

BizTalk Server 2002 Design and Implementation

XIN CHEN

Apress™

BizTalk Server 2002 Design and Implementation
Copyright ©2003 by Xin Chen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-034-1

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Gordon Mackie
Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Sofia Marchant
Development Editor: Andy Carroll
Copy Editor: Ami Knox
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Nancy Guenther
Cover Designer: Kurt Krames
Production Manager: Kari Brooks
Manufacturing Manager: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

Chapter 15 BizTalk Server with Microsoft	
Operations Manager	559
Appendix A Microsoft Message Queue	607
Appendix B BizTalk Server API Reference	645
Index	657

Advanced BizTalk Orchestration Features

In **CHAPTER 9**, you learned about some advanced features of BizTalk Messaging. In this chapter, I'll show you the many advanced features of BizTalk Orchestration: orchestration transactions, XLANG Schedule throttling, orchestration correlation, Web services in BizTalk Orchestration, orchestration dehydration, and so on. You'll continue to update Bob's ASP as you explore these features, because Bob always wants to add to his application.

Using Orchestration Transactions (Short, Long, and Timed)

In Chapter 5, you created an orchestration that processes the FundInvestors document. You used a Decision shape to determine whether your COM+ component completed successfully, but because the orchestration didn't have any transactional support, the actions that were performed couldn't be rolled back in the event of failure. Bob wants Mike to build a trading workflow that can process the buy and sell orders for the mutual fund companies, but he wants the trading XLANG Schedule to support transactions so that when an error occurs, any changes that have been made will automatically be rolled back to maintain the integrity of the data in the system. Let's take a quick look at transactions before tackling this assignment.

A *transaction* is a group of operations that are processed as a single unit. In the computer world, transactions follow these four rules, collectively known as the ACID rules:

- *Atomicity*: A transaction represents an atomic unit of work. Either all modifications within a transaction are performed, or none of the modifications are performed. In other words, if one job in the transaction fails, the work performed by the previous jobs in that transaction should be reversed as if nothing happened.
- *Consistency*: When a transaction is finished, whether committed or aborted, all the data must be in the proper state, with all the internal rules and relationships between data being correct.

- *Isolation:* Modifications made by one transaction must be isolated from the modifications made by other concurrent transactions. Isolated transactions that run concurrently will perform modifications that preserve internal database consistency exactly as they would if the transactions were run serially. In other words, all the data related to an ongoing transaction should be locked to prevent other transactions from interfering with this transaction.
- *Durability:* After a transaction has been committed, all modifications are permanently in place in the system. The modifications persist even if a system failure occurs. In other words, after the data in the transaction has been committed, any system failure shouldn't change the committed data in any way.

Types of Transactions

In BizTalk Orchestration, you can make your Action-shape implementations participate in a transaction in one of two ways: Run your XLANG Schedule as a traditional transactional COM+ component or as an XLANG Schedule transaction that is specific to BizTalk Orchestration.

When making an XLANG Schedule run as a COM+ component, external COM+ components can start such a XLANG Schedule as if it were a COM+ component. The transactional behavior of this type of XLANG Schedule is the same as a COM+ component.

To configure an XLANG Schedule to run as a COM+ component, open the properties page of the Start shape at the top of the workflow, and for the Transaction model field, select “Treat the XLANG Schedule as a COM+ Component” (see Figure 10-1).

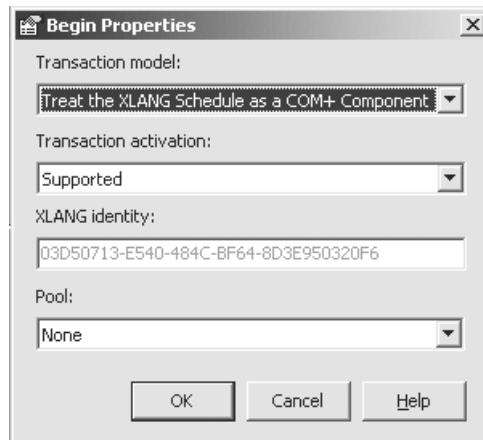


Figure 10-1. Configuring the transaction model for the orchestration

There are four transaction activation options available for this type of XLANG Schedule (which are the same as for COM+ components):

- *Not supported:* The schedule won't participate in the transaction. When the creator of the schedule aborts the transaction, the change made by the schedule won't be rolled back.
- *Supported:* The schedule will participate in the transaction of its creator, but it won't create one if its creator doesn't participate in a transaction. If its creator participates in a transaction, an abort in its creator will roll back all the actions performed inside the schedule.
- *Required:* The schedule will participate in its creator's transaction. If its creator doesn't participate in a transaction, the schedule will create a new transaction and make all the actions within the schedule participate in this transaction. An abort in its creator will roll back all the actions performed inside the schedule.
- *Required new:* The schedule will create a new transaction regardless of whether its creator already has a transaction. All the actions inside the schedule will participate in this new transaction created by the schedule. An abort in its creator won't roll back all the actions performed inside the schedule.



NOTE When you choose to treat the schedule as a COM+ component, you can't use the Transaction shape on the schedule, or an error will occur when compiling the schedule. The Transaction shape is exclusively used when choosing the other transaction model ("Include transaction in the XLANG Schedule").

The other transaction model for the schedule is "Include transaction within the XLANG Schedule". When selecting this transaction model, the actions within the schedule can participate in a transaction by locating them inside a Transaction shape. The four transaction activation types are also available for this transaction model.

When it comes time to choose which transaction model to use for your schedule, there are a number of factors to consider. In terms of performance, running the schedule as a COM+ component is much faster than using the other model. However, when running as an XLANG-style transaction, your schedule will be able to support Distributed Transaction Coordinator (DTC) style transactions, long-term transactions, timed transactions, and nested transactions. I'll discuss these transaction types later in this chapter. The COM+ transaction model only supports DTC-style transactions.

In fact, the Transaction shapes aren't even allowed in an orchestration unless the orchestration's transaction model setting is "Include Transaction within the XLANG Schedule" on the Begin shape. If the orchestration isn't configured this way and contains transactions, it will cause a compile error when you try to compile it into an .skx file.



NOTE DTC provides services to manage transactions that span different transactional sources. COM+ relies on DTC for its transactional support. For example, by using DTC, a COM+ component can write to a database and MSMQ, two separate transactional resources, in a single transaction.

With XLANG-style transactions, your schedule can also define actions to be taken in event of a transaction failure. This feature isn't available in COM+-style transactions. A schedule with XLANG-style transactions is shown in Figure 10-2.

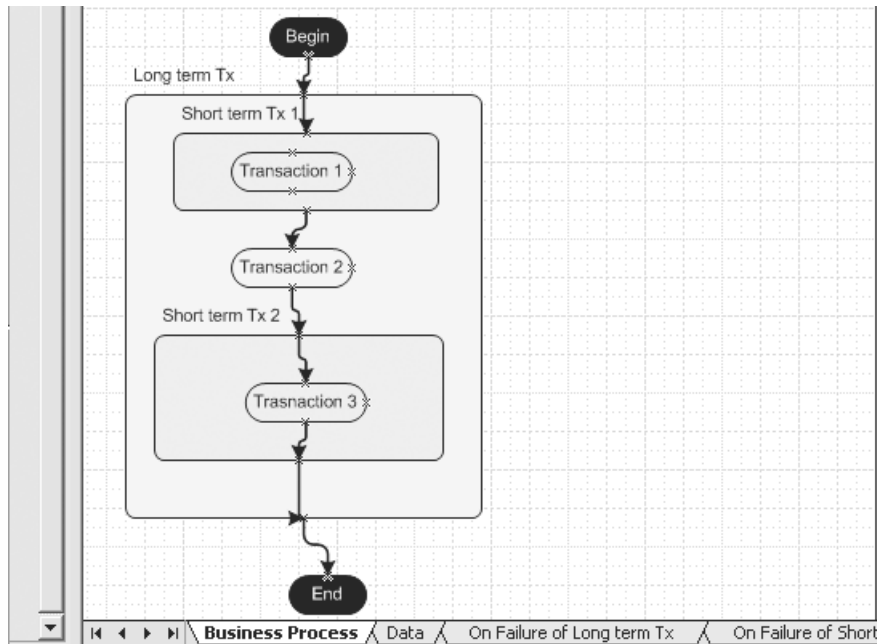


Figure 10-2. A schedule with XLANG-style transaction

To create a transaction inside another transaction, you need to first drag and drop a Transaction shape from the flowchart panel onto the Business Process page (which creates a new transaction), and then drag another transaction into the boundary of the first Transaction shape. You must make sure that all four sides of the second (inner) Transaction shape fit inside the four sides of the first (outer) Transaction shape, or the inner transaction won't be considered nested inside the outer one. To verify whether the transaction is nested properly, you can drag the outer Transaction shape around a bit—if the inner Transaction shape moves with the outer one, the inner transaction is properly nested. You can resize the Transaction shapes so that one can fit inside another.



NOTE If you delete the outer Transaction shape, the inner transaction, along with the shapes inside it, is deleted as well. If you want to keep the inner Transaction shape, make sure you move the inner transaction outside of the boundary of the outer one, unnesting the inner one, before deleting the outer transaction.

Short-Lived Transactions

Let's come back to Bob's trading XLANG Schedule. He wants the schedule to perform the actions shown in Figure 10-3.

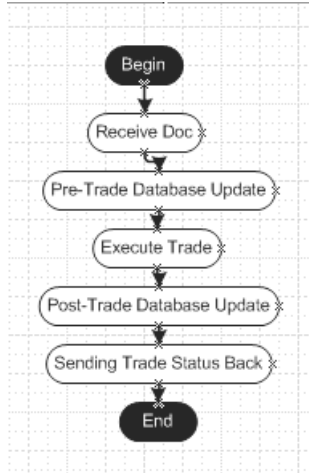


Figure 10-3. Actions involved in Bob's trading schedule

Five actions are involved in this trade schedule:

- *Receive Doc*: This action receives the document from the client.
- *Pre-Trade Database Update*: This action will update the database to indicate that the buy and sell transactions inside the client document are being processed. The information in this database will be exposed to the Internet so that each client can check the status of his or her trade orders.
- *Execute Trade*: This action will start Bob's existing trading component, which will execute the trade order and return a document that contains information on the transaction, such as fulfillment of the trade, final price per share, etc.
- *Post-Trade Database Update*: This action will take the document returned from the trading component and update the status information on the client's trade order in the database.
- *Sending Trade Status Back*: This action will send the document returned from the trading component back to the client.

Bob already has a document specification for this schedule, named Trade, defined in XML editor (see Figure 10-4).

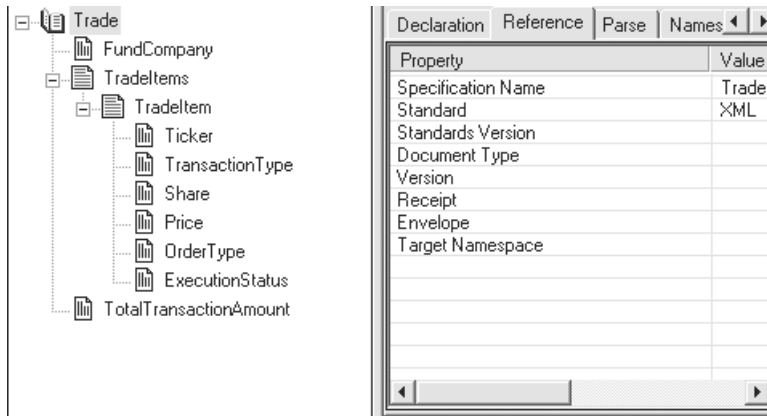


Figure 10-4. Trade document specification

A sample trade from a client would look like this:

```
<Trade>
  <FundCompany>Henry Fonda Inc.</FundCompany>
  <TradeItems>
    <TradeItem>
      <Ticker>IBM</Ticker>
      <TransactionType>Buy</TransactionType>
      <Share>10000</Share>
      <Price/>
      <OrderType>Market Order</OrderType>
      <ExecutionStatus/>
    </TradeItem>
    <TradeItem>
      <Ticker>MSFT</Ticker>
      <TransactionType>Buy</TransactionType>
      <Share>15000</Share>
      <Price>65.50</Price>
      <OrderType>Limit Order</OrderType>
      <ExecutionStatus/>
    </TradeItem>
  </TradeItems>
  <TotalTransactionAmount/>
</Trade>
```

The <ExecutionStatus> field will contain the execution information for each individual stock transaction in the document. Its content will be filled by Bob's trading component, and it will contain information on whether a specific transaction completed or not. For example, if the limit price isn't reached, the transaction won't take place; in such a case, the trading component will insert "Limited price is not met" in the field to indicate that this particular order was not processed. The trading component will also fill the <TotalTransactionAmount> and <ExecutionStatus> fields under each TradeItem record. The TotalTransactionAmount is the total dollar amount for each individual trade that is completed in the document.

In this trade schedule, you want to use XLANG-style transactions. You'll use short-term transactions in this schedule. Later in this chapter, I'll show you how to work with long-term and timed transactions, but for now, let's find out what advantages can be gained by making actions transactional.

The Trade Schedule

The Business Process page for Bob's trade schedule using transactions is shown in Figure 10-5. There are still five Action shapes in this schedule, but four of them are inside two short-term transactions. The Pre-Trade Database Update and Execute Trade actions are in the Trade Proc transaction. The Post-Trade Database Update and Sending Trade Status Back actions are in the Post-Trade Proc transaction.

Short-term transactions are also known as *DTC-style transactions*. These follow the same rules as DTC transactions in COM+. For instance, if the Execute Trade process aborts the transaction, any changes made by the Pre-Trade Database Update action will be automatically rolled back. After the transaction is committed, the changes made by both actions will be committed.

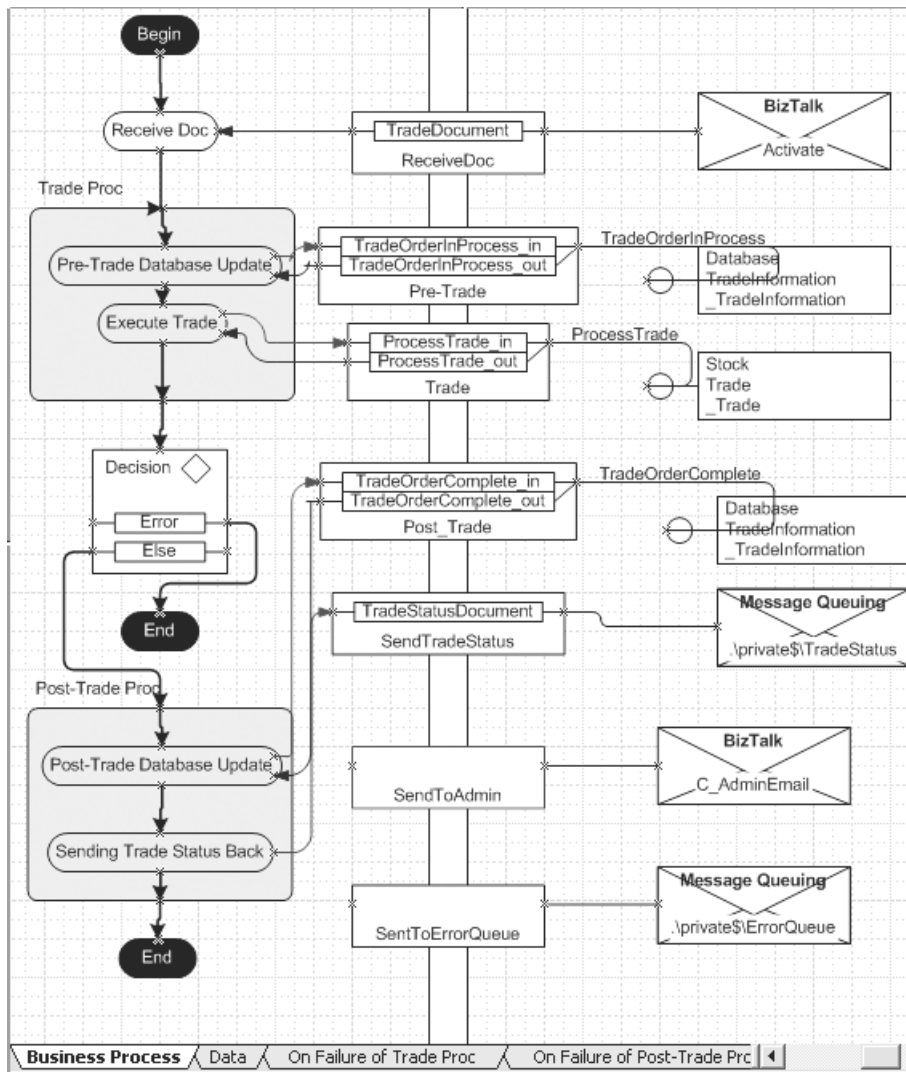


Figure 10-5. Business Process page of an orchestration with short-term transactions

Before you set up the properties for the Transaction shapes and specify what they should do in the event of failure, you must first define what you want the schedule to do. When a trade document enters the Trade Proc transaction and arrives at the Pre-Trade Database Update action, the TradeInformation COM+ component will be kicked off, and will update Bob's database with the order information in the document. The component will set the status in the database to In Process for each trade listed in the document. This means that if the client goes to the Web site, he or she will see the updated status for his or her outstanding buy and sell orders. If the TradeInformation COM+ component fails for whatever reason, the first thing you want to do is send an e-mail to the administrator to inform that person of this failure.

The next action is Execute Trade. This is the step that actually processes the document using Bob's trading component. The trading component is a COM+ component that will examine each individual trade in the document to verify whether the transaction is compliant with SEC rules and whether there are sufficient funds to make the transactions happen. If all the information in the document is correct, the component processes the orders, fills the status fields for the individual transactions along with the TotalTransactionAmount field, and returns the trade document. If a problem exists with the information in the document, the trading component will abort the transaction and end the current schedule, since there is no reason to continue. In this case, the database update will be rolled back, and you need to make another call to the TradeInformation component to mark the client's submission with the message "Contains errors, unable to process", and you must send an e-mail to the administrator and store the client document in the error queue for later examination. After all this work is completed, the schedule will end.

Unfortunately, there is no automatic way to terminate a whole schedule when a transaction aborts. When this happens, the schedule simply starts the process on its On Failure page and returns to the main business process after the actions on the On Failure page are performed. In some situations you'll want to terminate the schedule altogether if an important transaction fails, such as if the trading component fails—in that case, you want to perform the actions on the On Failure page and then terminate the schedule.

To achieve this goal, you can use a Decision shape to check whether or not the transaction has aborted by evaluating the value of `ProcessTrade_out.[__Exists__]`. `ProcessTrade_out` is the message that contains the output of the trading component. If the transaction succeeds, `ProcessTrade_out.[__Exists__]` will return true. If the transaction aborts, `ProcessTrade_out.[__Exists__]` will return false. Based on this test in the Decision shape, you can either continue the process flow or end it.

If the process flow enters the Post-Trade Proc transaction, the Trade Proc transaction has been successful and committed. It's now time to update the database with the new trade status information of each individual trade—the In Process status will be changed to whatever is in the status field of the document. If a client goes to Bob's Web site, they will find the status information for each individual trade he or she has sent. If this Post-Trade Database Update fails for whatever reason, you want to first send an e-mail to the administrator and then terminate the schedule. However, all the transactions have been committed at this point, so the administrator will need to inform the client that his or her transactions have been committed.

If the Post-Trade Database Update transaction is processed successfully, you want to send the document that is returned by the trading component to the client so that he or she will have a digital copy of this document. You can do this by sending the response document to a message queue to be picked up and forwarded later by BizTalk Messaging. If the schedule is unable to write the document to the message queue, it will roll back the database update that the Post-Trade Database Update transaction made, and it will send out an e-mail to the administrator to inform him or her of the situation.

Implementing the Transactions

To implement the transactions described in the previous section, you must set the transaction model of this schedule to “Include Transaction with XLANG Schedule” on the Start shape's properties page (refer back to Figure 10-1).

Next, double-click the Trade Proc transaction to open its Transaction Properties page (see Figure 10-6).

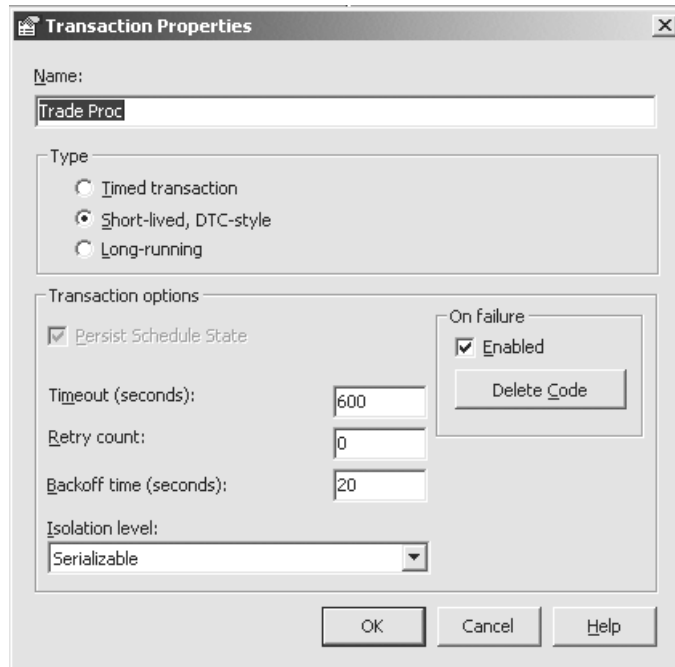


Figure 10-6. Transaction Properties page for the Trade Proc transaction

In the Type section of the Transaction Properties page, select Short-lived, DTC-style. Under the Transaction options section are three properties:

- *Timeout (seconds)*: This specifies how long the transaction may run before it's aborted or retried. This option is available for timed and short-lived, DTC-style transactions.
- *Retry count*: This specifies how many times the transaction can be retried if it can't be completed before the timeout.
- *Backoff time (seconds)*: This specifies how long the schedule will wait before continuing with the next retry. The actual wait time depends on the backoff time and the retry number: The formula is $\text{wait time} = B^R$ (B to the Rth power, where B equals the backoff time in seconds and R equals the number of retries). This option is available only for short-lived, DTC-style transactions.

The time it takes the trading component to complete depends on many factors, such as the size of the transaction and the activity on the stock market. You'll set the timeout time to 10 minutes (600 seconds) for this transaction. Set the retry

count to zero, since you don't want to execute the transaction again if it times out. The backoff time is irrelevant when you set the retry count to zero.

The Isolation level setting allows you to configure how this transaction will lock the resources it uses when it's running. There are four isolation levels:

- *Uncommitted read:* A transaction set with this isolation level will be able to read data regardless of whether the data is committed. For example, if a first transaction is updating a record but hasn't yet committed the change, a second transaction (with a setting of Uncommitted read) won't care about whether the first transaction will commit the change or abort. It simply reads what it sees in that record at that moment. This type of transaction is very fast, since it always gets the data it wants right away, but the data it reads may not be consistent, since it could read data that is rolled back later.
- *Committed read:* A transaction set with this isolation level will only read data that has been committed by other transactions. If data has been changed by another transaction but not yet committed, this transaction will wait until the data it wants to read has been committed. This type of transaction won't lock the data it reads, so other transactions could change the data before this transaction ends. This may cause inconsistencies if the transaction updates the data and assumes that the data is the same, because another transaction may have changed it. This type of transaction is slower than the uncommitted read, since it has to wait for all the uncommitted data to clear.
- *Repeatable read:* A transaction set with this isolation level locks all the data reads so that it can't be changed by other transactions until the repeatable read transaction is finished. This solves the problem with committed reads, but it's slower than a committed read because it has to lock all the data it reads, not just the data it updates. Repeatable reads are also slow in the sense that other transactions won't be able to update the data locked by the repeatable read transaction until the repeatable read transaction finishes.
- *Serializable:* A transaction set with this isolation level will lock the table to prevent inserts, updates, and deletes. This means that everything this transaction sees and doesn't see at the beginning of the transaction will remain exactly the same at the end of the transaction, except for changes made by the transaction itself. One problem with repeatable reads is that they don't prevent another transaction from inserting new data, which may cause some inconsistency in calculations that rely on information such as the total number of records—the Serializable isolation level solves this problem. Because this isolation level forces each transaction to be carried out and committed one at a time, it's the most restrictive of the four isolation levels, and it's the only isolation level that meets the ACID criteria.

For this example, choose the Serializable isolation level, which is the default setting, for the Trade Proc transaction. In the On failure section, check the Enabled checkbox, and then click the Add Code button to add a Business Process page that will be triggered when the transaction fails.

When you click the Add Code button, Orchestration Designer will add a new tab at the bottom of the window called On Failure of Trade Proc, to indicate that a new Business Process page exists for the transaction Trade Proc (see Figure 10-7).



Figure 10-7. A new On Failure page for the Trade Proc transaction

This On Failure page is similar to the regular Business Process page—here you can define Action shapes and implementation shapes. The process on the On Failure page starts right after the transaction it's associated with aborts. You have some idea of what you want to do in the case of the transaction aborting, so let's define those actions on the On Failure of Trade Proc page (see Figure 10-8).

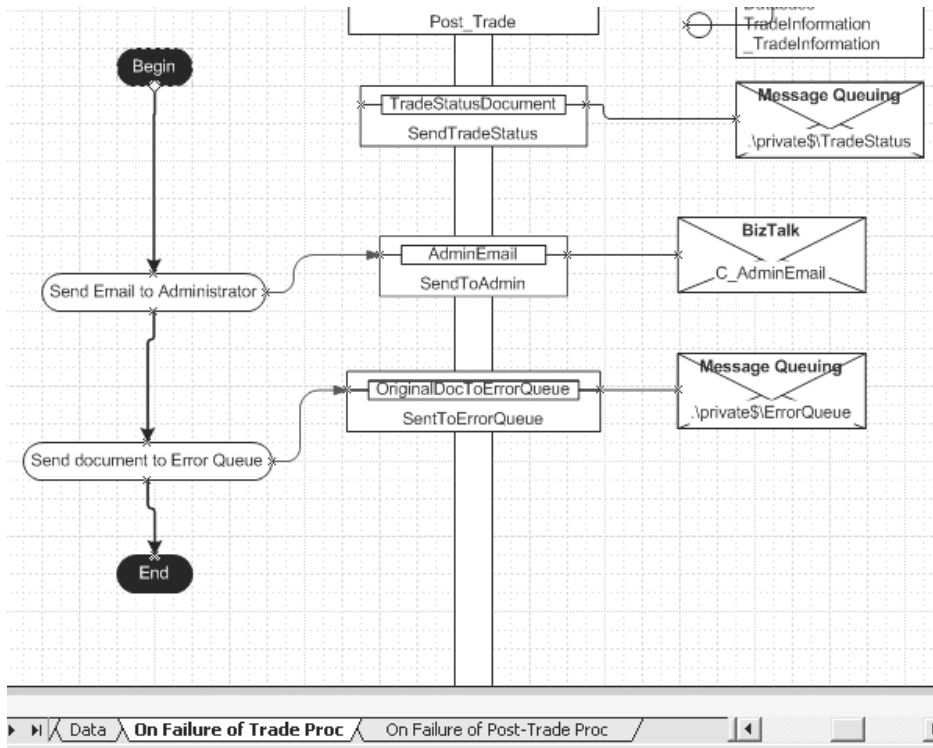


Figure 10-8. The On Failure of Trade Proc schedule

On the On Failure page, all the implementations on the Business Process page will still be available and can be linked with the Action shapes defined on this page. It's perfectly all right to create new transactions by dropping Transaction shapes on the On Failure page, and creating another On Failure page for this On Failure page. As far as the orchestration is concerned, the On Failure page is just another Business Process page.

In this case, you first want to send an e-mail to the administrator, so you need to implement a BizTalk Messaging shape on the page and link it with the Send Email to Administrator implementation. Next, you want to send the original document to a message queue so that the administrator can examine it later. (All the messages created on the Business Process page are still available in the BizTalk Messaging Binding Wizard.)

You've now completed the On Failure of Trade Proc page. The actions on this page will be processed when the transaction aborts. But what makes a transaction abort?

Many conditions can make a transaction abort, such as the following:

- The Abort shape is encountered within the process flow.
- A COM+ component or Scripting component returns a failure HRESULT (these components need to be configured to abort a transaction if a failure HRESULT is returned during the port binding).
- An abort transaction is called inside a COM+ or scripting component.
- A failure is introduced by a binding technology at a system level (for example, Message Queuing might fail to put a message on a queue).
- The XLANG Scheduler Engine (the COM+ application that executes instances of schedules) encounters an error (such as a DTC error) that causes it to abort a transaction within a given instance.
- A schedule is paused (this might require all transactions within that schedule to abort).
- A transaction time-out within the transaction properties.

To configure a component to abort the transaction when it causes a failure HRESULT, open the component's Binding Wizard (see Figure 10-9). Under the Error handling section, select the sole checkbox option so that the transaction this component is in will automatically abort when a bad HRESULT is returned from this component. Under the Transaction support section, you must select Supported to make this component participate in the existing transaction.

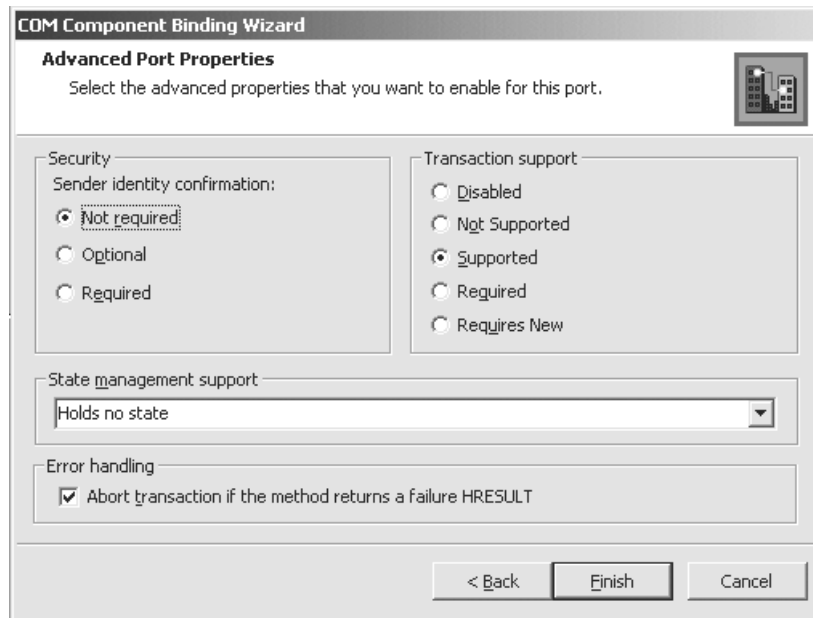


Figure 10-9. COM Component Binding Wizard

You also need to configure these properties for the TradeInformation and Trade components inside the Trade Proc transaction.

Next, you need to create an On Failure page for the Post-Trade Proc transaction (see Figure 10-10).

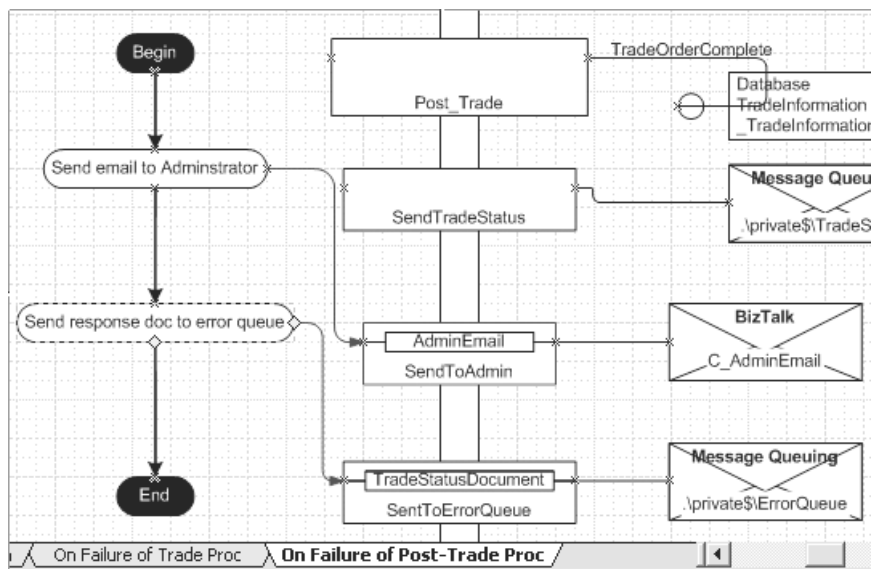
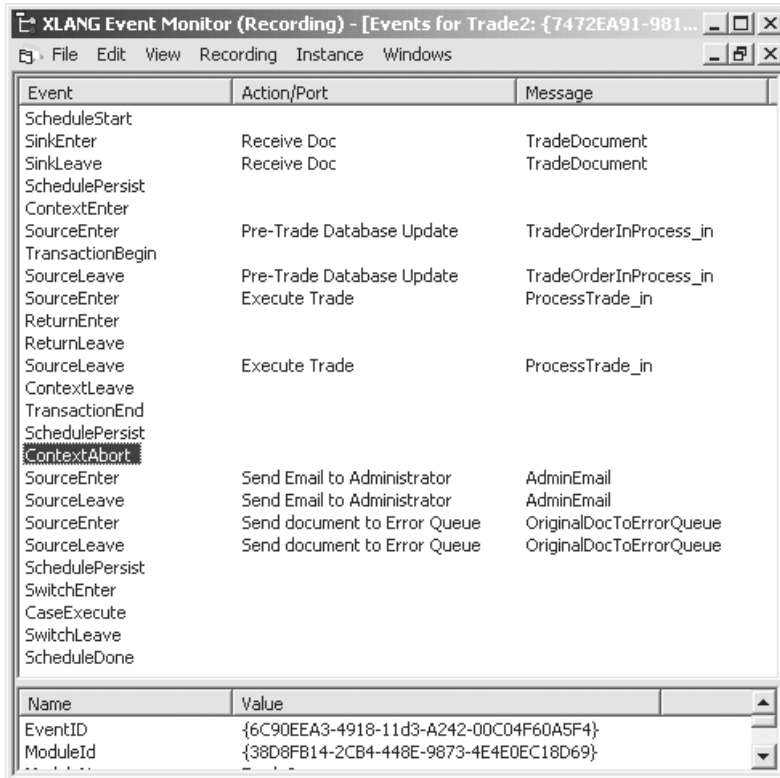


Figure 10-10. On Failure page for the Post-Trade Proc transaction

To test whether the On Failure page really works, let's raise an error in the trading component. Because the trading component is configured to abort the transaction when it encounters a bad HRESULT, you can watch what action will be taken in such a case in the XLANG Event Monitor (see Figure 10-11).



Event	Action/Port	Message
ScheduleStart		
SinkEnter	Receive Doc	TradeDocument
SinkLeave	Receive Doc	TradeDocument
SchedulePersist		
ContextEnter		
SourceEnter	Pre-Trade Database Update	TradeOrderInProgress_in
TransactionBegin		
SourceLeave	Pre-Trade Database Update	TradeOrderInProgress_in
SourceEnter	Execute Trade	ProcessTrade_in
ReturnEnter		
ReturnLeave		
SourceLeave	Execute Trade	ProcessTrade_in
ContextLeave		
TransactionEnd		
SchedulePersist		
ContextAbort		
SourceEnter	Send Email to Administrator	AdminEmail
SourceLeave	Send Email to Administrator	AdminEmail
SourceEnter	Send document to Error Queue	OriginalDocToErrorQueue
SourceLeave	Send document to Error Queue	OriginalDocToErrorQueue
SchedulePersist		
SwitchEnter		
CaseExecute		
SwitchLeave		
ScheduleDone		

Name	Value
EventID	{6C90EEA3-4918-11d3-A242-00C04F60A5F4}
ModuleId	{38D8FB14-2CB4-448E-9873-4E4E0EC18D69}

Figure 10-11. Actions taken for the trade schedule with an aborted transaction

Because you've raised an error in the trading component (for example, with the Err.Raise method in VB), and the trading component is configured to abort the transaction when encountering a bad HRESULT, the first transaction, Trade Proc, is aborted at the ContextAbort event. The process flow then continues with the On Failure of Trade Proc page where an e-mail is sent to the administrator and the document is sent to the message queue.

After the schedule finishes the process on the On Failure page, it enters the Decision shape, where it identifies that an error has occurred in the previous transaction, and it terminates the current schedule.

Long-Lived Transactions

It sounds like short-lived transactions do exactly what you ask them to do, so you may be wondering why you need long-lived transactions. Let's reexamine the previous schedule.

You know that when a client document first arrives at the schedule, you'll update the database to indicate that the transactions in the document are in progress. Then you start processing the document with the trading component. Depending on the size of the transaction, some orders may not be fulfilled for a long time. For example, if a fund company decides to buy a million shares of IBM, you really have no idea when the order can be fulfilled. It may take 3 minutes or 3 hours. If it indeed takes 3 hours to fulfill the order, do you really want to lock up the data you've updated (or read, depending on the isolation level) in the database so that no one else can read or update it? The answer is generally no. But what else can you do? As long as the trading component doesn't commit or abort, the database changes you made can't be committed or aborted either. Having too much data locked up for too long will dramatically degrade the performance of an application that depends on such data.

To solve this problem, BizTalk Orchestration introduced a new type of transaction called a *long-running transaction*. (Timed transactions are almost the same as long-running transactions, as you'll see later.) The biggest difference between long-running transactions and short-lived transactions is that long-running transactions aren't really transactions. Every action inside a long-running transaction will be committed or aborted as soon as it completes. When an action has been committed, the change is final and can't be rolled back regardless of whether other actions in the same long-running transaction succeed or fail. The only difference between processes in long-running transactions and those without any transaction is that you can perform certain actions in the event of transaction failure.

Here's the benefit of long-running transactions over short-lived transactions: Because everything is committed or aborted right away, the length of time locks are held on the data source is significantly reduced. For example, all the data that is updated by the Pre-Trade Database Update will become available for other applications immediately after the action completes. This means, though, that automatic rollback won't be available when the transaction aborts, since committed data can't be rolled back.

To better understand the properties of a long-running transaction, let's modify the schedule you created earlier to include a long-running transaction (see Figure 10-12). Here you have a long-running transaction, and as with a short-lived transaction, you can define an On Failure page for it. Only the actions wrapped in the long-running transaction won't be rolled back automatically—for those actions you'll need to reverse the changes yourself on the On Failure of Long Tx page.

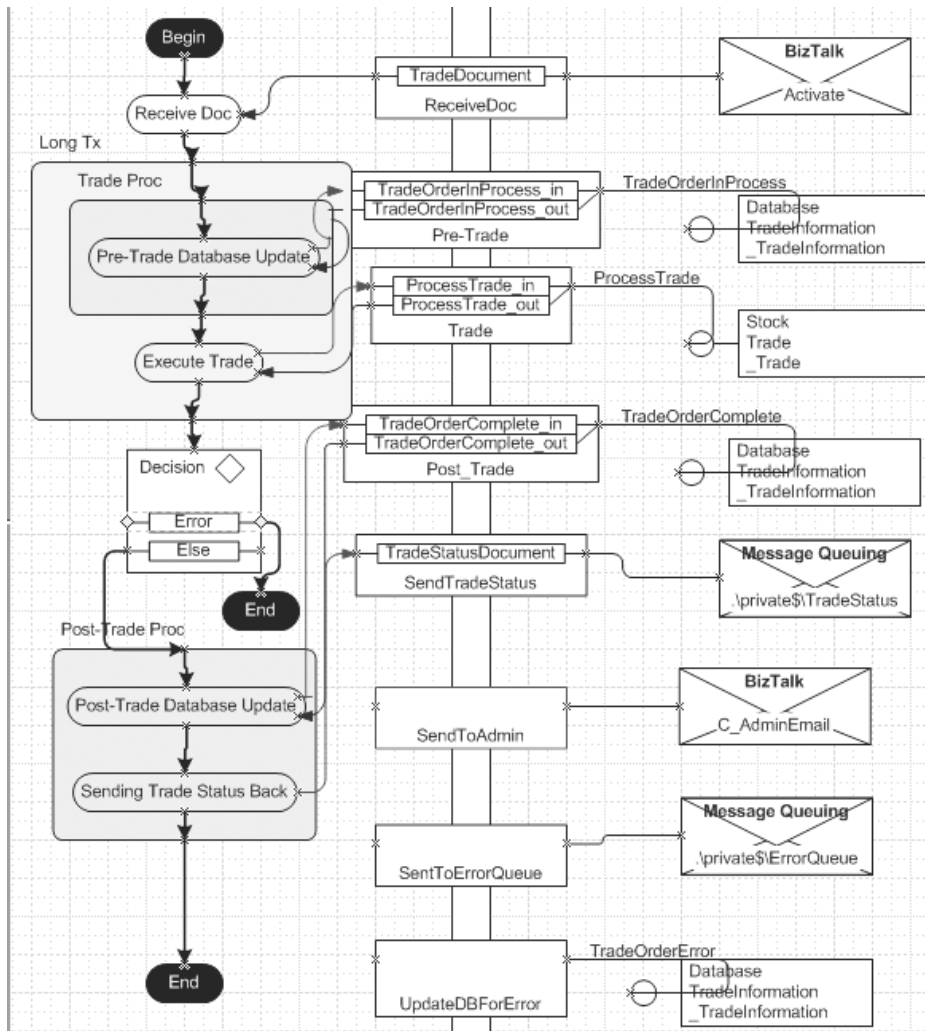


Figure 10-12. The trade schedule with a long-running transaction

Things are getting a little complicated with the short-lived transaction nested inside the long-running transaction in Figure 10-12. With nested transactions, you can delegate tasks that can be completed in relatively short time to short-lived transactions to take advantage of automatic rollback, and leave the tasks that take more time to run in the long-running transaction to take advantage of the faster release of resources.

In a nested transaction, there are two Business Process pages that are used to handle the “on failure” situation. The first one is the regular On Failure page, and

the second one is called the *Compensation page*, and both are specified in the Transaction Properties window (see Figure 10-13).

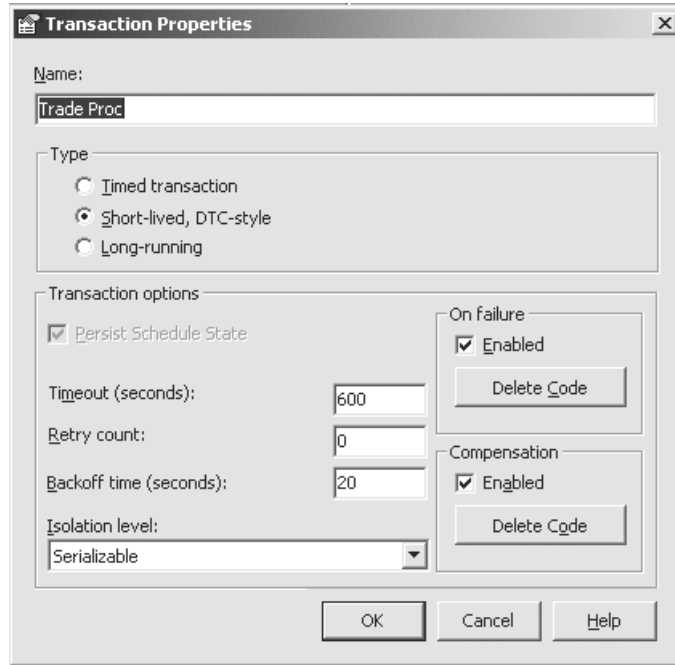


Figure 10-13. Properties page for a short-lived transaction inside a long-running transaction

When a short-lived transaction resides inside a long-running transaction, you can define a Compensation page on top of an On Failure page. In this case, you'll have a total of three "transaction failure" pages for the Long Tx transaction. There will be two On Failure pages for Long Tx and Trade Proc and one Compensation page for Trade Proc.

The Compensation page is only called when the Trade Proc transaction has been committed but the Long Tx transaction is calling for an abort. The processes on the Compensation page are defined in exactly the same way as those of the On Failure page. Keep in mind that the actions in the corresponding transaction have been committed by the time the Compensation page is called, so actions that are taken on this page must reverse the changes that were committed.

To test the new schedule, raise an error in the trading component, and watch it run in the XLANG Event Monitor. You should see the actions from the Compensation of Trade Proc page and from the On Failure of Long Tx page being called.

Timed Transactions

With long-running transactions, your trading component can run for hours without locking other data sources. However, at some point, you want to terminate the schedule. In such cases, you want to use timed transactions. The features and behavior of timed transactions are exactly the same as those of long-running transactions, except that you can define how long a transaction will run before it times out or aborts. With a timeout property, you can put a lid on how long a long-running transaction can run in the system.

A Review of Transactions

As you saw, there are two transaction models for XLANG Schedules: COM+ component-style transactions and XLANG-style transactions. Only with XLANG-style transactions can you define short-lived, long-running, and timed transaction types.

Short-lived transactions are DTC-style transactions, and they give you the benefit of automatic rollback when the transactions abort, but they also have the shortcomings associated with DTC transactions, such as locking up data for extended periods, which could hurt the performance of other applications accessing the same data.

Long-running transactions solve the data-locking problem by committing the changes as soon as each action in the transaction completes, but you have to provide your own rollback logic for when the transaction aborts.

When a short-lived transaction is nested inside a long-running transaction, two pages are called when the process aborts: The On Failure page defines actions that will be called when the short-lived transaction fails, and the Compensation page defines the actions that will be called when the short-lived transaction is committed and the surrounding long-running transaction is aborted.

Timed transactions are simply long-running transactions with a timeout limit.

Using XLANG Schedule Throttling

Schedule throttling is a new feature in BizTalk Server 2002. It makes your BizTalk application scalable and also solves a very real problem. Let's use the trade schedule as an example.

When clients send a document to Bob's BizTalk Server for processing, the XLANG engine will start an instance of the schedule and will start loading the components to process the document. What happens when Bob's client tries to

send several documents at once? For example, the news of a federal interest-rate hike was just released, and every client reacts to the news and starts buying or selling stocks. Bob's ASP will receive over 50 trade orders at once. How will BizTalk Server react to these sudden demands?

If you don't control how many trade schedules can run at once, BizTalk Server will attempt to start 50 trade schedules, and each of the schedules will attempt to instantiate all the components it needs to process the document. Having so many schedules running at once will significantly degrade the performance of every schedule running on the machine. The situation will become worse if the components used in the schedule are also making database reads or modifications. You can run into problems when the maximum number of database connections is reached. The performance of the database will also significantly degrade because the database server is overwhelmed with managing so many data locks at once.

The result is that all the schedules are running on the system, but none of them are getting any work done. What you need is a way to manage how many instances of a particular schedule can run at one time so that BizTalk Server won't be overwhelmed by a huge number of running schedules. The solution to this problem is XLANG Schedule throttling or pooling, which allows you to specify the maximum number of instances of a specific schedule that can run at one time.

To enable and configure this feature, open the schedule in Orchestration Designer, and open the Properties page of the Begin shape (see Figure 10-14).

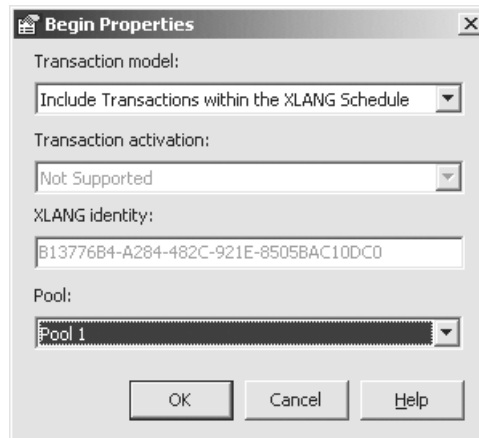


Figure 10-14. Configuring an instance pool for an XLANG Schedule

By default, the value selected in the Pool list box is None, which means schedule pooling isn't enabled for this XLANG Schedule. Change this value to a pool number—for the purposes of this example, select Pool 1. Click OK to save the changes. This is the only thing you need to do in the schedule.

Next, open Component Services. Under COM+ applications, locate the XLANG Schedule Pool application, and then expand its Components folder (see Figure 10-15). Under the XLANG Schedule Pool are 25 components. Each component will control the instance pooling for an XLANG Schedule. The default maximum number of instances for each of these components is 25, and you can modify these through the activation property of the components. Note that there are only 25 components under the XLANG Schedule Pool application. This number can't be changed.

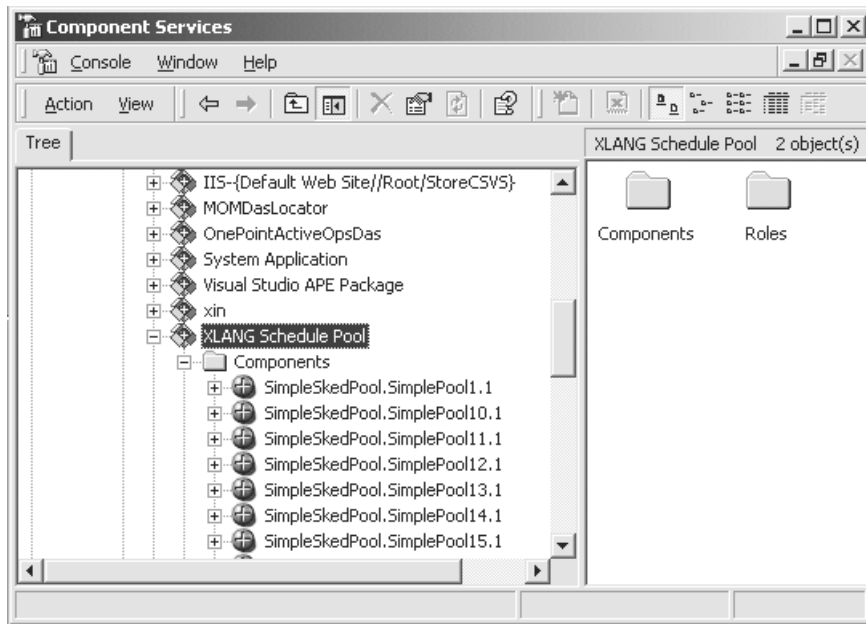


Figure 10-15. XLANG Schedule Pool application

To configure the XLANG Schedule pooling application, open the properties page for `SimpleSkedPool.SimplePool1.1`, which corresponds to the Pool 1 setting you selected on the Begin shape of the schedule (see Figure 10-16).

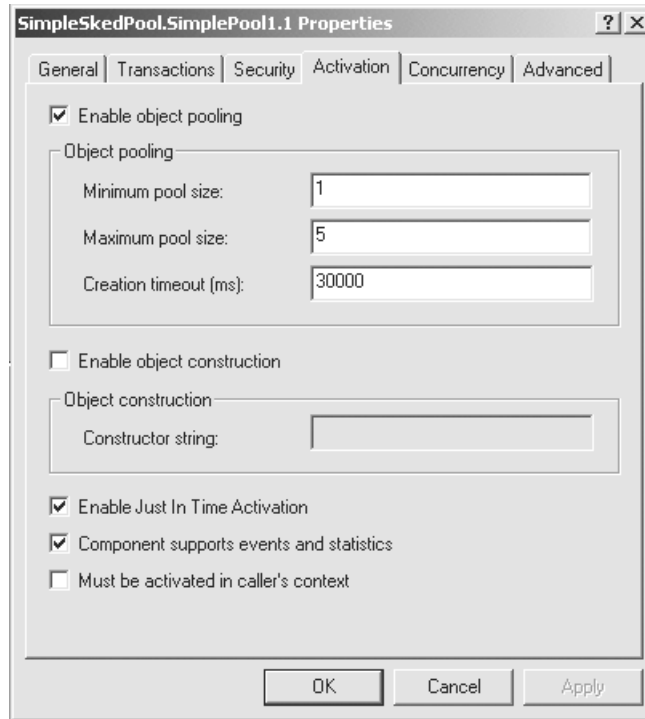


Figure 10-16. Configuring the pooling properties for the XLANG Schedule

On the properties page, select the Activation tab. Check the Enable object pooling checkbox and the Enable Just In Time Activation checkbox. Set the maximum pool size to 5 to specify that a maximum of five running instances of the trade schedule can exist on the system at one time. Click OK to save the changes.

To test the configuration, add some code to the trading component so that it will take longer to complete (for instance, put a msgbox function in the code so that it requires user interaction before proceeding). Then open the XLANG Event Monitor (see Figure 10-17).

Now start ten trade schedules by dropping ten copies of the document in the folder that is watched by the Receive Function that starts the trade schedule. Only five instances will run on the system at once.

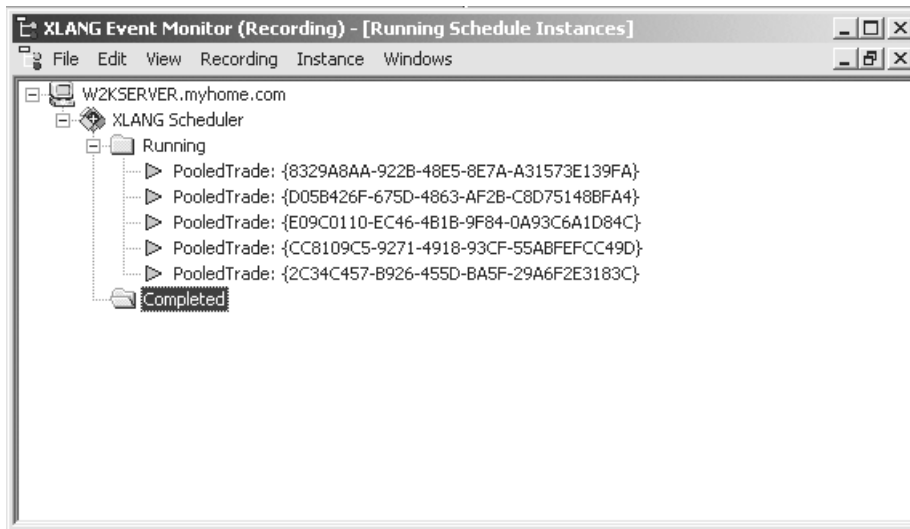


Figure 10-17. With pooled XLANG Schedules, you can limit the number of schedule instances running at one time.

If you look into the Queues folder in the BizTalk Server Administration Console, there will be five trade documents waiting in the Work Queue folder to be picked up as soon as a free spot becomes available in the schedule pool. As soon as a schedule instance is completed, a spot in the pool will become free for the next schedule instance.

In Bob's case, when 100 trade documents arrive, the BizTalk XLANG engine will process the documents 5 at a time, and the system resources won't be overloaded by the excessive outstanding schedule instances.

Dehydrating and Rehydrating XLANG Schedules

So far, you've seen many features of XLANG Schedules. You've looked at the Action shapes and implementation shapes and what they can do for schedules, but there is another less obvious, yet very important feature of XLANG Schedules. In this section, you'll learn about the *dehydration* and *rehydration* of XLANG Schedules.

When an XLANG Schedule is dehydrated, all the schedule's instance-related state information is persisted to the database, and that schedule's instance is erased from memory, freeing system resources. The rehydration of an XLANG Schedule reverses the dehydration process; it restores the dehydrated instance back to memory so that it can continue its processes. Figure 10-18 outlines the dehydration and rehydration processes.

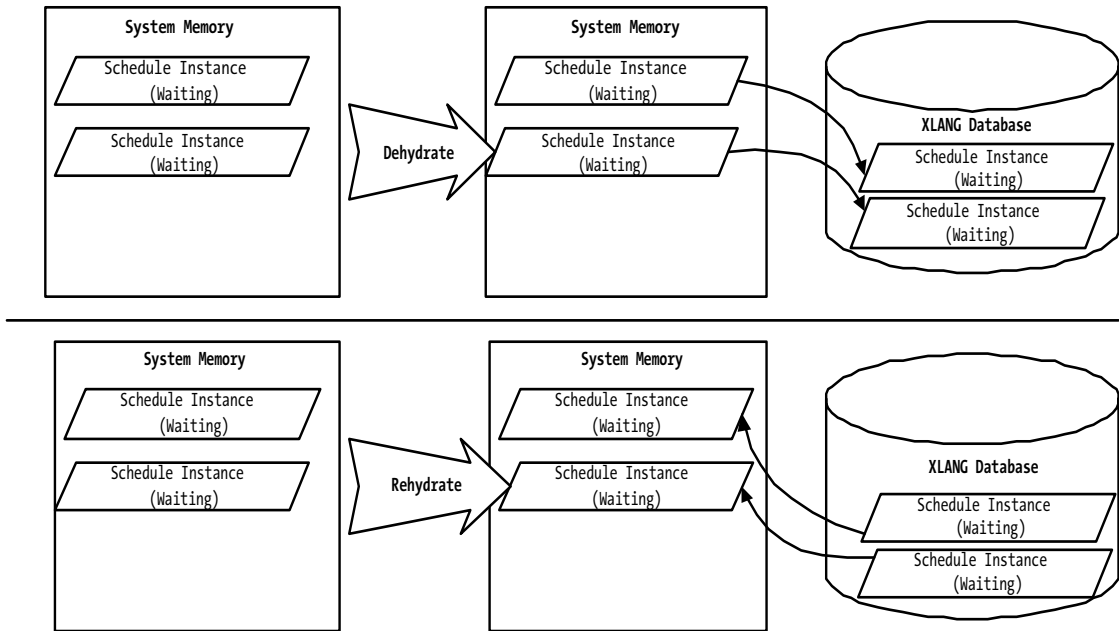


Figure 10-18. The dehydration and rehydration processes

In many B2B scenarios, a process in a workflow can be blocked for long periods of time before the workflow can continue. If a worker sent an order to the manager for approval and didn't expect to have the order approved for a couple of days, it would certainly be unacceptable for the worker not to do any other work while waiting for the approval. The same issue exists in XLANG Schedules—when a schedule instance is stopped and is waiting for a message to arrive before it can continue its process, BizTalk Server will consider dehydrating this particular instance so that the freed system resources can be used by other applications.

When a schedule is waiting on a port, for instance, one of the actions in the schedule will receive the document from a Message Queuing shape. If there is no message in the queue, the schedule will be blocked at that port until a message arrives at the message queue. In addition to the blockage, the following conditions must also be met before BizTalk Server will actually dehydrate this schedule instance to the database:

- The schedule instance must be in a quiescent state. All the actions in the schedule must remain inactive when a port is being blocked. If some components connected to the ports haven't yet completed, BizTalk Server won't dehydrate that schedule instance.

- The schedule instance must not contain any running DTC-style transactions. The XLANG engine won't dehydrate a schedule instance if it's running a short-lived transaction that hasn't yet aborted or committed.
- The implementation for the port must support persistence if it holds the state. For example, if the COM component connecting to the port holds the state but doesn't support the IPersist interface, then the XLANG engine can't dehydrate the schedule instance, because it will be unable to persist the state held by the component to the XLANG database.
- The latency setting for the port must be greater than 180. The latency setting assigned in the XML Communication Wizard should be greater than 180 seconds (see Figure 10-19). BizTalk Server will never dehydrate a schedule instance if the latency value for the blocked port is less than or equal to 180 seconds.



Figure 10-19. Configuring the latency value for the port in the XML Communication Wizard

The rehydration process is much simpler. When a message arrives at the port that has been blocked, the XLANG engine will instantiate the schedule and return the state information and persisted components of that dehydrated

schedule instance back to system memory, where the schedule can continue from where it left off.

Using Orchestration Correlation

In Chapter 5, you saw that BizTalk Orchestration can send data to BizTalk Messaging and to message queues by using BizTalk Messaging implementation shapes and setting the data flow on the port to Send. Similarly, BizTalk Orchestration can receive data from BizTalk Messaging and message queues by setting the data flow on the port to Receive. However, there is a problem that I haven't directed your attention to.

Consider the scenario in Figure 10-20. The schedule is simple. It receives a document from BizTalk Messaging and sends the document to a NewOrder message queue, where the document is picked up by another application for processing. The schedule then waits for a confirmation message. When the order is fulfilled, a confirmation message is sent to the Confirmation message queue, and the schedule will retrieve it from the queue and process the rest of the actions.

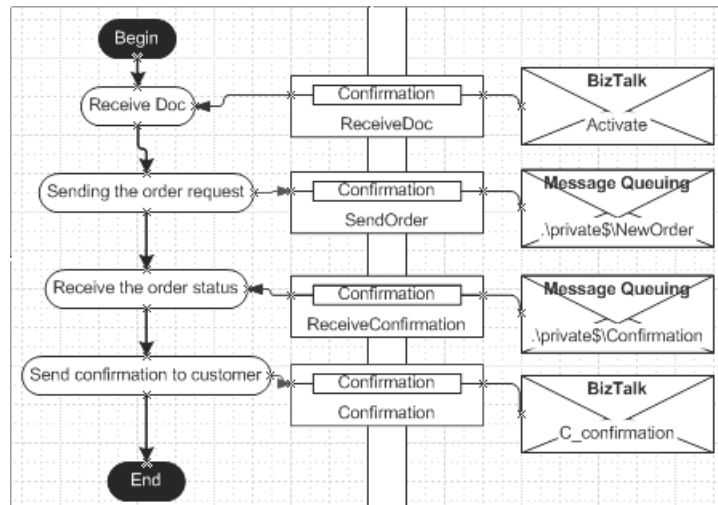


Figure 10-20. A schedule that sends and receives documents using message queues

If you have one schedule instance running on the system, you won't have any problem retrieving the document from the Confirmation queue, because the confirmation document in the queue must have originated from the order document the schedule was processing. However, when multiple instances of this schedule are running on the system, and each instance is processing an order

document from a different client, you'll have a problem. How do you determine which schedule instance will pick up the document that just arrived at the Confirmation queue? How do you know which schedule instance was processing the order document that this confirmation matches?

To this point, you've been thinking about XLANG Schedules in a single-instance scenario, but in the real B2B world, it's common to have multiple instances of the same schedule running on a system at once. You must find a way to ensure that the messages a schedule instance receives are indeed intended for that particular instance. This means that whenever a schedule instance receives a document, either from BizTalk Messaging or a message queue, it won't simply assume that the arriving document is associated with the document it has been processing. Instead, BizTalk Server has a technique called *orchestration correlation* to handle this problem.

Orchestration Correlation Through Message Queues

Mike has discovered some performance problems in the trade schedule. The problem lies with the trading component. Currently, the trade schedule makes a synchronous call to the trading component, which sometimes can take a long time to process the order, and many smaller orders have to wait on the queue even though they could be processed very quickly. Mike wants to change the current schedule to remove the direct synchronous calls to the trading component.

Mike decides to make the schedule send the trade order to a remote message queue instead of calling the trading component. The trading components will be moved to the server that hosts the remote message queue and will process the documents as soon as they arrive from the trade schedule. Multiple instances of the trading component will be activated to process the trade orders as quickly as possible. After an order is executed, the response document (which contains the execution status information) is sent to another message queue that is watched by the running trade schedule instances. As soon as a response document arrives, the correct schedule instance will pick up the response document and will process the Post-Trade Database Update transaction and forward the response documents back to the original clients.

To ensure that the running schedules will pick up the correct response documents from the message queue, the orchestration correlation technique will be used in this new trade schedule. The new trade schedule replaces the Execute Trade action with two new Action shapes that are connecting to message queuing shapes (see Figure 10-21).

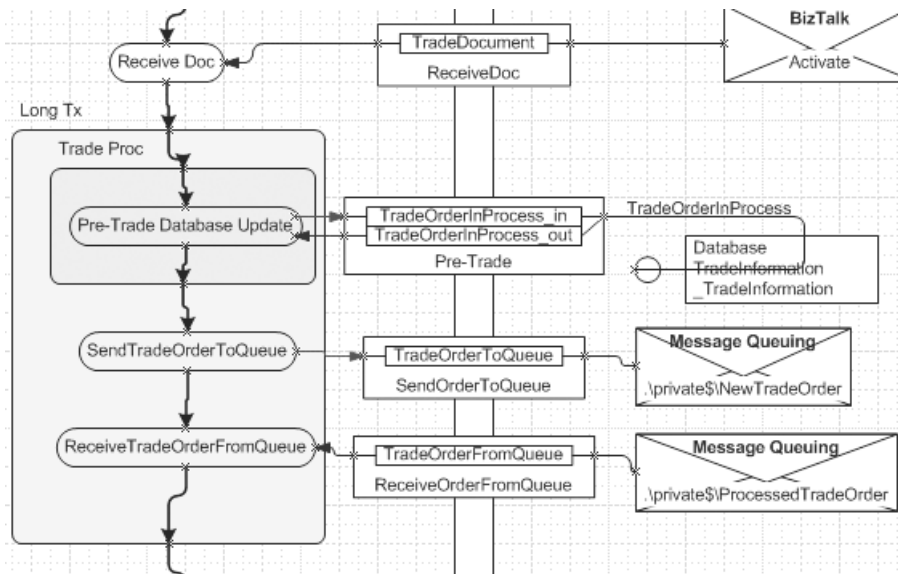


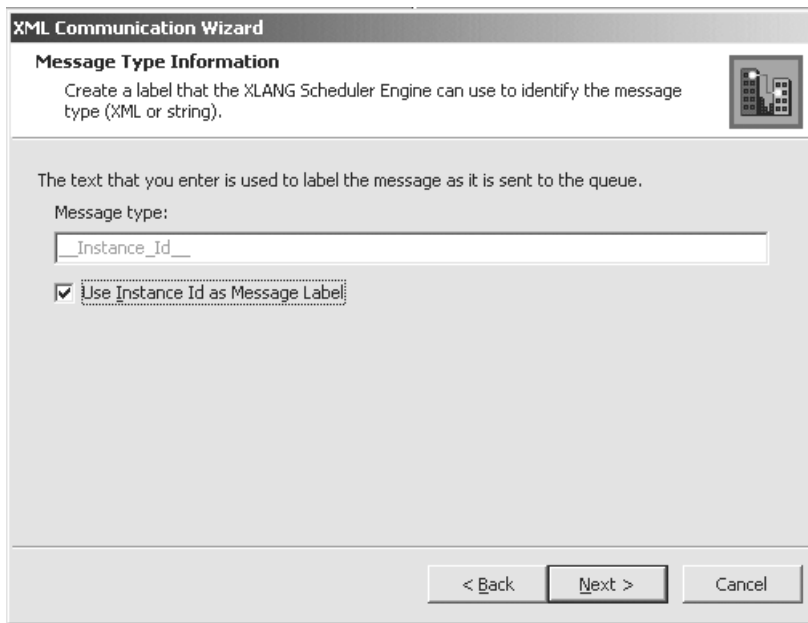
Figure 10-21. Revised trade schedule

The `SendTradeOrderToQueue` action will send the trade order to the message queue called `NewTradeOrder`, where it will be picked up to be processed by the trading component. After the trading component completes the trade order, it will generate a response document that contains the transaction status information, and will send the response document to a separate message queue called `ProcessedTradeOrder`. The `ReceiveTradeOrderFromQueue` action will then receive the correct response document from the queue and go on to the subsequent actions.

Now, let's add two Message Queuing implementation shapes to support orchestration correlation. Drag and drop the first Message Queuing shape onto the Business Process page. In the Message Queuing Binding Wizard, specify the port name as `SendOrderToQueue` and the static message queue path as `.\private$\NewTradeOrder`.

Next, connect the `SendTradeOrderToQueue` action with the appropriate port, and an XML Communication Wizard will open. You need to create a new message to represent the document that will be sent to the queue.

The next window will ask for the message label information (see Figure 10-22). Check the "Use Instance Id as Message Label" checkbox. The Message type field will then contain an uneditable value, `__Instance_Id__`. The instance ID is a GUID that uniquely identifies a schedule instance. This ID is critical in identifying which document is associated with a specific schedule instance. Click Next until you finish the wizard.



XML Communication Wizard

Message Type Information

Create a label that the XLANG Scheduler Engine can use to identify the message type (XML or string).

The text that you enter is used to label the message as it is sent to the queue.

Message type:

☒ Use Instance Id as Message Label

< Back Next > Cancel

Figure 10-22. Using the instance ID as the message label

Now you can drag and drop the second Message Queuing shape onto the Business Process page, and specify `ReceiveOrderFromQueue` as the port name and `.\private$\ProcessedTradeOrder` as the message queue path. Connect the `ReceiveTradeOrderFromQueue` Action shape with this Message Queuing shape.

In the XML Communication Wizard that opens, specify that this port is to receive messages. If you expect that the trade order will take a long time to process, you may want to set the wait time to a number greater than 180 seconds, which will allow the XLANG engine to dehydrate the schedule instance while it waits for messages to arrive. For this example, set it to 200 seconds.

In the next window, create a new message called `TradeOrderFromQueue` to represent the response document that arrives at the message queue.

Click Next to proceed to the window shown in Figure 10-23. Check the `Use Instance ID as Message Label` checkbox. This setting is important because a schedule instance will only accept a message from a queue whose message label is equal to its instance ID. You also must provide a Message type name, but this value is irrelevant if you are using the instance ID as the message label, so just type in anything and click Next until you finish the wizard.

XML Communication Wizard

Message Type Information

Create a label that the XLANG Scheduler Engine can use to identify the message type (XML or string).

The text that you enter is used to identify the correct messages to receive from the message queue.

First, the XLANG Scheduler Engine tries to match the message type information with the message label in the queue.

Second, if the message is in an XML format on the queue, the XLANG Scheduler engine tries to match the message type information with the XML root element of the message on the queue.

Message type:

test

☒ Use Instance Id as Message Label

< Back Next > Cancel

Figure 10-23. Specifying the message type information for the port

The central idea of using the instance ID as the message label for orchestration correlation is that all the running schedules will check the message label of the messages as they arrive at the queue. If the message label of a message in the queue has the same value as the instance ID of one of the running schedules, that particular running schedule will pick up the message and process it. Other running schedules won't pick up that document since none of them has the matching instance ID.

To make the whole correlation process work, the trade component must make sure the message label (which contains the instance ID) of the message it picked up from the queue has the same message label when the message is sent out after processing. Throughout the whole process, the message label must remain the same, or the correlation won't work. BizTalk Server wouldn't be able to match the messages with the correct running schedules otherwise.



NOTE You can read and assign message labels for messages in the queue programmatically though the use of the MSMQ object model. In this example, the trade component will need to call these MSMQ methods to read and assign the message labels to ensure they stay the same. For more information about the MSMQ object model, refer to Appendix A.

You also need to modify the data page to make sure all the messages are linked correctly.

After you compile the schedule, you are ready to test it. However, you must set up some Receive Functions, channels and ports for this schedule before you can test it. Figure 10-24 shows the CorrelationTrade schedule in the XLANG Event Monitor. Notice the snowflake icon beside CorrelationTrade—it indicates that the schedule instance has been dehydrated. Open the detailed events for the schedule instance (shown in the Events for CorrelationTrade window in Figure 10-24) to reveal that schedule is indeed dehydrated and is waiting on the ReceiveTradeOrderFromQueue action.

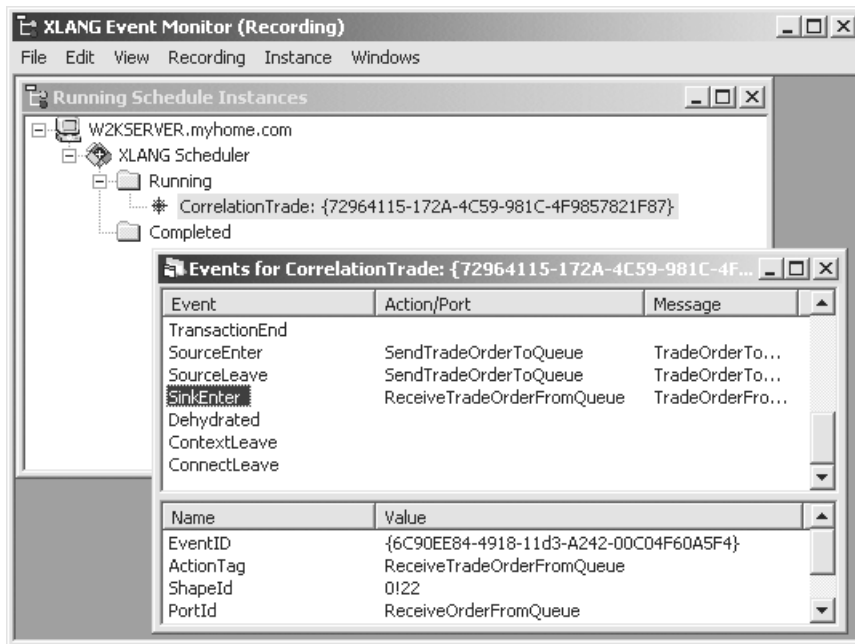


Figure 10-24. Dehydrated schedule instance is waiting for a message to arrive.

The next step is to send a message to the `.\private$\ProcessedTradeOrder` queue so that the schedule instance can continue. To test it, handcraft multiple response documents and give one of them the same label name as the GUID shown in the XLANG Event Monitor, in this case `{72964115-172A-4C59-981C-4F9857821F87}`. Shortly after you send these messages to the `ProcessedTradeOrder` queue, that dehydrated `CorrelationTrade` instance will be rehydrated and will start processing the message. When it's completed, you should find that there are still some messages in the queue, and only the matching message is removed.

With orchestration correlation, you can convert the many synchronous tasks into asynchronous ones by implementing this “send-receive” pair in the schedule. This asynchronous approach removes the binding between the schedule and the business components, and when combined with the XLANG Schedule’s ability to be dehydrated, will greatly improve the resource management and scalability of the XLANG Schedules under a heavy load.

Orchestration Correlation Through BizTalk Messaging

In the example you just saw, the message label is a critical piece of information that helps BizTalk Orchestration correlate the messages and the correct running schedule instances. However, sometimes you’ll want to send the document to a BizTalk Messaging shape instead of to a Message Queuing shape. In such cases, using message labels to keep track of which schedule instance the document belongs to won’t be feasible, since there is no place to store this message label information. You must come up with a different solution to correlate the messages and the schedule instances.

Bob decides not to use his internal trading component for some reason. He has found a company on the Internet that can offer the trade execution service to his clients instead. All clients need to do is send the trade order document to the external provider over HTTPS. As soon as the trades have been executed, a response document containing the execution status information is sent back to the clients, who can then update their database or perform any further post-trade operations.

Bob has decided to use this company’s service and asks Mike to replace his trading component with the external trading system, and integrate this trading service into the trade schedule. Figure 10-25 shows the general plan.

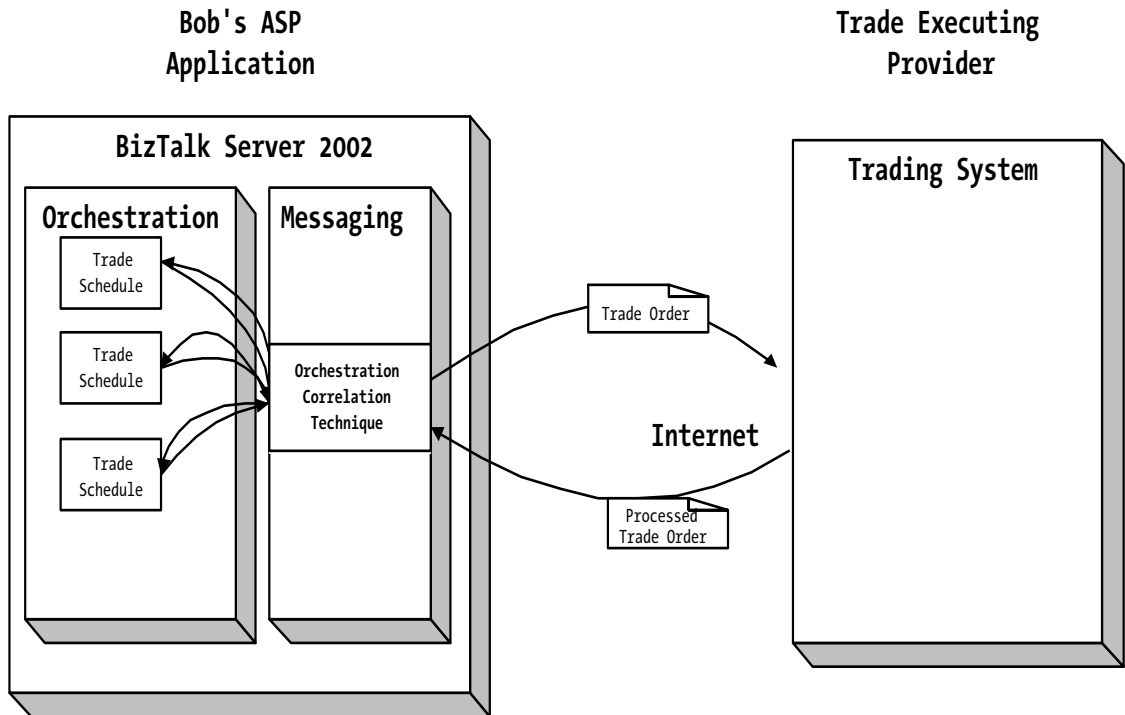


Figure 10-25. A revised system to process trades for clients

To help Mike make this change, you need to revise your trade schedule to replace your trading component and integrate with an external provider. You'll replace the two Message Queuing shapes with two BizTalk Messaging shapes, which are responsible for sending the trade order to the external provider through the Internet, and receiving the response document and correlating it with the correct trade schedule instance. Because you have to remove the message queuing part of the picture, and your external provider is outside the organization, you need to provide two pieces of information to the external provider. You need to tell them where you expect to receive the response document and what the instance ID should be for this response document.

One way to provide this information is to embed it inside the document when it's sent out. Let's modify the trade document specification to add a field called `ReplyTo` right below the `FundCompany` field. Figure 10-26 shows the trade schedule revised for this new change.

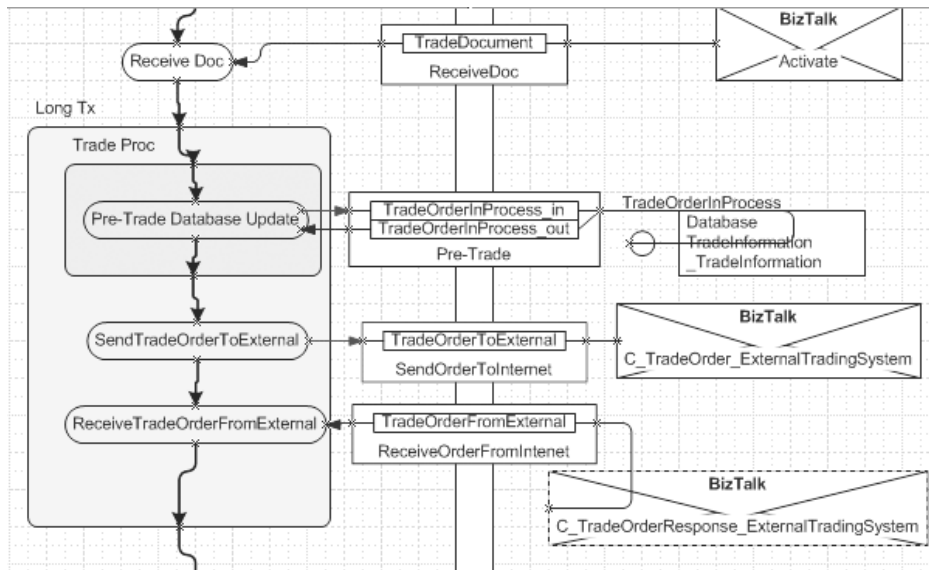


Figure 10-26. BizTalk Messaging shapes are used instead of Message Queuing shapes for sending and receiving documents.

As you can see in Figure 10-26, the two message queuing shapes are replaced by the two BizTalk Messaging shapes. The `SendTradeOrderToExternal` action will send the mutual fund company's trade order document to a `C_TradeOrder_ExternalTradingSystem` channel, which is connected to a port with the URL of the external trade execution provider as the destination.

When configuring this shape with the XML Communication Wizard, you need to pull out the `ReplyTo` message field to make it available for access on the data page, which you'll see shortly. Give any name for the message label for the XML message—the name doesn't matter here since you aren't writing it to the message queue.

Next, you need to add the BizTalk Messaging shape used for receiving the response document. Drag and drop the BizTalk Messaging shape onto the page. The BizTalk Messaging Binding Wizard will open, as shown in Figure 10-27.

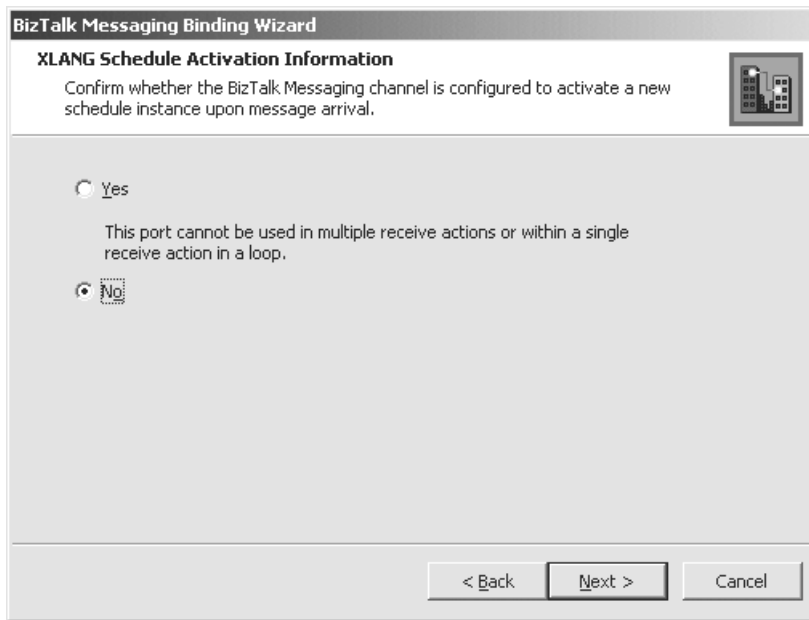


Figure 10-27. XLANG Schedule activation information

You've seen this window many times before in earlier chapters, and in it you've usually selected Yes. This time, though, you need to select No. When Yes is selected, BizTalk attempts to create a brand new schedule instance to process the document. When No is selected, BizTalk Server won't attempt to create a new instance of the schedule. In this case, you already have a running schedule instance—all you want to ask BizTalk Server to do is feed the data into this port.

Click Next to proceed to the window shown in Figure 10-28, which you haven't seen before.

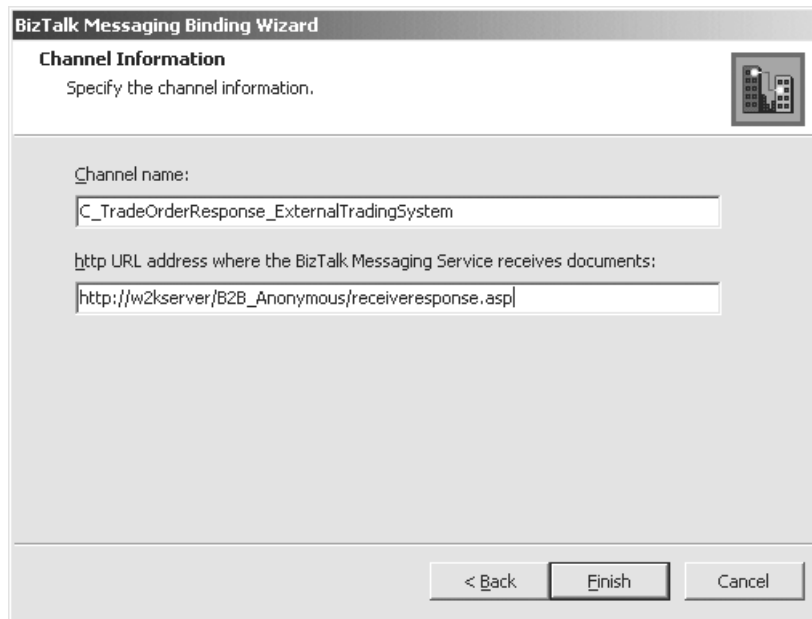


Figure 10-28. Specifying how BizTalk Server receives the response document

In this Channel Information page, you need to provide two important pieces of information to configure how BizTalk Messaging will receive the document before it can pass the document to the running schedule instance. In Figure 10-28, you are instructing the external trade execution provider to send the response document to `http://w2kserver/B2B_Anonymous/receiverresponse.asp`, and you can add the channel name `C_TradeOrderResponse_ExternalTradingSystem` as one of its query strings. In other words, you are specifying that your URL for accepting the response document is `http://w2kserver/B2B_Anonymous/receiverresponse.asp?channel=C_TradeOrderResponse_ExternalTradingSystem`.

You might ask why you specify this information in BizTalk Server. Shouldn't you tell your provider this information instead? Bear with me for a second, I'm getting to that. Click Finish to save the change.

Next, you must connect the `ReceiveTradeOrderFromExternal` action with the port. During the XML Configuration Wizard, specify the Message type as `Trade`, which is the root name for the response document. This value is important, because when documents arrive at BizTalk Server, this Action shape will only retrieve the documents with `Trade` as their root name. Leave the instance ID alone, since you aren't dealing with the message queue now.

After you finish the configuration of the BizTalk Messaging shapes on the business page, there is still something you must do to get the whole thing to work. You've provided the channel and URL that are used to receive the response

document on the port, as is shown in Figure 10-28. Next you must send them along with the document to the external system provider. To do so, you need to link the port reference and the ReplyTo field in the document, as shown in Figure 10-29. The value of the ReplyTo field is dynamically created when an instance of this schedule is started. You'll see what value it takes later.

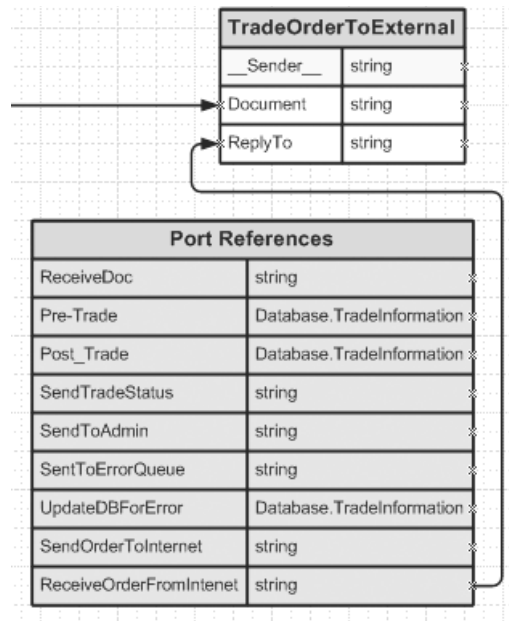


Figure 10-29. Connecting the port reference to the ReplyTo field in the document

If you caught the document posted to the external service provider, you would see that the ReplyTo field contains the following value:

```
<ReplyTo>http://w2kserver/B2B_Anonymous/receiverresponse.asp?Channel=
C_TradeOrderResponse_ExternalTradingSystem&
QPath=W2KSERVER.myhome.com\private$c_tradeorderresponse_externaltradingsystem
{8c2786fd-7e6f-4d07-a96b-92852f4151ec}</ReplyTo>
```

This ReplyTo field contains a URL to an ASP page with two query string items: Channel (Channel=C_TradeOrderResponse_ExternalTradingSystem) and Qpath (QPath=W2KSERVER.myhome.com\private\$c_tradeorderresponse_externaltradingsystem{8c2786fd-7e6f-4d07-a96b-92852f4151ec}). In order for the schedule to correlate the response document with the right running schedule instance, the sender (external system provider) must post the response document to the URL that is embedded in the document.

If you have a question about where the QPath value in the query string is coming from, just check the local message queue. You'll find that the QPath is actually the path to a local queue that is created by the schedule at runtime (see Figure 10-30). The name of this queue is formed by combining the channel name and the instance ID of the schedule. This name is unique, since the instance ID is always unique. With orchestration correlation through BizTalk Messaging, BizTalk Server actually creates one message queue for each running instance of the schedule for correlation purposes.

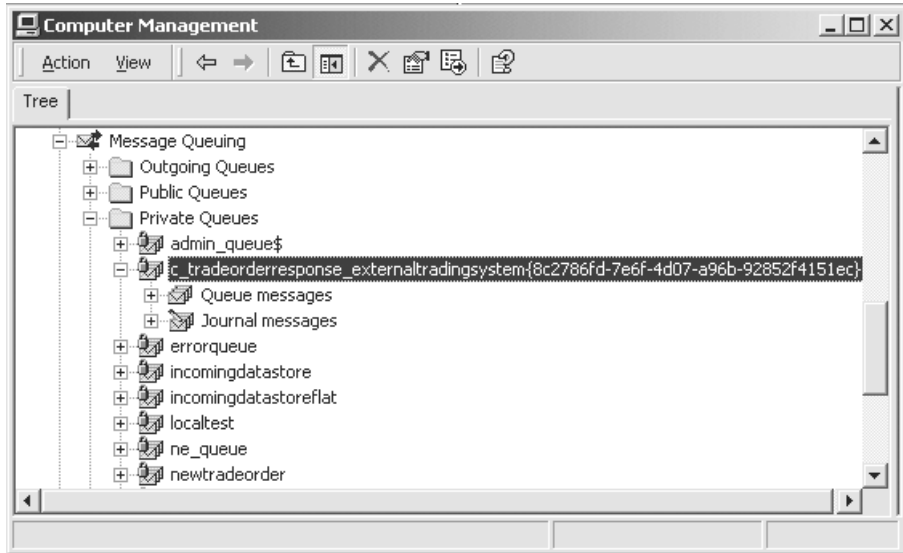


Figure 10-30. A local queue is created by BizTalk Server for storing the incoming message.

After the trade schedule sends out the trade order, the schedule will be dehydrated while it waits for the response document to arrive. In order for the dehydrated schedule instance to continue, the response document must arrive at the BizTalk Server. To be exact, a response document must arrive at this message queue, which is dynamically created to handle the document returned to this port for this particular schedule instance. As soon as a message arrives at this `c_tradeorderresponse_externaltradingssystem{8c2786fd-7e6f-4d07-a96b-92852f4151ec}` queue, the schedule will be rehydrated and will pull the message out of this queue, bring it into the rehydrated schedule, and continue with the rest of the orchestration. After the schedule is complete, this local queue will be automatically removed.

The external trading service provider doesn't have to know how to get the message to the dehydrated schedule instance. All they need to do is post

the response document using the URL stored in the ReplyTo field. Your job is to create an ASP page that will transfer the posted documents to those dynamically created local queues.

BizTalk Server provides an ASP page for this task. The file is located under [BizTalk Server installation directory]\SDK\ReceiveScripts\ReceiveResponse.asp. In this ASP page, there are a number of key lines of code that will do the trick:

```
'retrieve the parameter in the query string
queuepath = Request.QueryString("qpath")
queuepath = "queue://Direct=OS:" & queuepath
channelname = Request.QueryString("channel")
'add you code to extract the posted document
'submit to the BizTalk Server
call interchange.submit (4,PostedDocument,,,,queuepath,channelname)
```

The submit call uses 4 as its first parameter to specify that the document is sent to an open destination. The destination value for this document is queuepath, which will be used as the destination target for the open port. The channelname is the channel that connects to the open destination port.



NOTE *The channel information is optional, but it's helpful to provide it in the submit call if it's available. In order for BizTalk Server to find a channel on its own, you have to make sure the source and destination information provided is sufficient for BizTalk Server to find a distinct channel. In many cases, it's easier to provide a channel name in the submit call than to decide whether less information is sufficient. If you do provide a channel name, BizTalk Server will recognize it and will bypass the search for the channel and use the one you provide. This also applies to the Receive Function. When channel information is provided on the advanced properties page of the Receive Function, BizTalk Server will use it to pass the document to the port.*

To complete the process, you also must create a channel and an open destination port to be used for correlating the posted document back to the schedule-instance message queue, where the schedule instance will pick up the document and continue on its processes.

Using Dynamic Port Binding

I discussed open destination ports in BizTalk Messaging in Chapter 10. This is where the destination of the port is unknown until it's provided by a data field in the document or in submit call parameters. This concept decouples the destination from the port, and it's very useful when you don't know the destination beforehand. In BizTalk Orchestration, you need to provide either a channel name and message queue path during the configuration of the implementation shapes in order to send a document to BizTalk Messaging or to a message queue. You may be wondering whether there is a way to decouple the destination address from the implementation shape so that your schedule can have the flexibility of an open destination port.

The answer is yes. Dynamic port binding allows you to provide the address information to the implementation shapes at runtime. First, let's take a look at a dynamic port binding connected to a Message Queuing shape (see Figure 10-31).

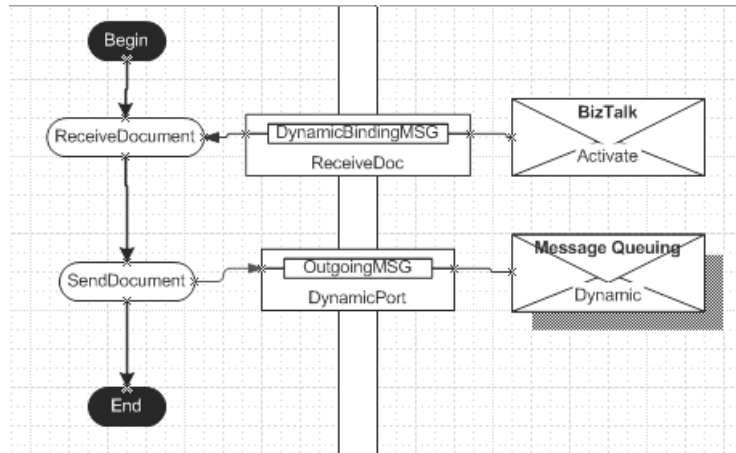


Figure 10-31. A schedule with dynamic port binding to a message queue

When you are using the Message Queuing Binding Wizard, specify a dynamic queue instead of a static queue. A Message Queuing shape with a shadow will appear on the page. To provide the address, you need to link a data field containing the address information to the port the dynamic messaging queue connects to on the data page (see Figure 10-32).

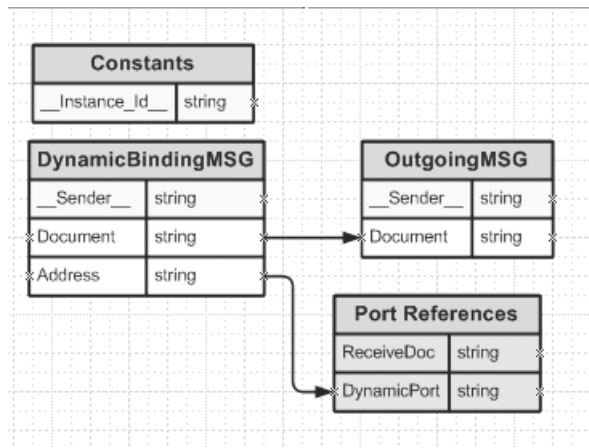


Figure 10-32. Providing the address information by linking the address data field to the dynamic port reference

Inside the document, you must have a data field that contains the address information for the message queue or this dynamic port binding won't work. Here is a sample document that can be used in this schedule:

```
<message>
  <Address>.\private$\DynamicBindingQueue</Address>
  <Data>this is a test message</Data>
</message>
```

When you run this schedule, the message is written to the message queue specified in the message queue information in the document.

You can create dynamic binding ports for BizTalk Messaging shapes the same way you did for the Message Queuing shape. When the BizTalk Messaging Binding Wizard asks for the channel information, specify a dynamic channel. You'll also need to link a data field that contains the name of the messaging channel to the port reference on the data page, and the address field in the document must be protocol specific. In other words, if you use dynamic binding on a message queue, the address field must be a queue path. If you use dynamic binding on BizTalk Messaging, then the address field must be a channel name. Here's an example, with `C_Message_ExternalClient` being the channel name:

```
<message>
  <Address>C_Message_ExternalClient</Address>
  <Data>this is a test message</Data>
</message>
```

Using Web Services in BizTalk Orchestration

A *Web service* is an application or program that can be accessed through the Internet using the HTTP(S) protocol. The intriguing aspect of Web services is their ability to integrate applications running on different platforms in different locations. This benefit has been sought by many technologies in the past, such as DCOM, but they all faced a common challenge—each of them used a proprietary protocol to communicate with clients and servers.

Simple Object Access Protocol (SOAP) provides a standard way to access applications on a remote system. SOAP is a network protocol, and it has no programming model. It's merely a specification for how to invoke and receive the services of a remote system. With SOAP as the communication protocol between client and server, Web services have become easier than ever to use and implement.

Web services can also be integrated with BizTalk Orchestration. There are two ways of integrating the Web services and BizTalk Orchestration:

- Accessing Web services within an XLANG Schedule
- Exposing XLANG Schedules as Web services

I'll discuss each in turn.

Accessing Web Services Within XLANG Schedules

BizTalk Orchestration doesn't provide a way to invoke Web services directly. However, because it can call COM components, you can make BizTalk Orchestration access the Web service through a COM implementation shape (see Figure 10-33).

Access Web Service within XLANG Schedule.

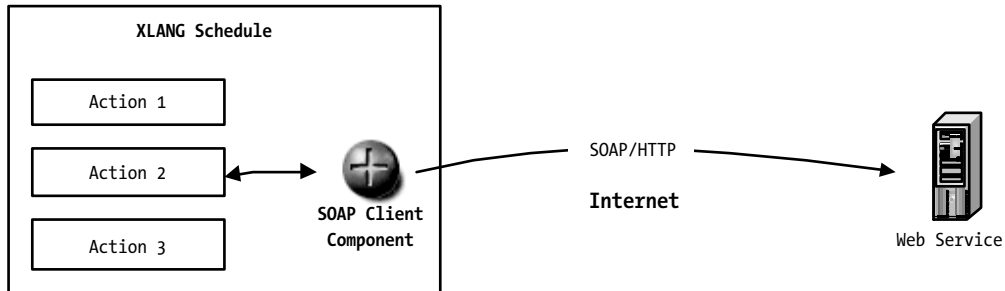


Figure 10-33. Accessing a Web service from within an XLANG Schedule through a COM component binding

Bob wants to retrieve a list of quotes from the stock exchange so that he can use it to calculate the current market values for the portfolios of his mutual fund clients. He heard that the major stock exchanges have implemented something called a Web service, which allows clients to request quotes through the Internet. He wants Mike to make their schedule capable of accessing this quote information through the Internet.

To enable the schedule to access the Web service, you need to develop a COM component that can access the Web service. The following code will create a SOAP client in Visual Basic:

```

Function QuoteServiceClient(doc As String) As String
Dim SoapClient As New MSSOAPLib.SoapClient
SoapClient.mssoapinit "http://w2kserver/QuoteService/QuoteService.wsdl",
"QuoteService", "QuoteServiceSoapPort"
QuoteServiceClient = SoapClient.ReceiveQuotes(doc)
End Function
  
```

You need to add “Microsoft SOAP Type library” to the VB project reference in order to access its functionality. First, you must instantiate a SoapClient object, since you are requesting the Web service as a client. Next, you need to initialize the SoapClient object with the Web Services Description Language (WSDL) file for the quote service. The WSDL file tells clients what methods and properties are available for service, and it also contains the data type for the parameter, and the result for the method calls. In other words, it’s effectively a type library for the Web service.

To make a request for the Web service, you simply make a method call on the SoapClient object. All the work of generating a SOAP message and sending the message over to the other side are handled by the SoapClient object behind the scenes. With all the complex tasks being handled for you, you can simply call a method of the Web service as you do with a regular COM component.

Exposing XLANG Schedules As Web Services

XLANG Schedules can also be exposed as Web services to the outside, with help from a COM component. An XLANG Schedule is BizTalk Server technology, and it can't be directly exposed as a Web service as a COM component can be. However, there is a BizTalk API that allows a COM component to start an XLANG Schedule. You can use such a COM component as a wrapper for the XLANG Schedule, which will provide the real service.

Suppose the Stock Exchange is using BizTalk Orchestration to provide the quote service, and it has decided to expose this quote service as a Web service so that clients can request the stock quotes through the Internet (see Figure 10-34).

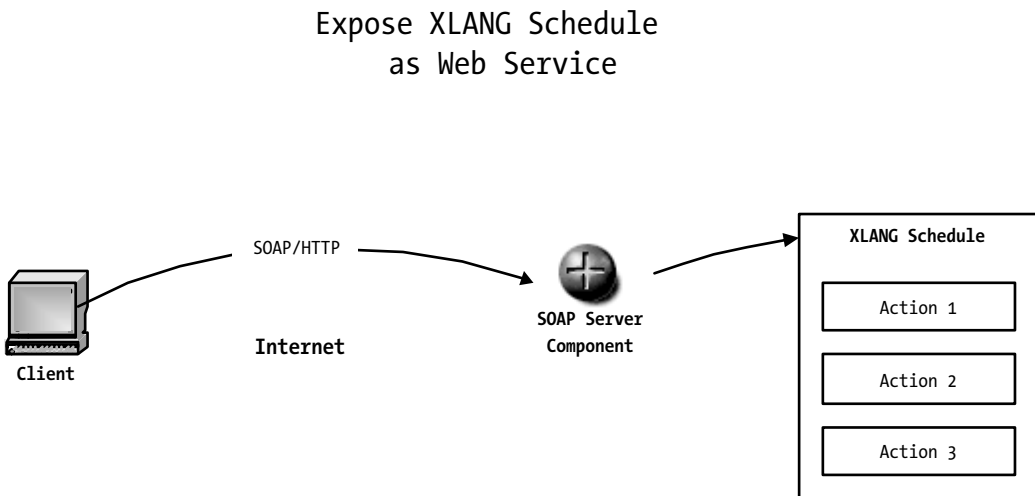


Figure 10-34. Exposing an XLANG Schedule as a Web service through a COM component

Three steps are involved in exposing an XLANG Schedule as a service:

1. Create an orchestration that has entry points for programmatic access from external applications.
2. Create a COM component that can interface with the XLANG Schedule.
3. Create the WSDL and WSML files for this COM component, and expose these files to the Internet for clients to access.

Step 1: Create an Orchestration with Entry Points

Figure 10-35 shows the Quote Service schedule running on a stock exchange's BizTalk Server to provide the quote service.

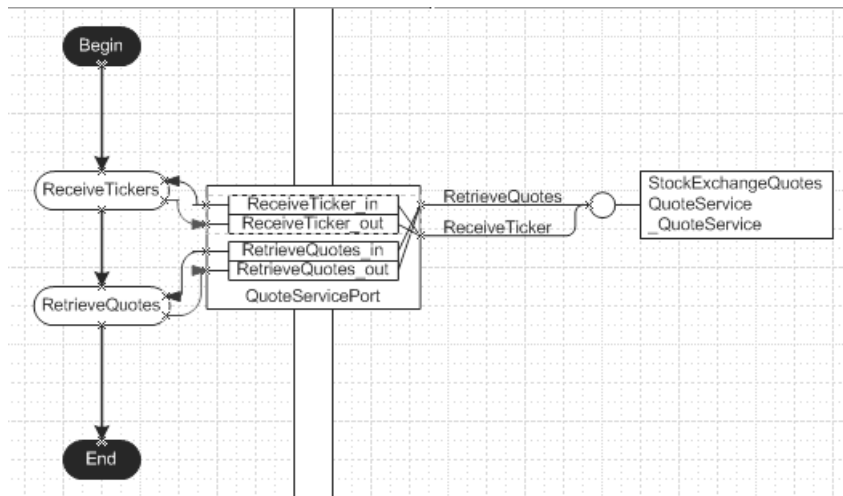


Figure 10-35. An orchestration that provides a quote service for clients

An important aspect of this schedule is that it contains entry points for external applications to start and control its flow. There are two actions on the schedule: *ReceiveTickers*, which receives documents that contain stock tickers from clients, and *RetrieveQuotes*, which returns the quotes for each ticker in the document.

When you configure the Method Communication Wizard for both actions, you'll see the window in Figure 10-36.

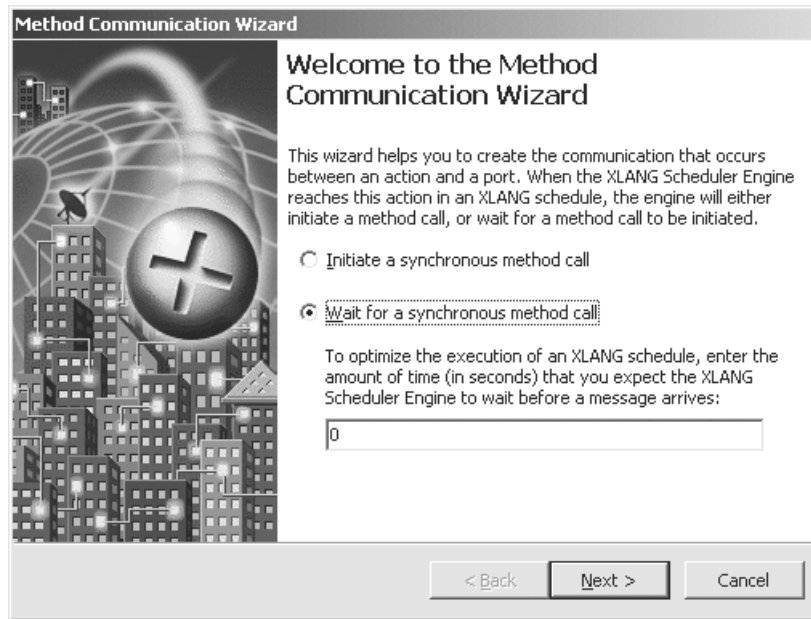


Figure 10-36. Method Communication Wizard

In the earlier situations, you’ve selected the “Initiate a synchronous method call” option, because the schedule is really in control of the process flow in earlier examples. This isn’t the case in the quote service, where the external application (the COM wrapper in this case) is in control of the process flow of the schedule. When you select “Wait for a synchronous method call” on the port, the schedule process flow will stop when it reaches this port, and it won’t continue until an external program makes a call to the method that the port is connecting to.

If you still feel unclear about this option, don’t worry. Things will be clearer when you take a look at the COM wrapper for this schedule. That’s where you’ll see how you can control the flow of this schedule programmatically.

Step 2: Create a COM Component to Interface with the Schedule

The following code creates the COM wrapper that you’ll expose as a Web service shortly:

```
Public Function QuoteServiceController(Tickers As String) As String
    Dim ssked_URL As String
    Dim sked As Object
```

```

'set the URL for quote service schedule.
ssked_URL = "sked:///C:\Program Files\Microsoft BizTalk Server\
XLANG Scheduler\quoteservice.sx/QuoteServicePort"
'Load and start the schedule
Set sked = GetObject(ssked_URL)

'Provide Ticker document by calling the method on the port
sked.ReceiveTicker (Tickers)

'Retrieve the quote result by calling the method on the port
QuoteServiceController = sked.RetrieveQuotes()
End Function

```

There are four key points you should know about the preceding code:

- *ssked_URL*: This is the path for the schedule. Port reference (QuoteServicePort) is part of the path.
- *set sked = GetObject(ssked_URL)*: The `GetObject` method takes one parameter—the path of the skx file for the schedule you want to load. After you call this method, you have a running schedule that can be referenced by the object `sked`.
- *sked.ReceiveTicker (Tickers)*: Before this method is called, the schedule you've just started is waiting on the QuoteServicePort port. This is because you've configured the port to wait until someone else initiates the `ReceiveTicker` method. By calling the `ReceiveTicker` method and passing in the ticker document, the process flow of the schedule can now continue to the next action: `RetrieveQuotes`.
- *sked.RetrieveQuotes*: Before this method is called, the schedule waits at the QuoteServicePort port. The schedule can't continue until something calls the `RetrieveQuotes` method. The `RetrieveQuotes` method returns the quotes for the tickers, and the return value is then assigned to the return value of the `QuoteServiceController` function. The schedule is then finished after the method call is completed.

As you can see, if you didn't set the schedule to wait for the initiation of the method, the schedule wouldn't wait for the external application to provide input data, and it wouldn't extract the output data at the right moment.

Step 3: Create the WSDL and WSML Files and Expose Them to the Internet

Once you've compiled this VB code into a DLL, you have a COM component that contains a method called `QuoteServiceController`. This method takes a ticker document as its input parameter and returns the quotes document. The next step is to enable the Web service on this COM component, and expose the `QuoteServiceController` method.

To enable the Web service on the COM component, you must create a WSDL file to describe the service provided in this component, and a WSML file to map the service in WSDL to the component. Creating these two documents can be an extremely tedious task. However, it only takes few clicks to create them if you use the WSDL generator that comes with SOAP Toolkit 2.0. SOAP Toolkit is a separate product, and you can download it free from the Microsoft Web site.

After you've installed SOAP Toolkit 2.0, open the WSDL generator by selecting **Start > Programs > Microsoft SoapToolkit** (see Figure 10-37). In the first SOAP Toolkit window, specify the name of the Web service and .dll file for which you'll be generating the WSDL file, and click **Next** to proceed.

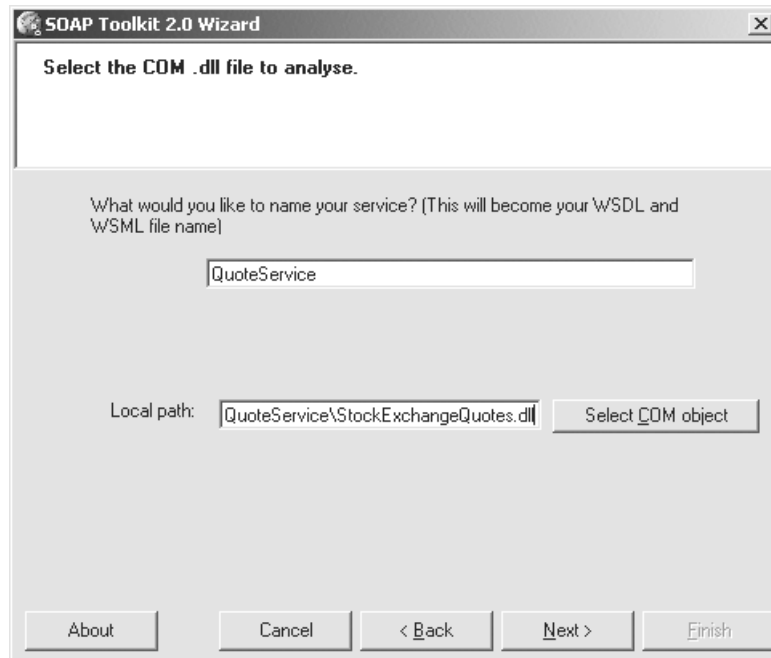


Figure 10-37. Defining the name and COM component for the Web service

When you click the Next button, the wizard will examine and gather information about the interface and methods of the COM component. In the window shown in Figure 10-38, the SOAP Toolkit displays the possible interfaces and methods you can expose for clients to access. Click the QuoteServiceController method, and click Next to proceed.

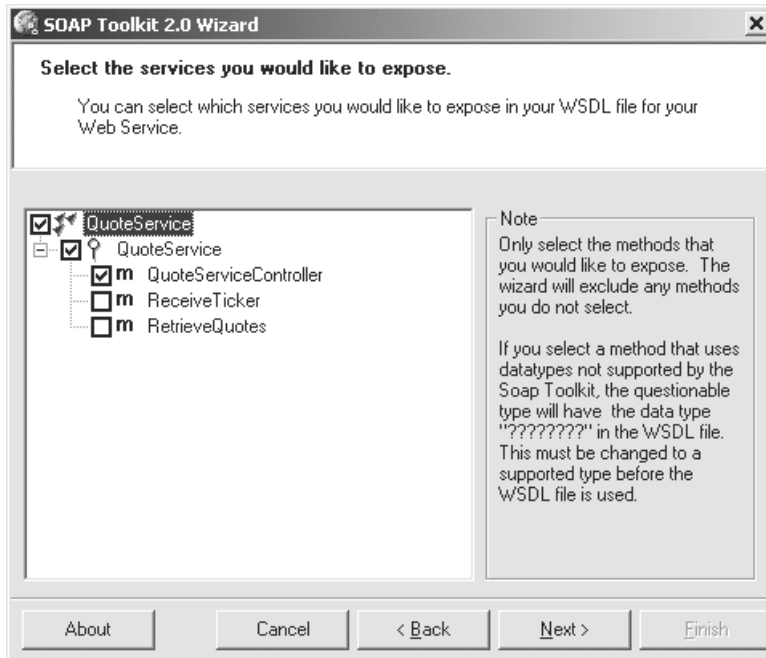


Figure 10-38. Selecting the method to expose through SOAP Toolkit

In the next window, shown in Figure 10-39, you need to provide the Uniform Resource Identifier (URI) of the listener file and the type of the listener. The listener can be either an ASP file or ISAPI .dll, as these are the files responsible for accepting the HTTP requests from clients and returning the results of their requests back to clients.

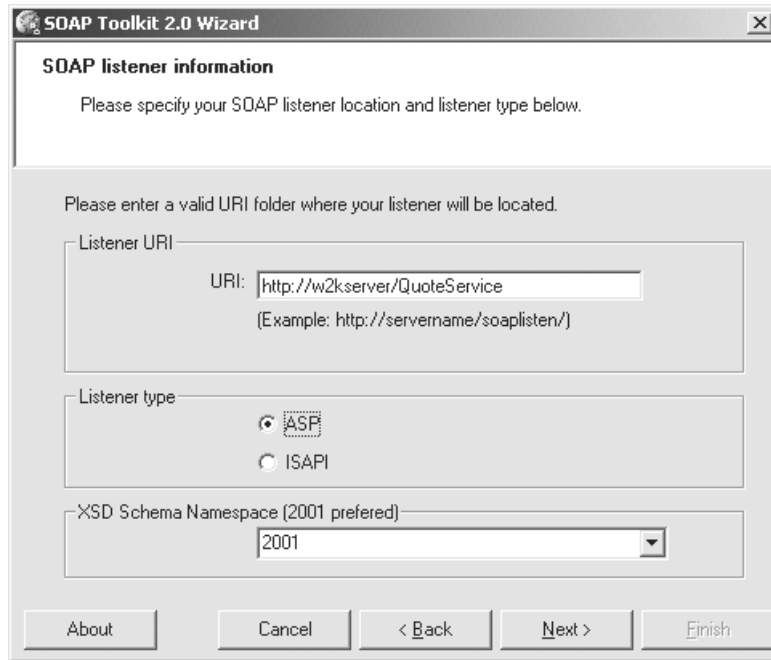


Figure 10-39. Configuring the SOAP listener for the service

The URI information will be saved to the WSDL file so that clients know where the entry point for the service is. The SOAP Toolkit Wizard will automatically generate the ASP file or ISAPI .dll at the end. You must then copy the files to the location of the URI you specify in this window.

Click Next to proceed. In the next window, specify the location where you want the SOAP Toolkit Wizard to save the generated file. Click Next until you finish.

The SOAP Toolkit Wizard has just created three files: QuoteService.wsdl, QuoteService.wsml, and QuoteService.asp. The last thing you need to do before you can start testing is to copy these three files to the file directory that `http://w2kserver/QuoteService` references.

Test the Web Service

To test the integration between the Web service and BizTalk Orchestration, you can start the QuoteServiceClient.skx file (included as part of the source code, available on the Downloads section of the Apress Web site, at <http://www.apress.com>). The QuoteServiceClient schedule will call the QuoteServiceClient component, which

will make a SOAP call to the QuoteServiceController method hosted as a Web service under the `http://w2kserver/QuoteService` directory. QuoteServiceController will then start the QuoteService.skx file (included in the source code) to process the document and return the quotes back to the QuoteServiceController method, which in turn will pass the quotes back to the QuoteServiceClient component hosted inside the QuoteServiceClient XLANG Schedule.

In my case, both the QuoteServiceClient schedule and the QuoteService schedule are on the same machine. However, the communication between them is through a Web service. When I start the QuoteServiceClient schedule, my XLANG Event Monitor shows what you can see in Figure 10-40.

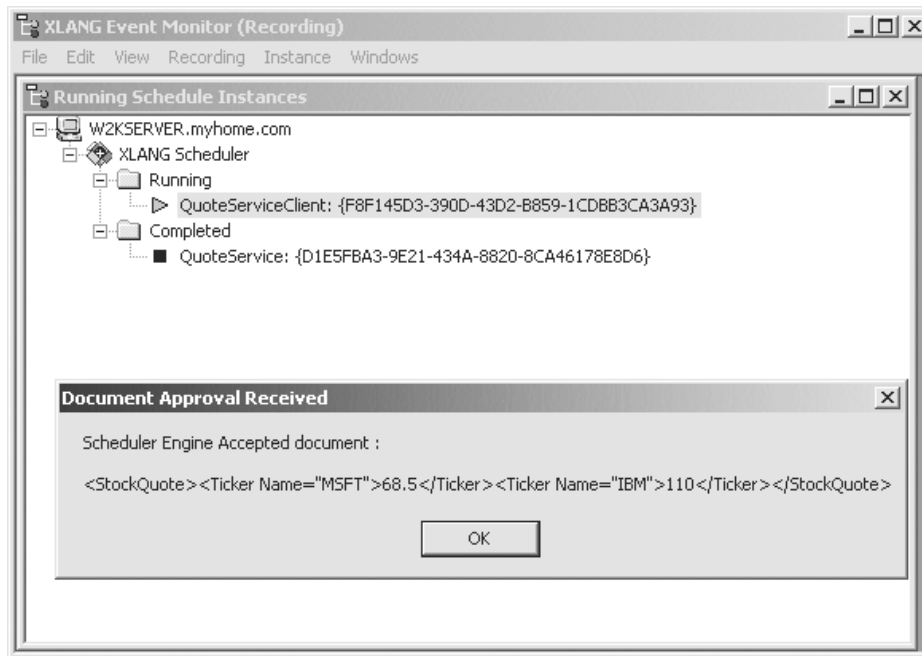


Figure 10-40. QuoteServiceClient schedule is calling the QuoteService schedule through a Web service.

In this test, you learned that you can make an XLANG Schedule access a Web service through a SOAP client component, and you can also turn an XLANG Schedule into a Web service by exposing it through a COM component that controls it.

Summary

You've learned several important features about BizTalk Orchestration in this chapter. You started with an exploration of orchestration transactions, and the three transaction styles—short-lived, long-running, and timed—and you saw their application in different business scenarios.

You also learned about XLANG Schedule throttling, a useful feature that allows you to control how many active XLANG Schedule instances are running at a time, hence controlling how much resources can be used for each type of orchestration on the system. This feature significantly boosts the scalability of XLANG Schedules on BizTalk Server.

You learned about orchestration correlation features, which allow you to correlate documents to their corresponding XLANG Schedule instances, and you also learned about dehydration and rehydration of XLANG Schedules, which are BizTalk Server's way of persisting running schedules to disk to free up system memory used by the schedules, boosting the system resources on that server.

Combining the dehydration/rehydration feature, orchestration correlation, and long-running transaction support in BizTalk Orchestration, you have an excellent solution for handling large amounts of workflow that can take hours or even days to complete, while still using minimum amounts of system resources.

Dynamic ports is another feature covered in this chapter that adds value to BizTalk Orchestration. With dynamic ports, your orchestrations now have the ability to decide where to send the document at runtime, as opposed to having to specify the destination at design time.

Last, but not least, you looked at how to leverage Web services with BizTalk Orchestration. You looked at a sample stock quote service that exposes the services of an orchestration as a Web service, hence making the services of BizTalk Orchestration available to multiple clients across the Internet at the same time.

In the next chapter, I'll show you what BizTalk Server offers from a programming perspective, and what you can achieve with BizTalk Server APIs.



<http://www.springer.com/978-1-59059-034-8>

BizTalk Server 2002 Design and Implementation

Chen, X.

2003, XVII, 704 p. 471 illus., Softcover

ISBN: 978-1-59059-034-8

A product of Apress