

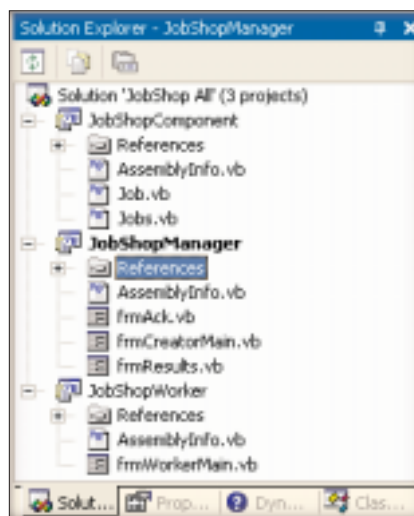
## MSMQ: Infrastruktur für den Nachrichtentransport

# Einschreiben mit Rückschein

Microsoft Message Queuing kann mehr als nur Nachrichten versenden und empfangen. Mit Timeout-Werten, dem Versenden von Empfangsbestätigungen und der Nutzung verteilter Transaktionen können Sie eine leistungsfähige Infrastruktur für den Nachrichtentransport aufbauen. Die Möglichkeiten von MSMQ gehen über das, was Web Services bieten, weit hinaus.

**D**er erste Teil der Serie hat eine Einführung in Microsoft Message Queuing und in das Versenden von Objekten gegeben [1]. Der vorliegende zweite Teil demonstriert anhand zweier Applikationen die Möglichkeiten von Message-Queuing-Applikationen und erläutert, wie die Applikationen zusammenarbeiten. Dazu gehören Journal- und Benachrichtigungsqueues, der Offline-Einsatz und vieles andere mehr.

Auf der Heft-CD befinden sich drei Beispielprojekte, ein *JobShopManager*, ein *JobShopWorker* und eine *JobShopComponent* – siehe Abbildung 1. Ein Manager erzeugt neue Aufgaben in einer Job Queue und ein oder mehrere Worker be-



**Abbildung 1** Der Manager erzeugt Aufgaben und erhält die Ergebnisse von den Workern.

arbeiten diese Aufgaben und senden die Ergebnisse wieder zurück in eine Result Queue. Die Worker können über mehrere Rechner verteilt sein und erledigen ihre Aufgaben über das Netzwerk, siehe Abbildung 2. Wie Sie die Beispielprojekte ausprobieren können, beschreibt der Kasten „Die Beispielprogramme einsetzen“.

### Das Message-Objekt

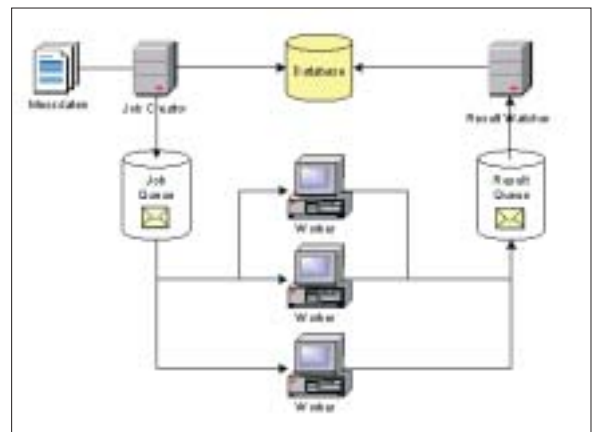
Im ersten Teil der Serie hat dotnetpro gezeigt, wie Nachrichten direkt über die *Send*-Methode des *Queue*-Objektes versendet werden. Für die Versendoptionen stellt jede Queue eine *DefaultPropertiesToSend*-Auflistung bereit, mit der die wichtigsten Parameter konfiguriert werden können. Das *Message*-Objekt bietet eine noch feinere Kontrolle für den Versand von Nachrichten.

Für den Empfang einer Nachricht ist ebenfalls ein *Message*-Objekt erforderlich. Es bietet Zugriff auf den Inhalt und auf viele Eigenschaften. Aber Achtung: Beim Lesen einer Nachricht aus einer Warteschlange werden nicht alle Eigenschaften standardmäßig gelesen. Über die *MessageReadPropertyFilter*-Eigenschaft kann dieses Verhalten gesteuert werden. Details dazu finden Sie in der Online-Hilfe.

### Timeout

Das Versenden einer Nachricht erfolgt asynchron, es gibt also keine Information darüber, ob die Nachricht eingetroffen ist oder gelesen wurde, genau wie bei einem Standardbrief der Post. Warteschlangen müssen nicht immer online sein, das ist ja einer der Vorzüge von MSMQ. Der Abschnitt über den Offline-Einsatz weiter unten bietet dazu genauere Informationen. Soll der Versende- oder der Lesevorgang zeitlich begrenzt werden, dann bietet sich der Einsatz von Timeouts an.

Das Setzen eines Timeouts erfolgt über die Eigenschaften *TimeToReachQueue* und *TimeToBeReceived*. Standard-



**Abbildung 2** Das Beispielszenario realisiert verteilte Transaktionen.

### Auf einen Blick

#### Autor



**Marcel Gnoth** ist Leiter der Entwicklungsabteilung bei der Berliner NTeam GmbH. Außerdem führt er Trainings für Entwickler durch und tritt regelmäßig als Sprecher auf der BASTA auf. Sie erreichen ihn unter [mgnoth@nteam.de](mailto:mgnoth@nteam.de) oder über [www.gnoth.net](http://www.gnoth.net).

**dotnetpro.code**  
A0404MSMQ



**Sprachen** VB.NET, C#

**Technik** Microsoft Message Queuing (MSMQ)

**Voraussetzungen** VS.NET, Windows XP oder Windows Server 2003

#### Serie

1. Mitteilungen veröffentlichen und lesen
2. Infrastruktur für den Nachrichtentransport



nach jedem Lesen eine Kopie der Nachricht in der Journal Queue abgelegt.

Achten Sie darauf, den Speicherplatz für Journal Queues zu beschränken, sonst wird MSMQ beim Start irgendwann sehr langsam und die Festplatte voll. Über die Computerverwaltung oder die *MaximumJournalSize*-Eigenschaft einer Queue können Sie die Größe der Journal Queue konfigurieren.

### Response Queue

Und dann gibt es noch eine Queue, diesmal wieder eine ganz „normale“: Vor dem Versenden einer Nachricht kann dem *Message*-Objekt eine Response Queue übergeben werden.

In Listing 1 ist das die lokale Result Queue. Der Empfänger der *Job*-Nachricht bearbeitet die Aufgabe und schickt dann die Antwort an die mit übertragene Adresse der Ergebniswarteschlange zurück.

MSMQ generiert für jede Nachricht – auch für *Acknowledge*-Nachrichten – nach dem Versand eine eindeutige ID, bestehend aus der Computer-GUID des sendenden Computers und einer eindeutigen Nachrichten-ID auf diesem Computer.

Bei *Acknowledge*-Nachrichten wird in der *CorrelationID*-Eigenschaft die *MessageID* der betroffenen Nachricht hinterlegt, für die die Benachrichtigung verschickt wurde. So kann der Computer, der die *Acknowledge*-Nachricht erhält, in seiner Journal Queue die betroffene Nachricht finden und erneut versuchen, die Nachricht zu versenden oder den Anwender über den Fehlschlag zu informieren.

Im Beispiel auf der Heft-CD werden alle erzeugten und versendeten *Job*-Objekte in einer *Jobs*-Auflistung gespeichert und bei jedem *Job*-Objekt wird nach dem Versenden die *MessageID* hinterlegt (Listing 1), damit später Benachrichtigungen diesem Job zugeordnet werden können.

Listing 2 demonstriert, wie auf das Eintreffen von Benachrichtigungen reagiert wird. Als Erstes wird geprüft, ob die eingetragene Nachricht vom Typ *Acknowledgment* ist. Wenn ja, wird die *CorrelationID*-Property ausgelesen. Mit dieser ID wird der betroffene Job und das dazugehörige *ListViewItem* gesucht. Dann wird das *SubItem* auf den Wert der Nachricht gesetzt. Dadurch erscheinen in der Tabelle in Abbildung 3 die Statusmeldungen.

Die Worker-Anwendung sendet die Ergebnisse zurück an den Manager zur angebenen Response Queue. Dabei könnte

### Listing 1

#### Zahlreiche Optionen steuern den Nachrichtenversand.

```
Private Sub CreateAndEnqueueJob()
    Dim s As String
    Dim j As JobShopComponent.Job = New JobShopComponent.Job(1, 88)
    m_Jobs.Add(j)
    sbpStatus.Text = "Versende Job " + j.Name
    sbpJobs.Text = "Jobs: " + m_Jobs.Count.ToString
    Dim theMsg As System.Messaging.Message = New System.Messaging.Message(j)
    theMsg.Label = "Cool " + j.Name + " " + Now.ToLongTimeString
    theMsg.AdministrationQueue = m_JobAckQueue
    theMsg.ResponseQueue = m_ResultQueue
    theMsg.AcknowledgeType = CType(1stAckType.SelectedItem, AcknowledgeTypes)
    theMsg.UseDeadLetterQueue = chkUseDeadLetterQueue.Checked
    theMsg.UseJournalQueue = chkUseJournalQueue.Checked
    If NumericUpDown1.Value > 0 Then
        theMsg.TimeToBeReceived = New TimeSpan(0, 0, CInt(NumericUpDown1.Value))
        theMsg.TimeToReachQueue = New TimeSpan(0, 0, CInt(NumericUpDown1.Value))
    End If
    m_JobQueue.Send(theMsg)
    j.MQMessageID = theMsg.Id 'save MessageID in Job-Object
    Dim li As ListViewItem = New ListViewItem(New String() {j.Name, "", "", ""})
    li.Tag = j.MQMessageID 'find ListViewItem by MessageID
    lvwJobs.Items.Add(li)
    sbpJobs.Text = "Jobs: " + m_Jobs.Count.ToString
End Sub
```

### Listing 2

#### Benachrichtigungen empfangen und korrekt reagieren.

```
Private Sub m_AckQueue_ReceiveCompleted(ByVal sender As System.Object, _
    ByVal e As ReceiveCompletedEventArgs) Handles m_JobAckQueue.ReceiveCompleted
    Dim Msg As System.Messaging.Message
    Dim j As Job
    Dim li As ListViewItem

    Msg = e.Message
    If Msg.MessageType = MessageType.Acknowledgment Then
        Dim cid As String
        cid = Msg.CorrelationId
        j = m_Jobs.ItemByMQMessageID(cid)
        li = GetListViewItemByMsgID(cid)

        If Not j Is Nothing Then
            'found the job of the Ack Msg
            Select Case Msg.Acknowledgment
                Case Acknowledgment.ReachQueue, Acknowledgment.ReachQueueTimeout
                    li.SubItems(1).Text = Msg.Acknowledgment.ToString
                Case Acknowledgment.Receive, Acknowledgment.ReceiveTimeout
                    li.SubItems(2).Text = Msg.Acknowledgment.ToString
                Case Else
                    li.SubItems(3).Text = Msg.Acknowledgment.ToString
            End Select
        End If
    End If

    'nächste Nachricht abgreifen
    m_JobAckQueue.BeginReceive()
End Sub
```

die Worker-Anwendung in der Antwortnachricht die *CorrelationID*-Eigenschaft auf den Wert der Jobnachricht setzen.

In der Beispielanwendung speichert der Worker die *MessageID* der Anfrage in der *MQMessageID*-Eigenschaft des empfangenen Job-Objektes. Nachdem der Job serialisiert mit der Result Message zum Manager zurückgesendet wurde, kann die ursprüngliche, anfordernde Nachricht leicht in der Journal Queue gefunden werden.

## Offline-Einsatz

Das Schöne an einer Programmkommunikation mit MSMQ ist, dass der Empfänger nicht online sein muss. Stellen Sie sich den klassischen Außendienstmitarbeiter vor, der abends im Hotelzimmer seine täglichen Verkaufserfolge in das Laptop eingeben möchte und sich nur gelegentlich ins Firmennetz einwählt. Seine Anwendung kann neue Auftragsobjekte erzeugen und versenden.

Alle Nachrichten, die nicht zugestellt werden können, weil eine Netzwerkverbindung fehlt, werden vom lokalen Queuemanager zwischengespeichert. In der Computerverwaltung – siehe Abbildung 4 – sehen Sie unter dem Ordner *Message Queuing* einen Ordner *Ausgehende Warteschlangen*. In diesem Ordner sehen Sie alle zurzeit nicht erreichbaren Warteschlangen und deren Nachrichten. Ist eine Netzwerkverbindung hergestellt, dann werden nach kurzer Zeit die Nachrichten automatisch zugestellt.

Die Ziel-Queue muss mit einem *DirectFormatName* geöffnet werden (Näheres dazu in [1]), Nachrichten müssen im Modus *Recoverable* versendet werden.

```
theMsg.Recoverable = True
```

In diesem Modus werden alle Nachrichten so lange auf der Festplatte des Computers gespeichert, bis sie an den Zielcomputer weitergesendet werden können. Dadurch überstehen die Nachrichten auch einen Neustart, aber der Versand wird durch den zusätzlichen Festplattenzugriff viel langsamer, als wenn Sie die Nachrichten im Express-Modus verschicken würden, der die Nachrichten nur im RAM zwischenspeichert.

## Transaktionen

Mit MSMQ und dem *Recoverable*-Modus können Nachricht sehr zuverlässig versandt werden, aber in einigen Situationen

## Listing 3

### MSMQ-interne Transaktion.

```
Dim mqTransX As New MessageQueueTransaction
mqTransX.Begin()

msgTest = mqReceive.Receive _
(New TimeSpan(0, 0, 3), mqTransX)
mqSend.Send("Ein Body...", "TX Msg " _
+ Now.ToLongTimeString, mqTransX)

If Now.Second Mod 2 = 0 Then
mqTransX.Commit()
Else
mqTransX.Abort()
End If
```

reicht das nicht aus. Jetzt kommen *Transaktionen* ins Spiel. Jeder Sendevorgang und jeder Lesevorgang kann innerhalb einer Transaktion ablaufen. Wenn ein Rollback durchgeführt wird, dann werden auch diese Vorgänge rückgängig gemacht. Wichtig ist, dass diese Nachrichtenwarteschlangen beim Erzeugen als *transactional* gekennzeichnet werden. Details dazu finden Sie in der Dokumentation zur Methode *MessageQueue.Create*.

Stellen Sie sich vor, Ihre Applikation liest Einträge aus einer Bestell-Queue, verarbeitet diese und schreibt Aufträge in Queues der Fachabteilungen. Tritt

zwischen durch ein Fehler auf, dann kann es passieren, dass eine Bestellung gelesen, also aus der Queue entnommen wurde, aber die Aufträge nicht vollständig generiert und versandt wurden. Dadurch können Inkonsistenzen im System auftreten.

Es gibt zwei verschiedene Möglichkeiten, Nachrichten transaktional zu versenden. MSMQ bietet einen eigenen Mechanismus an, der schneller ist, aber nur MSMQ-Vorgänge einschließen kann [2]. Message Queuing kann aber auch als Ressourcen-Manager auftreten und an verteilten Transaktionen teilnehmen, die vom Distributed Transaction Coordinator (DTC) koordiniert werden. Das ist zwar langsamer, aber an solch einer Transaktion kann auch ein SQL Server oder jeder andere Ressourcen-Manager teilnehmen [3].

## MSMQ-interne Transaktionen

Listing 3 zeigt ein Beispiel für eine MSMQ-Transaktion. Dafür wird ein *MessageQueueTransaction*-Objekt erzeugt und bei jedem Sendevorgang mit übergeben. Setzen Sie einen Breakpoint im Modul *frmCreatorMain* in der Methode *cmdMSMQTX\_Click* und beobachten Sie während der Ausführung den Inhalt der Warteschlangen in der Computerverwaltung. Verwenden Sie *Refresh* aus dem Kontextmenü, um die Anzeige zu aktualisieren.

## Listing 4

### Eine Transaktion mit einer COM+-Komponente.

```
public class cUhura : ServicedComponent {private MessageQueue m_MQOrders;
private string m_MQOrdersPath = @".\private\TXOrders";
public cUhura() { } //empty Default Constructor

public string SendOrderToQueue() {m_MQOrders = new MessageQueue(m_MQOrdersPath);
if (!m_MQOrders.Transactional) {
throw new System.ApplicationException("Queue need to be transactional!");
}
try {
Message mqMsg = new Message();
mqMsg.Body = "A lot of Tu Du!";
mqMsg.Label = "Order " + DateTime.Now;
m_MQOrders.Send(mqMsg, MessageQueueTransactionType.Automatic);

//Execute here some Database operations, connections will
//enlist automatically in the transaction ContextUtil.SetComplete();
}
catch (Exception e){ContextUtil.SetAbort();
}
return "Done msmq";
}
}
```

Die *Receive*-Methode entnimmt erst einmal die Nachricht aus der Queue, während die Nachricht, die über die *Send*-Methode versendet wird, noch nicht in der Zielqueue auftaucht. Erst wenn *Commit* aufgerufen wird, erscheint die gesendete Nachricht. Wird *Abort* aufgerufen, kehrt die Nachricht unverändert in die Receive Queue zurück.

Auf diese Art können Sie Nachrichten aus einer Warteschlange entnehmen, verarbeiten und Antworten in eine Ergebnis-Queue stellen. Tritt ein Fehler auf, werden beide Vorgänge rückgängig gemacht. Leider scheint dieser Mechanismus nicht mit dem asynchronen Empfangen von Nachrichten zu funktionieren.

Der *BeginReceive*-Methode kann als Parameter kein Transaktionsobjekt übergeben werden.

Das *MessageQueueTransaction*-Objekt hat eine *Status*-Eigenschaft, die anzeigt, ob die Transaktion noch läuft, abgeschlossen ist oder bestätigt wurde.

### Transaktionen mit DTC und COM+

In der Praxis reicht das bloße Versenden von Nachrichten meist nicht aus. Oft müssen auch noch Daten in eine Datenbank eingetragen werden. Jetzt kommen

COM+ und der DTC ins Spiel. In einer Komponente, die von *System.EnterpriseServices.ServicedComponent* erbt und im COM+-Katalog eingetragen ist, können Nachrichten versendet und Datenbankoperationen durchgeführt werden. Die Koordination zwischen den verschiedenen Ressourcenmanagern MSMQ, SQL Server oder Oracle übernimmt dabei der DTC, der bei einem Commit oder Abort dafür sorgt, dass alle Ressourcenmanager informiert werden und ihre Aktionen speichern oder zurückrollen.

Listing 4 zeigt das Versenden einer Nachricht in einer DTC-Transaktion. Eine ausführliche Beschreibung finden Sie in [3].

### Fazit

MSMQ bietet dem Entwickler viele Möglichkeiten, um verteilte Applikationen oder eine Workflow-Anwendung zu erstellen. In der Version 3.0 sind einige neue Elemente wie Trigger, Multicast-Nachrichten und der Zugriff auf Warteschlangen über HTTP hinzugekommen, die Bestandteil eines weiteren Artikels sein werden. Dadurch werden die Grenzen zwischen Web Services und MSMQ fließend.

Da MSMQ nahtlos in Windows integriert ist, bietet es einen viel größeren Umfang an Funktionalitäten als Web Services. Wird MSMQ aber über HTTP eingesetzt, so fallen viele dieser Vorteile weg. Trotzdem sollten Sie nicht immer gleich an Web Services denken, nur weil diese gerade modern sind. MSMQ stellt im Intranet eine mächtige Alternative zur Verfügung. Die Anbindung an andere Messaging-Systeme, wie IBM MQSeries, ist ebenfalls möglich. |||||

- [1] Marcel Gnoth, Auf die Post ist Verlass, Microsoft Message Queue Service: Mitteilungen veröffentlichen und lesen, dotnetpro 3/2004, Seite 114 ff.
- [2] Duncan Mackenzie, Reliable Messaging with MSMQ and .NET, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnda/html/bdadotnetasync2.asp>
- [3] Marcel Gnoth, Einer für alle, alle für einen: verteilte Transaktionen, dotnetpro 6/2003, Seite 90 ff.
- [4] Bernd Marquardt, Multiple Threading mit .NET, Teil 1, dotnetpro 6/2002, Seite 103 ff.
- [5] Bernd Marquardt, Multiple Threading mit .NET, Teil 2, dotnetpro 3/2003, Seite 88 ff.

## Die Beispielprogramme einsetzen.

Das *Worker*- und das *Manager*-Programm können auf dem gleichen oder auf verschiedenen Computern eingesetzt werden. Die Computer können in einer Domäne oder in einer Arbeitsgruppe sein, wichtig ist, dass Sie sich mit dem gleichen Konto anmelden. Ist das nicht möglich, dann müssen Sie bei Problemen mit dem Versand oder dem Empfang von Nachrichten die Sicherheitseinstellungen der betroffenen Queues prüfen. Das .NET Framework 1.1, MDAC und MSMQ müssen installiert sein, siehe [1].

Starten Sie als Erstes auf einem Computer den Worker und geben Sie in der Computer-Textbox den Namen des lokalen Computers ein. Durch Anklicken von *Open Queue* wird die Job Queue lokal angelegt. Weitere Worker können gestartet werden, auf dem gleichen Rechner oder auf einem anderen. Geben Sie jeweils den gleichen Computer- und Queue-Namen ein. Als Nächstes starten Sie den Manager (siehe Abbildung 1) und geben in der obersten Textbox den Pfad zur Job Queue an. Hier müssen Sie die *FormatName*-Syntax ver-

wenden, siehe [1]. Hinter OS: folgt der Name des Computers, auf dem Sie die Job Queue erzeugt haben. In den beiden darunter folgenden Textboxen ersetzen Sie *Chihiro* durch den Namen des Computers, auf dem der Manager gestartet wurde. Klicken Sie dann auf *Create/Open Queues*. Die Result Queue und die Acknowledge Queue werden angelegt und geöffnet, die Verbindung zur Job Queue wird hergestellt.

Durch einen Klick auf *Create Jobs* werden Nachrichten erzeugt und zur Job Queue versandt. Jetzt klicken Sie beim Worker auf Start und der oder die Worker beginnen mit der Bearbeitung der Nachrichten.

Im Formular *frmCreatorMain* werden neue Nachrichten erzeugt und zugleich wird auf Ereignisse der Acknowledge Queue und der Result Queue gewartet. Werden neue Jobs generiert, dann erfolgt das in einer Schleife, die nach jeder Iteration *DoEvents* aufruft. Dann können die Events der beiden Nachrichtenwarteschlangen empfangen werden. Es handelt sich um eine Windows-Forms-Anwendung mit nur einem

Thread, der auch gleichzeitig noch für die Interaktion mit dem Anwender verwendet wird. Wenn Sie die Anwendung ausprobieren, kann es zu Multithreading-Problemen kommen. Auch die Prozessorlast durch das Erzeugen neuer Nachrichten in der Schleife ist zu hoch. Ein besserer Ansatz wäre es, wenn die Jobs in einem neuen Thread erzeugt würden, der unabhängig von der Hauptanwendung läuft. Auch das Überwachen der Warteschlangen könnte in eigenen Threads erfolgen. Aber dann müssen alle Zugriffe der drei Queue-Threads auf die Oberfläche der Anwendung synchronisiert werden. Das ist nicht trivial und wurde schon früher in dotnetpro beschrieben (siehe [4] und [5]).

Die Probleme lassen sich sehr leicht umgehen, indem Sie für das Erzeugen der Jobs und das Überwachen der Warteschlangen jeweils eigene Projekte anlegen, die dann in eigenen Prozessen laufen. Durch MSMQ ist es sehr einfach, aus der Manager-Applikation drei eigenständige Applikationen zu machen, da die Anwendungen über Nachrichten kommunizieren und sehr schön entkoppelt sind.