Reliability

Estimated time for completion: 45 Minutes

Goals:

- Decoupling service and client with queues

Message Queues

Message queues can provide further decoupling between Service and Client. A client can send messages even when the target service is not running, e. g. due to maintenance, failure, or a planned downtime. Message queues can also relieve a service during periods with peak request counts.

1. Open the before Queues.sln solution file in this directory.
2. Open the file Contract.cs from the service project and notice that it contains the methods `StartPlaceOrder`, `AddOrderItem`, and `FinishedPlaceOrder`. To see how these methods can be used, open Program.cs from the client project and inspect `Main`. In OrderService.cs the service contract is implemented in the class OrderService. In Host.cs, this class is hosted as a service. Open App.config in the service project to find out how the service is hosted.
3. Notice that in App.config, NetMsmqBinding is used to host the service. Also notice that the relative URL for the queue is `"private/Orders"`. To make your application work, you have to create a private queue called `Orders`. You can do this either within Visual Studio or via the Computer Management MMC snap-in (Control panel / Administrative Tools / Computer Management.) Make sure that the queue is transactional.
4. Start your client and inspect the 4 messages created in the orders queue. Start the service. You should see that the messages in the queue are handled.
5. Stop the service. Without a running service, start the client a couple of times. Start the service again. You should see that the order is not predictable. Furthermore, it is currently not possible to determine which `AddOrderItem` call relates to what `StartPlaceOrder` call.
6. To solve this, modify your service contract so that it requires sessions. To be explicit, modify the operation contracts so that a call to `StartPlaceOrder` is required to initialize a session. (You can achieve this by setting the `IsInitiating` property of the

other operation's contracts to false.) Furthermore specify that `FinishedPlaceOrder` terminates a session.

7. Rebuild and start the service and update the service reference in the client project. To reflect the changes of the service contract in the client project, update the service reference. Start the client and notice the exception that is thrown now. Fix the problem by placing the calls to the service in a transaction scope.

```
using (TransactionScope ts = new TransactionScope())
{
    OrderService.IOrderService proxy =
            new OrderService.OrderServiceClient();
    proxy.StartPlaceOrder("Order1");
    proxy.AddOrderItem("Parrot", 2);
    proxy.AddOrderItem("Dog", 3);
    proxy.FinishedPlaceOrder();
    Cleanup(proxy);
}
```

8. Run the client a couple of times. You should not see any new messages in the message queue. Why?
9. Within the using block, after calling `Cleanup(proxy)`, call `Complete` on the transaction scope. Run the client again. This time, you should see a new message. Inspect the body of this message and notice that it contains information all the four operations.
10. To handle all calls within a single transaction, add an `[OperationBehavior]` attribute to the implementations of `StartPlaceOrder`, `AddOrderItem`, and `FinishedPlaceOrder`. For all these methods, set `TransactionScopeRequired` property of the attribute to true. Since the calls to `StartPlaceOrder` and to `AddOrderItem` should not complete the transaction, set the `OperationBehavior`'s `TransactionAutoComplete` property of these two methods to false.
11. In all operations, call `DumpTransactionInfo` after calling `DumpSessionInfo`. Take a quick look at the implementation of `DumpTransactionInfo` to see what it does.
12. Run the client to add a new message to the queue. Run the service to consume the message. Inspect the services output. You should see that all messages are handled in the same transaction. It should also be obvious a distributed transaction is used here, even though only a single resource manager (the Order queue) is used. This is the case, because MSMQ does not support promoting local transactions to distributed transactions. Verify this distributed transactions are used by inspecting the statistics of the MS-DTC. To see these statistics run DCOMCnfg.exe and select the following path in the tree view: Component Services -> Computers -> My Computer -> Distributed Transaction Coordinator -> Local DTC -> Transaction Statistics. The usage of distributed transactions is an overhead that must be considered in performance critical applications.

Error! Reference source not found.