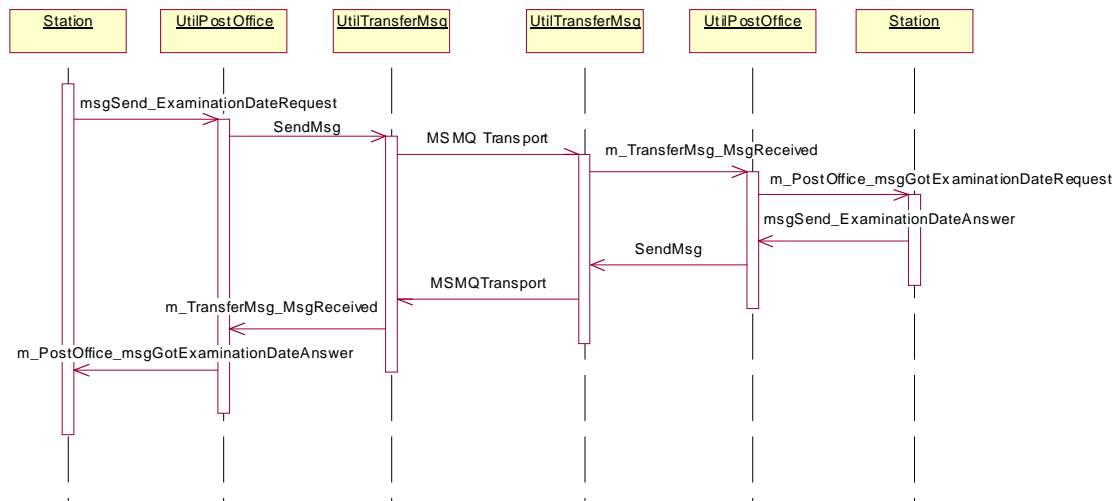


Abbildung 9: Sequenzdiagramm; eine Station fordert einen Untersuchungstermin an

Die Oberfläche ist in vier Bereiche aufgeteilt: Buttonleiste, Adressenliste, Nachrichtenliste, Nachrichtentext. Die Adressen stammen aus einer XML-Datei, die über den Button *LoadAddresses* neu geladen werden kann. Das ist sehr einfach, da die Klasse *UtilMessage.CAddresses* sich als XML-Datei persistieren kann. Die Datei befindet sich im gleichen Verzeichnis wie die Datei *Client.vbp* und kann mit jedem Texteditor bearbeitet werden. Sie müssen *HostIP*, *HostName* und eventuell *Private* anpassen (Arbeitsgruppeninstallationen unterstützen nur *Private* Queues). In der *Messages*-Liste werden sowohl ausgehende als auch eingehende Nachrichten angezeigt. Wird auf eine Nachricht geklickt, erscheint in der Textbox darunter der XML-String.

OpenQueue öffnet eine Queue, die durch die **markierte** Adresse selektiert wird. Existiert die Queue nicht, dann wird sie neu angelegt. Beim Neuanlegen wird die *CheckBox chkCreateWithJournal* berücksichtigt.

SendPingMsg und *GetExaminationDate* (Abbildung 9) senden an alle **gecheckten** Adressen Nachrichten.

Sowohl beim Öffnen einer Queue als auch beim Versenden einer Nachricht werden die Optionbuttons *Open* & *Send* berücksichtigt.

Die Clientanwendung ist fein raus. Sie hat eine mit *WithEvents* deklarierte Objektreferenz auf ihr *PostOffice*. Für sie erscheint das *PostOffice* wie ein zusätzliches COM-API. Sie braucht nur dessen Methoden aufzurufen und auf dessen Events zu reagieren (Listing 2). Die einzige Besonderheit sind die asynchronen Methodenaufrufe (Nachrichten). Ergebnisse / Antworten kommen nach unbestimmter Zeit an.

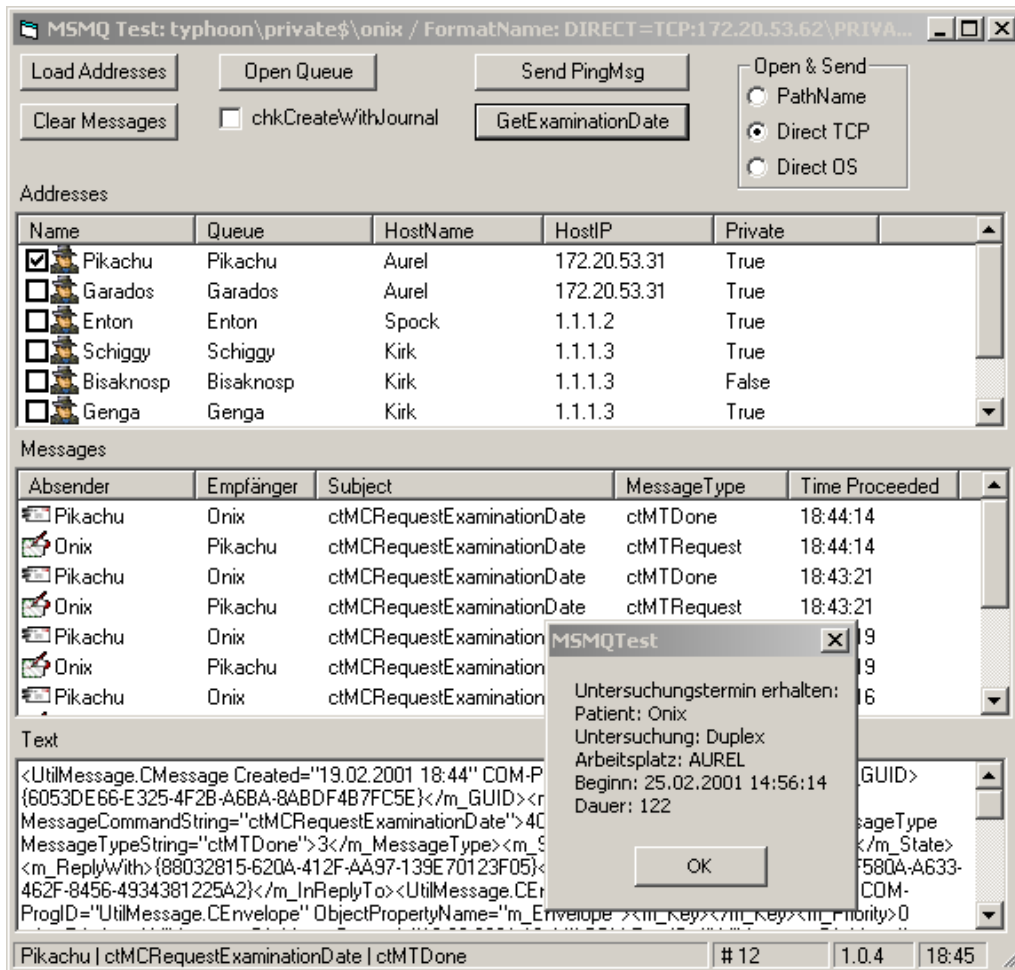


Abbildung 8: Oberfläche des Testclients

```
Private Sub m_PostOffice_msgGotExaminationDateRequest(xExam As CExamination, _
    Sender As UtilMessage.CAddress, _
    ReplyWith As String)

    Dim IsOK As Boolean

    IsOK = (GetRndNumber(1, 2) = 1)

    xExam.WorkplaceName = m_LocalComputerName
    If IsOK Then
        xExam.Duration = GetRndNumber(30, 150)
        xExam.StartDate = DateAdd("n", GetRndNumber(1440, 10000), Now)
        Call m_PostOffice_msgSend_ExaminationDateAnswer(xExam, Sender, _
            True, GetQueueOpenModeEnum)
    Else
        Call m_PostOffice_msgSend_ExaminationDateAnswer(xExam, Sender, _
            False, GetQueueOpenModeEnum)
    End If

End Sub
```

Listing 2: Eventprozedur des PostOffice-Objektes für Anforderung von einem Untersuchungstermin. Die Antworten werden über Methoden des PostOffice-Objektes geschickt.

UtilPostOffice

Diese Komponente wurde als zusätzliche Abstraktionsschicht eingeführt, um die Entwicklung der Clients zu vereinfachen. Für alle ausgehenden Nachrichten gibt es spezielle *msgCreate*-Methoden (Listing 4) und für eingehende Nachrichten spezielle *msgReceive*-Methoden (Listing 5). Die *msgCreate*-Methoden erhalten konkrete Eingabeparameter, je nach Nachrichtenzweck, und bauen daraus ein *CMessage*-Objekt (Listing 4).

UtilPostOffice verfügt über eine mit *WithEvents* deklarierte Referenz auf ein *UtilTransferMessage*-Objekt. Trifft eine Nachricht in der MessageQueue ein, dann wird sie von *UtilTransferMessage* in eine dem System verständliche

UtilMessage.CMessage-Nachricht umgewandelt (Syntax), die dann wiederum von einer *msgReceive*-Methode in ein ganz gewöhnliches COM-Event (*Listing 3*) mit konkreten Parametern (Semantik) transferiert und an den Client hochgereicht wird. Der Client braucht von dem ganzen Zirkus des Nachrichtentransports nicht viel zu wissen (kein *CMessage*, *CMSMQTransfer*,...). Er braucht nur die richtigen *msgCreate*-Methoden aufzurufen und kann in den Eventprozeduren des *CPostOffice*-Objektes auf eintreffende Nachrichten sehr einfach reagieren.

```
'Service
Public Event ChangedService( _
    ByVal ServiceID As Long, _
    ByVal ClientOUID As Long, _
    ByVal ProviderOUID As Long, _
    ByVal oldProviderOUID As Long, _
    ByVal ChangeMode As ctChangeModeService, _
    ByVal OrgMsg As CMessage, _
    ByVal SendingUAID As Long)
```

Listing 3:PostOffice - Deklaration des Events für den Client

```
Public Function msgCreate_ChangedService( _
    ServiceID As Long, _
    ClientOUID As Long, _
    ProviderOUID As Long, _
    oldProviderOUID As Long, _
    ChangeMode As ctChangeModeService, _
    SendingUAID As Long, _
    RecAdr As CAddress) As CMessage
    Dim aMsg As CMessage
    Dim root As MSXML.IXMLDOMElement
    Dim xEntry As MSXML.IXMLDOMElement
        'Parameter
    Set root = GetXMLRootElementUserDefined("Parameter")
    Call SetXMLParameterIntoElement(root, "ServiceID", ServiceID)
    Call SetXMLParameterIntoElement(root, "ClientOUID", ClientOUID)
    Call SetXMLParameterIntoElement(root, "ProviderOUID", ProviderOUID)
    Call SetXMLParameterIntoElement(root, "oldProviderOUID", oldProviderOUID)
    Call SetXMLParameterIntoElement(root, "ChangeMode", ChangeMode)
    Call SetXMLParameterIntoElement(root, "SendingUAID", SendingUAID)
        'Message
    Set aMsg = New CMessage
    Call aMsg.Constructor(ctMCServiceChanged, _
        ctMTInform, root, _
        ctCTDataAsXMLDOMElement, _
        RecAdr, PostOfficeAddress, "" _
    )
    Set msgCreate_ChangedService = aMsg
End Function
```

Listing 4:PostOffice - Methode zum Erstellen einer neuen Nachricht

```
Private Sub msgReceived_ServiceChanged(ByVal xMsg As CMessage)
    Dim root As MSXML.IXMLDOMElement
    Dim COUID&, POUID&, oldPOUID&, CM&, SrvID&, SendingUAID&
    Set root = xMsg.Content.GetContent
        'Parameter
    SrvID& = root.selectSingleNode("ServiceID").nodeTypedValue
    COUID& = root.selectSingleNode("ClientOUID").nodeTypedValue
    POUID& = root.selectSingleNode("ProviderOUID").nodeTypedValue
    oldPOUID& = root.selectSingleNode("oldProviderOUID").nodeTypedValue
    CM = root.selectSingleNode("ChangeMode").nodeTypedValue
    SendingUAID& = root.selectSingleNode("SendingUAID").nodeTypedValue
        'Event auslösen
    RaiseEvent ChangedService(SrvID, COUID, POUID, oldPOUID, CM, xMsg, SendingUAID&)
End Sub
```

Listing 5:PostOffice - Methode zum Auswerten einer empfangenen Nachricht

Für verschiedene Clients eines Systems könnten unterschiedliche PostOffices implementiert werden, die jeweils nur den benötigten Wortschatz enthalten. Das erwies sich aber als unpraktisch, wenn häufig Änderungen an den Nachrichten vorgenommen werden. So existieren *msgCreate*-Methoden und *msgReceive*-Methoden meist paarweise. Deshalb haben wir zur Zeit alle Nachrichten in einem PostOffice zusammengefaßt, es ist aber bei Bedarf auch hier sehr einfach dieses, in mehrere aufzuteilen.

Projekt „OoModel“

Dieses Projekt ist sehr einfach gehalten. Normalerweise würde hier ein komplettes Objektmodell (die Anwendungslogikschicht) implementiert werden. Es gibt nur eine Klasse, *CExaminations*, die eine med. Untersuchung repräsentieren soll. Diese Klasse implementiert das Interface *IXMLObjekt* und kann so direkt als peristierter XML-String versendet werden. Der Anforderer erzeugt ein *CExamination*-Objekt, setzt die gewünschte Untersuchung und schickt es zum Funktionsbereich, der dann, wenn möglich, einen Termin einträgt und das Objekt zurück sendet.

Projekt „UtilMessage“

CMSTMQTransfer

Diese Klasse hat die Aufgabe, *UtilMessage.CMessage*-Objekte zu einer Zieladresse zu transportieren. Nur in dieser Komponente wird mit dem MSMQ Service gearbeitet. Es wäre also später denkbar, eine weitere Transferkomponente zu implementieren, die einen Transport über POP3 / SMTP, WinSock oder andere Protokolle realisiert.

Über eine *Constructor*-Methode, die immer nach dem Instanzieren aufgerufen werden muß, wird die Komponente initialisiert und ein *MSMQQueue*-Objekt geöffnet, an dem auf eintreffende Nachrichten „gelauscht“ wird.

Mit der *Send*-Methode können *UtilMessage.CMessage*-Objekte versandt werden. Dabei können verschiedene Adressierungsarten verwendet werden. Das *UtilMessage.CMessage*-Objekt wird in der *Send*-Methode serialisiert (XML-String) und dann in den Body einer *MSMQMessage* gepackt.

Wenn eine neue Nachricht eintrifft, wird ein COM-Event durch das *MSMQEvent*-Objekt gefeuert. In der Ereignisprozedur dieses Objektes wird aus dem Body des *MSMQMessage*-Objektes wieder ein *UtilMessage.CMessage*-Objekt erzeugt, welches dann mittels *RaiseEvent* an *UtilPostOffice* hochgereicht wird.

CMessage

Das *UtilMessage.CMessage*-Objekt wird über den MSMQ Server versendet. Dabei werden die Eigenschaften *MessageCommand* und *MessageType* angegeben. *MessageCommand* ist der eigentliche Befehl oder die Anfrage. *MessageType* definiert, wie mit der Nachricht umgegangen werden soll. *Inform*-Nachrichten werden nur zur Kenntnis genommen, *Requests* sollen beantwortet werden. Bei Antworten oder Anfragen ist *ReplyWith* das eigene Aktenzeichen und *InReplyTo* das fremde.

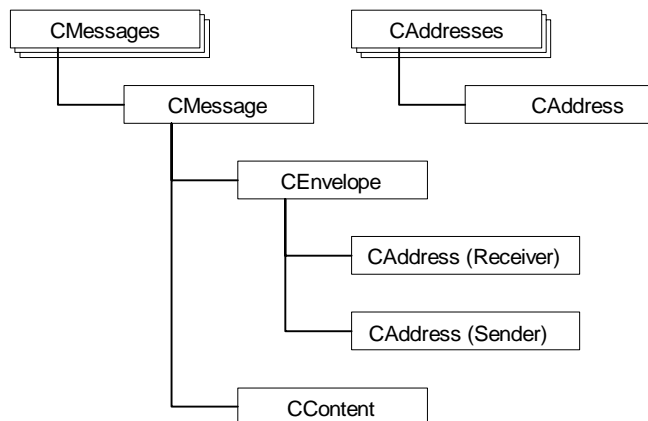


Abbildung 10: Objektmodell der Komponente UtilMessage

Durch das *UtilMessage.CMessage*-Objekt ist das System unabhängig vom MSMQ und seinem *MSMQMessage*-Objekt. Darüber hinaus enthält es systemspezifische Erweiterungen.

Zum *UtilMessage.CMessage*-Objekt gehören einige untergeordnete Objekte, die im folgenden erläutert werden. (Abbildung 10)

CEnvelope

Der Briefumschlag enthält Angaben, die für das Versenden der Nachricht wichtig sind. Ein Transporteur braucht dadurch nicht die ganze Nachricht zu untersuchen, alle Angaben wie Absender, Empfänger und Zustelloptionen findet er im *CEnvelope*-Objekt. Da die Objekte als XML-Strings übertragen werden, könnte ein eventgesteuerter XML-Parser nach dem Envelope aufhören und dann die Nachricht an die Empfänger weiterleiten. In unserem System gibt es tatsächlich eine solche Verteilungskomponente, die dafür sorgt, daß Änderungsinformationen an alle betroffenen Agenten verteilt werden.

CAddress

Das *CAddress*-Objekt (*Listing 4*) enthält alle Angaben zum Finden eines Kommunikationspartners. Wenn die Transportschicht auch andere Transportmechanismen unterstützt, dann wird über die Adresse die Transportmethode für den Client völlig transparent definiert. Eine Unix-Maschine erhält die XML-Strings über Socket, andere über MSMQ. Die Transportschicht prüft das *CAddress*-Objekt und wählt dann die erforderliche Transportmethode.

```
<CTUtilMessage.CAddress Created="05.01.2001 22:56"
  COM-ProgID="CTUtilMessage.CAddress"
  ObjectPropertyName="m_Col">
  <m_GUID>{655CFE66-F95E-4388-9229-4DC047C9EFEB}</m_GUID>
  <m_AddressName>Enton</m_AddressName>
  <m_HostName>Scotty</m_HostName>
  <m_Queue>Enton</m_Queue>
  <m_QueueIsPrivate>0</m_QueueIsPrivate>
  <m_HostIP>1.1.1.1</m_HostIP>
  <m_HostPort>0</m_HostPort>
  <m_AddressType AddressTypeString="ctMTS_MSMQ">2</m_AddressType>
  <m_Tag/>
</CTUtilMessage.CAddress>
```

Listing 4: Ein CAddress-Objekt als serialisierter XML-String

CContent

Das Content-Objekt ist dafür verantwortlich, verschiedene Inhalte aufzunehmen und zu serialisieren. Mit der Methode *SetContent* können verschiedene Objekttypen oder Strings zugeordnet werden. (*Tabelle 3*)

```
Public Sub SetContent(vContent As Variant, ConType As ctContentType)
```

Enum ctContentType	Erläuterung
ctCTDataAsXMLDOMEElement	beliebiges MSXML.IXMLDOMEElement
ctCTObjectAsICT_XML	ein COM Objekt, das IXMLObject implementiert
ctCTADODBRecordset	ein ADODB.Recordset
ctctMessageAsString	beliebiger Text
CtCTBinary	Daten, die Mime codiert werden müssen (Bilder, Binärdaten)
ctCTDataAsXMLString	Daten kommen als XML String und werden als MSXML.IXMLDOMEElement gespeichert

Tabelle 3: mögliche ContentTypen für das UtilMessage.CContent-Objekt

Über *GetContent* wird dann wieder eine Objektreferenz bzw. ein String nach außen gegeben. Die eigentliche Herausforderung besteht darin, die verschiedenen Contenttypen so zu serialisieren, daß ein gültiger XML-String herauskommt und umgekehrt wieder ein identisches Objekt zu erzeugen (siehe Objektpersistenz [6]). Die Implementierung für das Recordset kann seit MDAC 2.5 sehr schön mit einem Stream-Objekt gelöst werden (zur Zeit wird noch mit MDAC 2.1 das Recordset temporär auf die Festplatte geschrieben). Für die Binärdaten muß man sich etwas anderes einfallen lassen. Meine Implementierung dafür ist nicht fertig. Ich habe dafür eine Klasse aus der BP Wildsite [9] *Base64* eingesetzt. Mit deren Hilfe werden aus den Binärdaten ASCII Zeichen, die dann in einen XML `<![CDATA[. . .]>` Abschnitt eingepackt werden können.

Objekte mit dem MSMQ verschicken

Der *Body*-Eigenschaft eines MSMQMessage-Objektes wird normalerweise ein String zugewiesen. Es ist aber auch möglich, Referenzen auf instanziierte COM-Objekte zu übergeben. Diese müssen allerdings eines der folgenden Interfaces implementieren [8]:

- IPersistStream,
- IPersistStorage

In VB setzen Sie dazu einfach die *Persistable*-Eigenschaft eines Klassenmoduls auf „Persistable“ (1) und implementieren die zusätzlichen Methoden *Class_ReadProperties* und *Class_WriteProperties*. Für die Persistierung wird ein PropertyBag-Objekt eingesetzt, welches allerdings ein proprietäres Format verwendet. Deshalb haben wir eine eigene Schnittstelle implementiert, die auf XML basiert: *IXMLObject*. Wird ein *CMessage*-Objekt versendet, so

packt es sich und seine untergeordneten Objekte ein. Alle Objekte, die versendet werden, implementieren diese Schnittstelle. In den Klassenprozeduren für die Persistierung können die Methoden der *IXMLObject*-Schnittstelle benutzt werden. In das *PropertyBag*-Objekt wird einfach der XML-String reingeschrieben.

Werden in einem verteilten Informationssystem Objekte versendet, dann muß man sich Gedanken über eindeutige ObjektIDs machen. Objekte, die aus einer zentralen Datenbank kommen, können den Primärschlüssel verwenden. Werden jedoch verteilte Datenbanken eingesetzt oder stammen die Objekte gar nicht aus Datenbanken (wie die Adressen oder erzeugte und versendete Nachrichten) dann wird eine systemweit eindeutige Unterscheidung notwendig. Damit solche Objekte unterscheidbar sind, werden GUIDs vergeben.

Schlußbemerkungen

Vor- und Nachteile

Der Microsoft Message Queue Server stellt eine komfortable Infrastruktur für den Nachrichtentransport zwischen Computern zur Verfügung. Auch für Nicht-Windowsplattformen wurden Anbindungen an MSMQ entwickelt, diese werden aber wohl nicht sehr häufig eingesetzt.

Schade ist, daß die Installation des MSMQ unter NT4 und Win9x so aufwendig ist. Existiert kein PEC (mit lokalem SQL-Server), dann gibt es auch kein MSMQ. Außerdem hatten wir mit privaten Queues so unsere Probleme. (*Kasten 1*)

Wenn die einzelnen Systemkomponenten nur über XML-basierte Nachrichten kommunizieren, dann ist es sehr einfach, Komponenten, die auf Nicht-Windowsplattformen laufen, einzubinden. Auch die Erweiterung eines solchen Systems und die Zusammenarbeit des Systems mit fremden Systemen wird einfacher. Wir haben ein Terminplanungssystem entwickelt, welches die Termine der Inneren Klinik verwalten soll. In der Radiologie existiert ein elektronischer Kalender, in den alle geplanten Untersuchungen eingetragen werden. Dieses System schickt eine HL7-Nachricht an unser System, diese muß syntaktisch übersetzt werden (in eine unserer Nachrichten), und schon haben unsere Patienten einen neuen Termin in ihrem Kalender. Auf welcher Plattform / welcher Sprache das andere System realisiert ist, ist uns egal. Im klinischen Bereich existieren dafür ganz gute Ansätze. Als Kommunikationsstandard wurde dort HL7 vereinbart. Vereinfacht gesagt, werden textbasierte Nachrichten zu Kommunikationsservern gesendet, die diese dann verteilen. In der neuesten Version von HL7 werden diese Nachrichten dann auch XML-basiert sein.

Die meisten Nachrichten, die wir versenden, entsprechen inhaltlich Methodenaufrufen, nur eben asynchron. Bei klassischen Methodenaufrufen werden die Methodensignaturen von einem Compiler geprüft und die Intellisense Hilfe (Lobet Bill!) nimmt einem viel Arbeit ab. Wenn man XML-Nachrichten versendet, dann muß man beim Ein- und Auspacken darauf achten, daß alle Signaturen korrekt sind. Durch das *PostOffice* haben wir diese Probleme vereinfacht, da sich die Prozeduren für das Ein- und Auspacken der Nachrichten in der gleichen Klasse befinden. Will man aber verschiedene Systeme aneinander koppeln, oder hat man unterschiedliche *PostOffices* (bei einer Nachricht gibt's meist einen bestimmten Sender (Station) und einen bestimmten Empfänger (Diagnostischer Bereich), so daß nicht in jedem *PostOffice* alle *Send*- und alle *Receive*-Methoden implementiert sein müssen), dann muß man gut aufpassen, daß keine Parameter falsch übergeben werden. Hier könnte zum Beispiel eine Schemadefinition der XML-Nachrichten helfen, oder ein Mechanismus ähnlich der *Web Service Description Language* (WSDL) für SOAP. In der nächsten Version des Visual Studios wird die IDE dann auch Intellisense für Webservices unterstützen, die Informationen dafür werden aus einer solchen Schemadatei geholt. Vielleicht kann man das dann auch für sein eigenes Nachrichtensystem einsetzen.

Während unserer Arbeiten an diesem System stellt sich das Kompilieren der Projekte immer wieder als ein Problem heraus. Wir verwenden noch einige weitere Komponenten und für das komplette Durchkompilieren benötige ich inzwischen nur noch 30-240 Minuten (war schon viel schlimmer). Die Behandlung der GUIDs und der Kompatibilitätseinstellungen durch VB sind eine Katastrophe. Unter [10] wird eine Lösung durch ein eigenes Tool für dieses Problem vorgeschlagen, ein Kollege hat inzwischen mit Visual Make [11] ganz gute Erfahrungen gemacht. Unter anderem deswegen habe ich die Beispielkomponenten in drei Projekte zusammengefaßt. Wenn auf eine klare Funktionstrennung der einzelnen Klassen geachtet wird, dann können die Komponenten bei Bedarf sehr einfach zerlegt werden.

Die Entwicklung eines nachrichtenbasierten Informationssystems ist aufwendiger als die eines klassischen Client-Server Systems. Das liegt zum einen an der Asynchronität der Methodenaufrufe und zum anderen daran, daß die Programmierwerkzeuge nicht für eine solche Entwicklung ausgelegt sind. Ich hoffe jedoch, daß sich dies durch VStudio7 ändern wird. Als Lohn der Mehrarbeit winkt jedoch ein immer aktuelles und flexibles Informationssystem.

Ressourcen

- [1] Ted Pattison: Programming Distributed Applications with COM and Microsoft Visual Basic 6.0; Microsoft Press 1998
- [2] Christian Gross, Isabelle Laurin: Microsoft Message Queue mit VB nutzen; BasicPro 2/98, Seite 66
- [4] Marcel Gnoth: ChariTime-Systementwurf für ein verteiltes Multiagentensystem, Diplomarbeit, Humboldt Universität, Berlin, 2000, www.gnoth.net/ChariTime
- [5] ChariTime; Termin- und Informationsmanagement im Krankenhaus, www.charitime.de
- [6] Ralf Westphal: Persistente Objektnetzwerke; BasicPro 2/99, S 45
- [7] MSDN: Message Queuing Offline Support
- [8] Q175872-HOWTO: Sending Persistent Objects Using MSMQ in Visual Basic
- [9] Andreas Mey: MIME und base64; BP Wildsite 1/99
- [10] L.J.Johnson: Take Control of Your Build Cycle, VBPI December 2000
- [11] <http://www.mobi-sys.com/>

Schnellstart

Schritt 1: MSMQ Installation

Auf mindestens einem Computer MSMQ installieren. Die Installation unter Windows 2000 ist am einfachsten. Details siehe Abschnitt „Installation“.

Schritt 2: „Addresses.xml“ editieren

Im Verzeichnis „MSMQ Sample“ die Datei „Addresses.xml“ mit einem Editor öffnen und mindestens eine Adresse anpassen. Wichtig sind „HostName“, „HostIP“ und „QueueIsPrivate“. Setzen Sie „QueueIsPrivate“ erst mal auf „-1“.

Schritt 3: Komponenten registrieren

Die Komponente *OOModel.dll* im Verzeichnis „MSMQ Sample“ muß registriert werden. Sie enthält Klassen, die per XML verschickt werden. Der Empfänger packt diese mit *CreateObject* wieder aus, und *CreateObject* holt sich die ClassID aus der Registry. Benutzen Sie dafür *RegComponents.bat*.

Schritt 4: Testprojekt und Komponente öffnen

Im Verzeichnis „MSMQ Sample“ *Client.exe* starten oder die Projektgruppe „Client+Komponente.vbg“ öffnen und starten.

Schritt 5: eine Queue öffnen

Selektieren Sie die angepaßte Adresse und klicken Sie „OpenQueue“. Wenn alles geklappt hat, dann erscheint in der Titelzeile der Form der Pfad der Queue.

Setzen Sie jetzt ein **Häkchen in die Checkbox** der geöffneten Adresse und klicken Sie auf „SendPingMessage“. Sie haben gerade eine Testnachricht an sich selbst gesendet, es sollten vier Einträge erscheinen: Ping raus, Ping rein, Pong raus, Pong rein.

MSMQ Architektur

Der Message Queue Server wurde 1998 in der Version 1.0 mit dem NT Option Pack ausgeliefert. Mit Windows 2000 erschien die Version 2. Unter NT4 übernimmt ein sogenannter Primary Enterprise Controller (PEC) die Funktion eines zentralen Informationssystems (Message Queue Information Store, MQIS). Dadurch können Clients nach einer Queue suchen und diese über ihren Namen lokalisieren, sie müssen nicht wissen, auf welchem Rechner sich die Queue befindet. Der PEC übernimmt eine Reihe weiterer Aufgaben unter anderem das Routing von Nachrichten. Site Controller (SC) fassen die Clients eines LANs zusammen und übermitteln Nachrichten zwischen einzelnen LANs.

Es gibt Independent Clients mit einer eigenen Queue-Verwaltung und Dependent Clients, die permanent mit einem PEC oder SC verbunden sein müssen. Der PEC kann auch die Aufgaben eines Site Controllers und eines Independent Clients übernehmen. Independent Clients können auch Offline betrieben werden (Laptop). Ausgehende Nachrichten werden lokal gepuffert und nach dem Wiederherstellen der Verbindung abgesendet. [7]

Nachrichtenwarteschlangen (Queues) können Public (sind im MQIS eingetragen) oder Private sein. Um Nachrichten an Private Warteschlangen zu senden, muß deren genaue Adresse bekannt sein, da sie nicht im MQIS nachgeschlagen werden kann.

Für das Versenden von Nachrichten gibt es verschiedene Optionen, so können Nachrichten innerhalb von MTS Transaktionen versendet werden. Es gibt eine Journalfunktion, die darüber Buch führt, welche Nachrichten eingetroffen sind bzw. abgesandt wurden, und Nachrichten können Express (die Nachricht wird nur im Speicher gehalten, schneller) oder Recoverable (Nachricht wird auf der Festplatte gespeichert und erst bei Zustellung gelöscht, sicherer) gesendet werden.

Unter Windows 2000 werden PEC und SC durch das Active Directory abgelöst.

Kasten 1: MSMQ Architektur.

MSMQ Probleme

Während unserer Arbeit mit dem MSMQ sind wir auch auf einige Probleme gestoßen. So stand etwa eines Tages unser Server auf 100% und nur ein deinstallieren und neu installieren half. Einige Tage später trat das Problem erneut auf. Nach sehr langem Suchen in den News erfuhr ich zwar, daß es einige weitere MSMQ-Probleme gibt, fand aber keine Lösung.

Ich vermute, daß es mit privaten Warteschlangen und der Journalfunktion zusammenhängt. Als wir dann auf Public Queues umstiegen, trat das Problem nicht mehr auf.

Daraufhin bemühte ich den MS Support, der mir dann nach langer Zeit einen Hotfix für Win2k zur Verfügung gestellt hat, der für NT4 steht noch aus. Dieser HotFix „Q280088“ wird hoffentlich bald in der KnowledgeBase veröffentlicht. Was es nun genau war, wurde mir noch nicht mitgeteilt. Ein ähnliches Problem ist unter „Q269803“ beschrieben.

Auf alle Fälle sollten Sie immer darauf achten, die aktuellen Windows Service Packs zu installieren, wenn Probleme mit dem MSMQ auftreten.

Ein weiteres „lustiges“ Problem ist ein Counterüberlauf. Alle 49 Tage steht der Server auf 100%, da eben ein Counter übergelaufen ist. (Q256323)

Informationen finden Sie, in dem Sie unter msdn.microsoft.com in der KnowledgeBase nach MSMQ suchen oder in den Newsgroups unter msnews.microsoft.com/microsoft.public.msmq.programming.

Bei Problemen mit dem MSMQ geben Sie an der Kommandozeile „Net Stop MSMQ“ ein. Dadurch wird der Dienst gestoppt. Anschließend muß der MSMQ deinstalliert und neu installiert werden.

Kasten 2: Probleme, die während unserer Arbeit mit dem MSMQ auftraten.

Marcel Gnoth ist Senior Consultant bei der Berliner NTeam GmbH (www.nteam.de). Seine Arbeitsschwerpunkte liegen im Bereich COM und verteilte Informationssysteme. Neben Entwurf und Implementierung von Softwaresystemen gibt er Trainings im Bereich Windows DNA, VB und ASP. Sie erreichen ihn unter mgnoth@nteam.de oder www.gnoth.net.