Orchestrations II

Processing techniques for the next step

Agenda

- Managing orchestration processes
- Port binding options
- Using pipelines in orchestration
- Convoy messaging patterns

Exception Handling

- **Exception handling uses the scope shape to define boundaries**
  - Just like .NET, exceptions are filtered by type
  - Most specific type matched first
  - Unhandled exceptions suspend the orchestration
- **You add an exception handling block to an existing scope**
  - You can use any activities to handle the exception
  - The throw shape can be used to throw a particular exception

Transactions

- **Atomic Transactions**
  - Provide ACID transactions for orchestration state
- **Long Running Transactions**
  - Provide modeling and mechanics for business interactions
  - Used when duration or trust prevent atomic transactions
  - Allow for compensation of committed work

Atomic Transactions

- **Provide *consistency of state* in the orchestration**
  - Includes variables, messages, and process
  - Must use serviced components to include external resources
- **Transactions impact orchestration persistence**
  - No persistence during transaction
  - Guaranteed persistence at the end of the transaction
  - Use when a non-serializable object must be used in orchestration
  - Declare the variable at the scope level

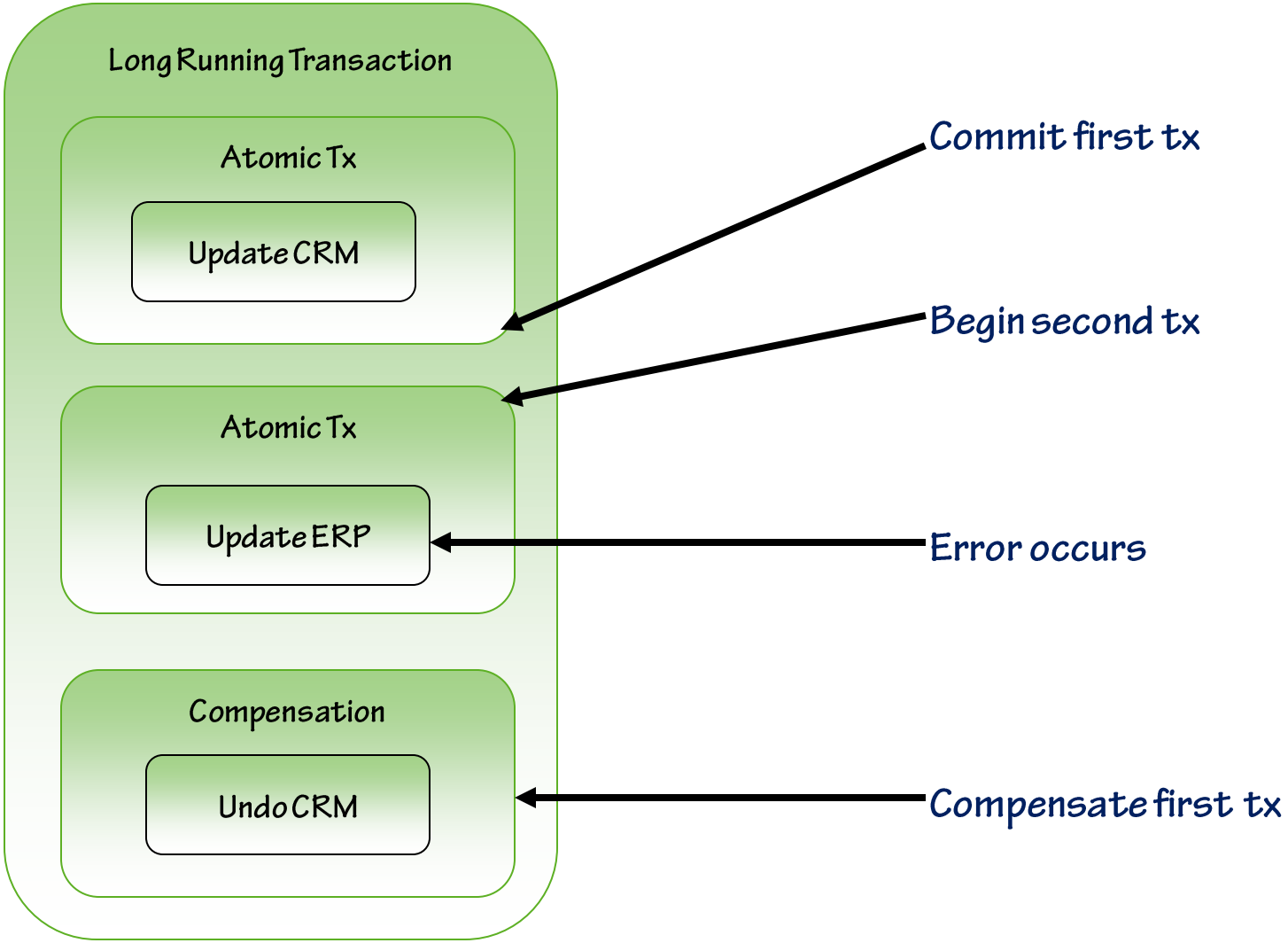
Long Running Transactions

- **System interactions may be *long running* (days, months)**
  - Updates to different systems may still require consistency
  - Long running transactions provide the mechanism
  - Scope the activities participating in the transaction
- **Handle errors and *compensate* where appropriate**

Transaction Compensation

- ***Compensation scopes* define the compensation steps**
  - Added to long running transaction scope
- ***Compensation shape* is used to invoke specific compensation**
  - Directly manage the compensations called
  - Determine which compensation blocks to execute
  - Alter the execution sequence of compensation blocks
  - Use the succeeded operator to determine transaction outcomes

Compensation



Port Binding Options

- **There are several different ways to bind logical orchestration ports to physical ports**
  - Bind at design time – "specify now"
  - Bind explicitly at deployment time – "specify later"
  - Dynamic binding – "dynamic"
  - Direct binding – "direct"
  - Role Links
- **Each binding option comes with its own set of benefits**
- **Most people use only "specify now" and "specify later"**

Dynamic Binding

- **At runtime, the orchestration code sets the address for the port**
  - Address includes moniker which indicates the adapter and address (e.g. FTP://somehost:21/drop)
- **Other information can be dynamically set in the context**
  - User name and password
  - Pipeline configuration data for pipeline components
  - All properties are dependent on the adapter being used

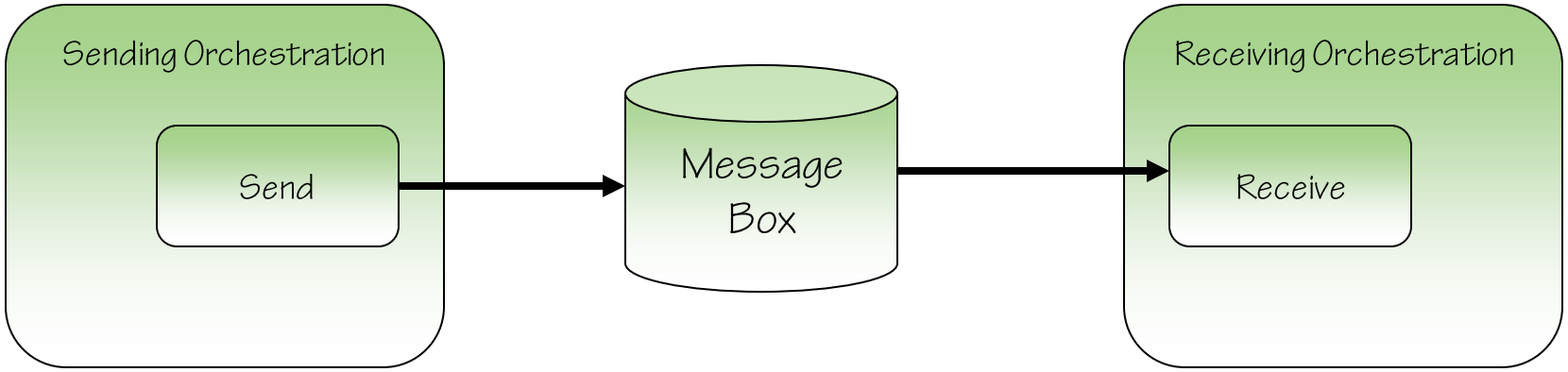
```
ShipperPort(Microsoft.XLANGs.BaseTypes.Address) =
        mailto:someone@microsoft.com;
```

Direct binding

- **Direct binding provides three options**
  - MessageBox
  - Self Correlating
  - Shared Port

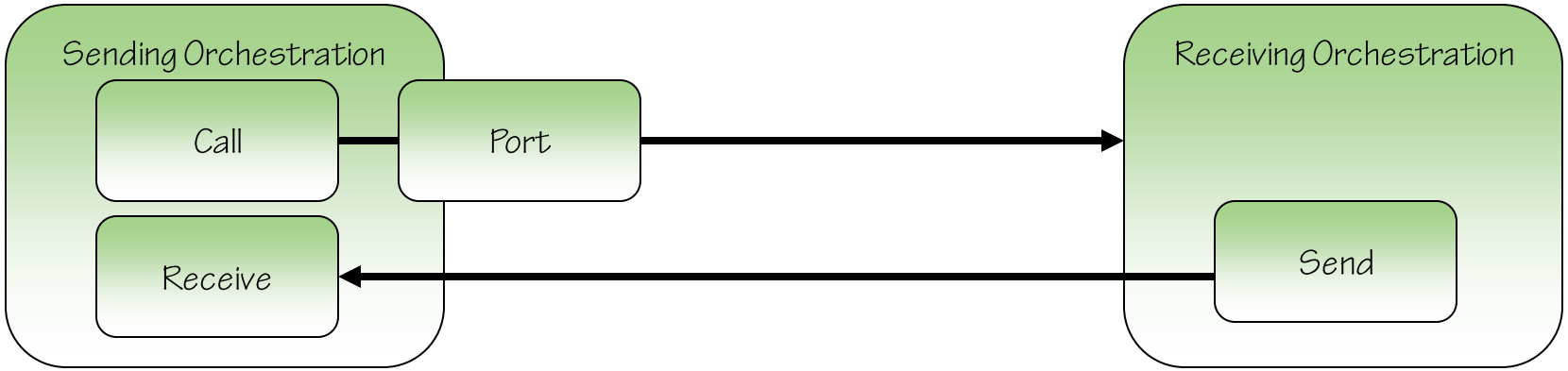
Direct binding with the message box

- **Bound to the message box rather than ports**
  - Use correlation and message context properties
  - Direct bound receive creates subscriptions based on filters
  - Direct bound send is published in the message box and routed
  - No subscribers = exception in orchestration
- **Provides for loosely coupling orchestrations and systems**
  - Uses the publish and subscribe architecture of BizTalk



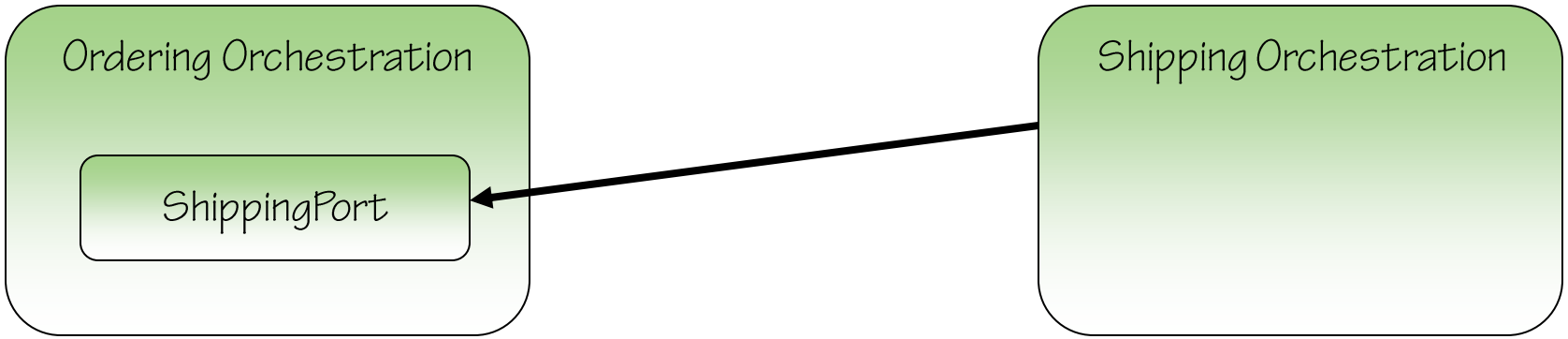
Direct binding with self correlation

- **Self correlating ports rely on the instance being shared**
  - Pass an instance of a port into another orchestration
  - Useful for receiving messages back from a related process
  - Can be used to send more data/messages to running process
  - BizTalk infrastructure handles the correlation with token



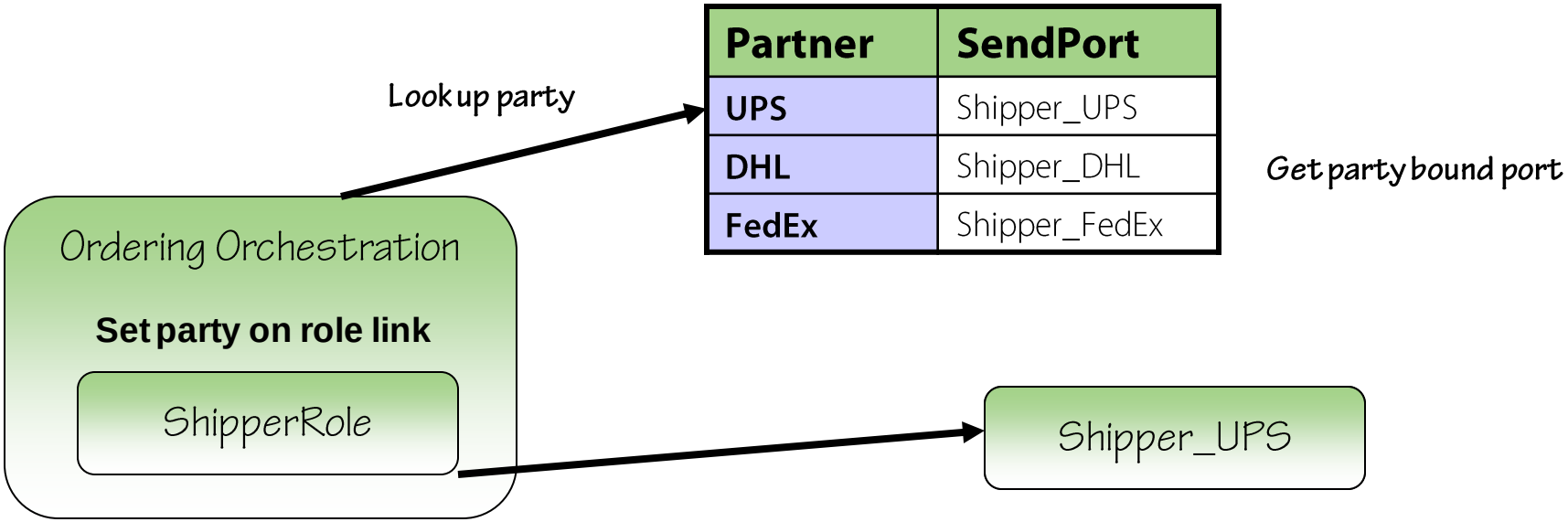
Direct binding with shared ports

- **Allows for starting partner orchestration with a message**
  - Similar to StartOrchestration shape
  - Use single port type and define port in each orchestration
  - Use the same port to define the binding in both orchestrations
  - Reference can be from sender to receiver or vice versa



Role Links

- **Define a process that can apply to multiple partners or services**
  - Role links contain send and receive port types
- **The orchestration and the partner organization each play a role**
  - Provider – orchestration receives and then sends
  - Consumer – orchestration sends and then receives



Benefits of Role Links

- **Selection of the physical ports based on the current partner**
  - Partner can be another organization, department, or application
  - Can be used in place of dynamic ports if partners are defined
  - Party identification can use rules or custom logic
- **Adding a new partner**
  - Define partner
  - Edit aliases for the partner
  - Enroll partner in the orchestration role
  - No changes to the orchestration needed

Choosing partners for role links

- **Party identifiers or aliases are used to determine the partner**
- **Receive ports**
  - Party identification handled in the receive pipeline
  - Any receive port can be used by any partner or application
  - Requires that the receiving host is Authentication Trusted
- **Send ports**
  - Set the DestinationParty property on the role link in orchestration

```
TradingPartnerRoleLink(Microsoft.XLANGs.DestinationParty) =
       new Microsoft.XLANGs.BaseTypes.Party("keyvalue", "alias");
```

Dynamic binding options

- **Message Box**
  - You want the loose coupling of publish and subscribe
- **Dynamic addressing**
  - You will set the address and/or transport dynamically at runtime
- **Shared ports**
  - You want to subscribe to messages from an orchestration
- **Self-correlating ports**
  - You don't have a common property to correlate on
  - You are calling the orchestration and can pass the port as a param
- **Role links**
  - You are creating a reusable process used with several "partners"
  - You can define the parties and configure identifiers for them
  - You can determine at runtime the current "partner" in the process

Executing pipelines in orchestrations

- **Why use pipelines in an orchestration?**
  - Handle interchange as individual messages, but in one orchestration
  - You need control over assembling a multi-part message.
  - You need to "flatten" a particular message part (flat file)
  - Perform other pipeline activities to a non-body message part

Executing pipelines in orchestrations

- **New namespace Microsoft.XLANGs.Pipeline and classes for executing pipelines in orchestrations**
  - Use the *XLANGPipelineManager* to execute pipelines
  - For receive pipelines, execute in an expression shape
  - For send pipelines, execute in a message assignment shape

Executing pipelines in orchestrations

- **Executing Receive Pipelines**
  - Declare an orchestration variable of type Microsoft.XLangs.Pipeline.ReceivePipelineOutputMessages to collect the messages at the end of the pipeline.
  - Call ExecuteReceivePipeline
  - Iterate over the messages with a looping shape, if needed

```
//initialize the collection of output messages
outputMessages = null;

//execute the pipeline
Microsoft.XLANGs.Pipeline.XLANGPipelineManager.ExecuteReceivePipeline(
  typeof(PS.Pipelines.POBatchReceivePipeline), receivedMessage,
    outputMessages);
```

Executing pipelines in orchestrations

- **Executing Send Pipelines**
  - Declare an orchestration variable of type Microsoft.XLangs.Pipeline.SendPipelineInputMessages to collect the messages that will be sent to the pipeline.
  - Call ExecuteSendPipeline
  - Send the message that was created or use it later in orchestration

```
//initialize the collection of output messages
inputMessages.Add(receivedMessage); //this might be in a loop

…

//execute the pipeline
Microsoft.XLANGs.Pipeline.XLANGPipelineManager.ExecuteSendPipeline(
  typeof(PS.Pipelines.POFlatteningSendPipeline), inputMessages,
    outputMessage);
```

Convoy Messaging Patterns

- **A convoy is a special type of message pattern using correlation**
  - Convoys are used to address race conditions in messaging
  - Convoys setup instance subscriptions
  - At routing, convoy information is used to correlate messages
- **Two types of convoys**
  - Sequential
  - Parallel

Sequential convoys

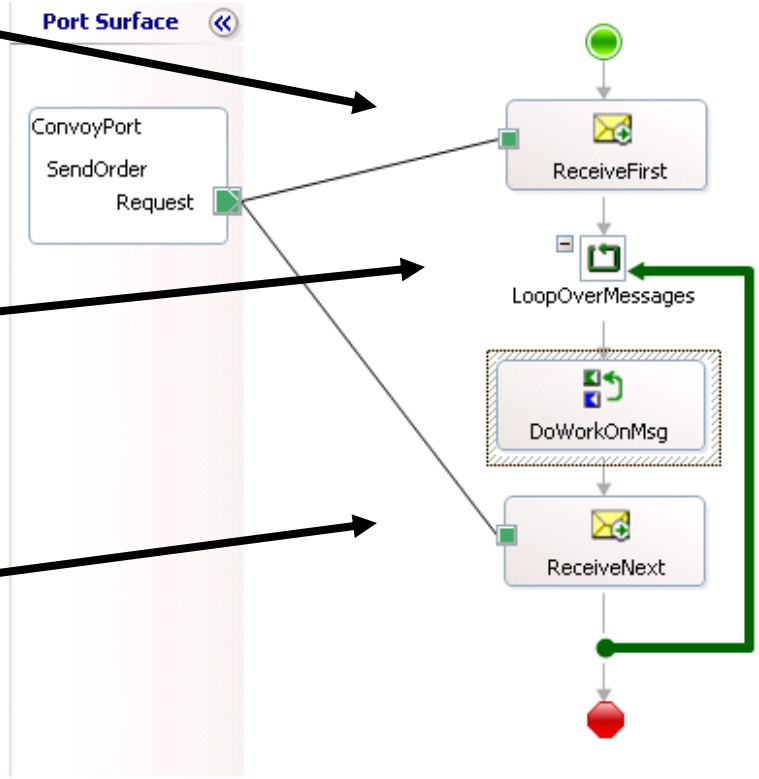
- **Messages are received in a series**
  - Process messages in a loop until known stop condition
  - Messages must be received from the same port
- **Uniform sequential convoy**
  - All messages are of the same type
- **Non-uniform sequential convoys**
  - Involve different message types

Implementing Sequential Convoys

Initialize Correlation Set

Determine how many messages to process

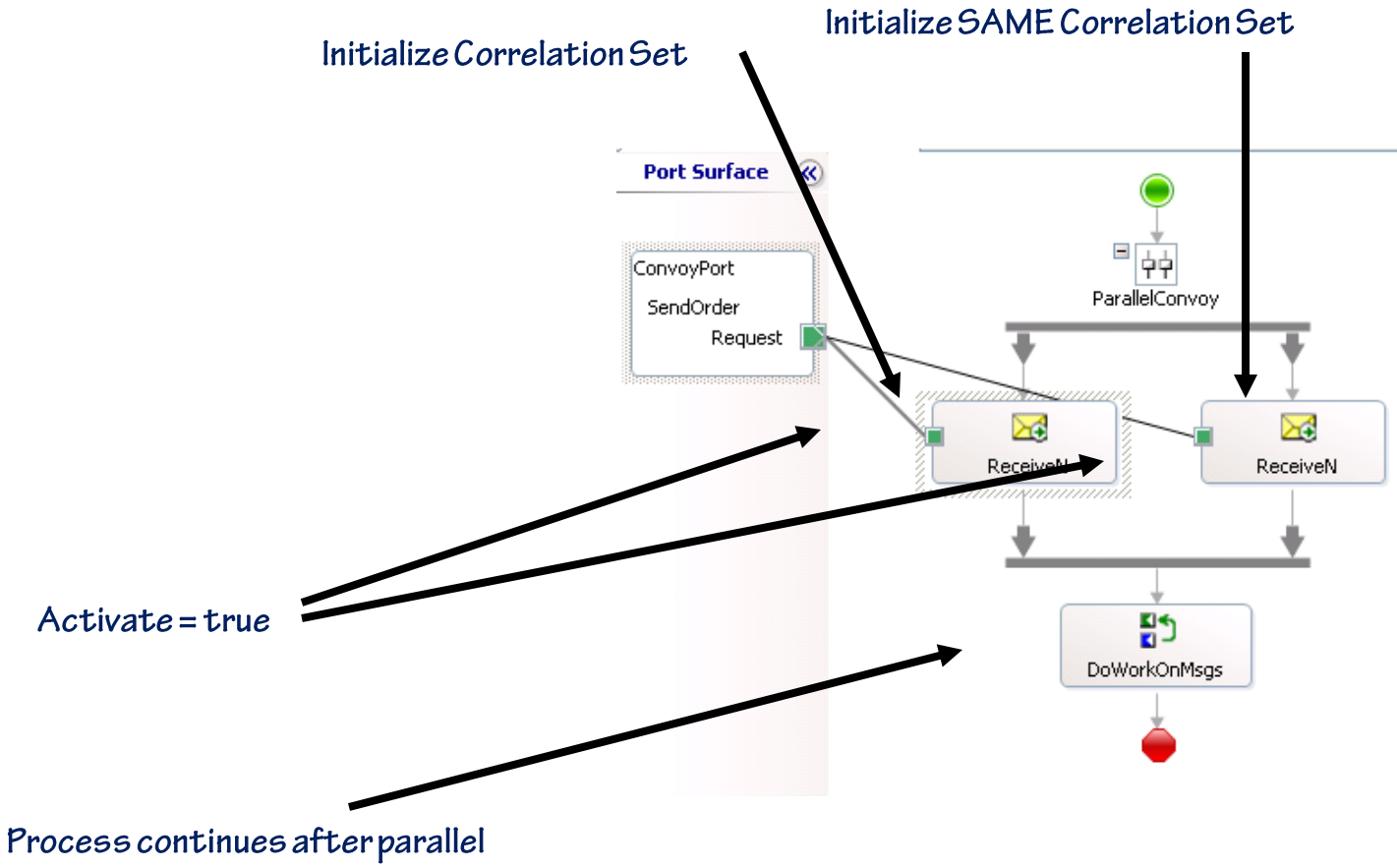
Follow Correlation Set



Parallel convoys

- **Different messages to be received in unknown order**
  - Any message can be received first
  - Must know how many messages are coming at design time
  - Processing continues once all the messages have arrived

Implementing Parallel Convoys



Summary

- Exception handling is modeled and straight forward
- Transaction support allows for atomic and long running tx
- Direct port binding provides flexibility and loose coupling
- Using parties and role links allows process reuse
- Pipelines can be useful within orchestrations
- Convoys are messaging patterns in orchestration

Resources

- **BizTalk developer center orchestration learning**
  - http://msdn.microsoft.com/biztalk/learning/dev/orch/default.aspx
- **Convoy deep dive whitepaper**
  - http://msdn.microsoft.com/library/en-us/BTS_2004WP/html/956fd4cb-aacc-43ee-99b6-f6137a5a2914.asp