Service Broker Guide

Sergey Minyurov

Moscow, 2019, version 1.2

Content

Thankfulness

It is wonderful to have friends and colleagues who, despite the constant fuss and everyday life, develop, retain a keen interest in the profession and share their thoughts, ideas and problems.

Talib Gasanov is a very interesting interlocutor as an IT manager who has a broad outlook. Despite the organizational load, he always tries to use new interesting technologies. Keeps a taste for programming and energizes.

I always wonder with Talib to discuss different topics and tasks. From time to time he gives me professional creative projects. This guide is the product of just such a project.

Purpose

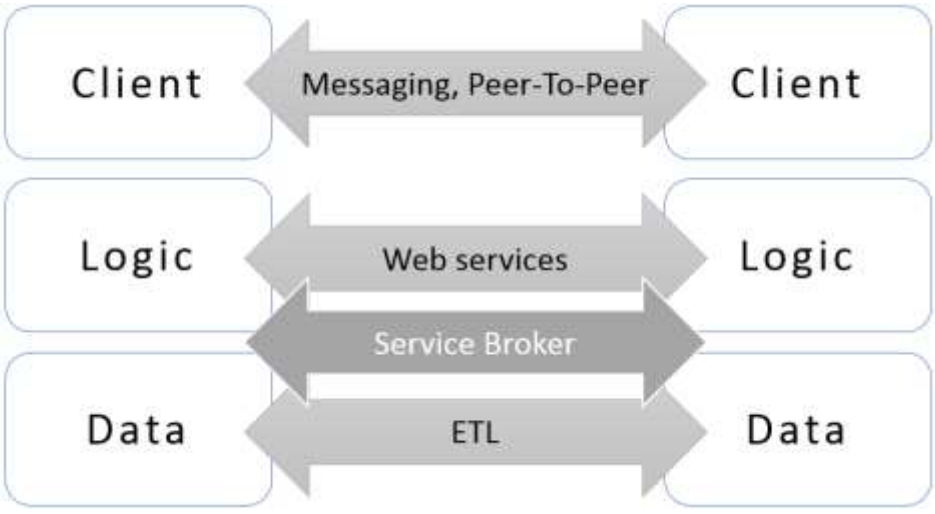
Service Broker is a software technology for developing integration solutions. To understand its purpose, it is important to consider the overall objectives and technologies. Integration can be performed at different levels of the information system architecture and in various ways:

Integration task	Technology	Aspects		
		Data	Logic	Interaction
Distributed database	Replication	Large volume, complex structure	Simple	Synchronous / asynchronous
Data transfer	ETL	Large volume, complex structure	Simple	Asynchronous
The working process	SOA (web services)	Medium volume, complex structure	Complex, distributed	Synchronous / asynchronous
	Service Broker	Medium volume, complex structure		
Communication	Messaging, Peer-To-Peer	Small volume, simple structure	Simple, distributed	Synchronous

When choosing a solution, it is important to distinguish between tasks 1) data processing and 2) process execution. Service Broker is a hybrid technology that provides the processing of large amounts of data with a complex structure, and the execution of complex distributed logic (Workflow). In this way, Service Broker combines the functionality of ETL and web services.

The principal limitation is to work only on the SQL Server platform.

The closest analogue is SOA (web services), which, unlike Service Broker, is a cross-platform solution and can be a simpler solution when building an integration solution with external agents.



Criteria for choosing a solution development on Service Broker:

Use	**Do not use**
Load balancing	All local tasks must be performed synchronously.
Distributed workflow	Only data exchange required
Implementation within the corporate system on the platform of Microsoft SQL Server	It is required to implement the solution regardless of databases and platforms, incl. for external contractors

The universal criterion is the relative simplicity in the development and support of the solution, its organic inclusion in the existing information system.

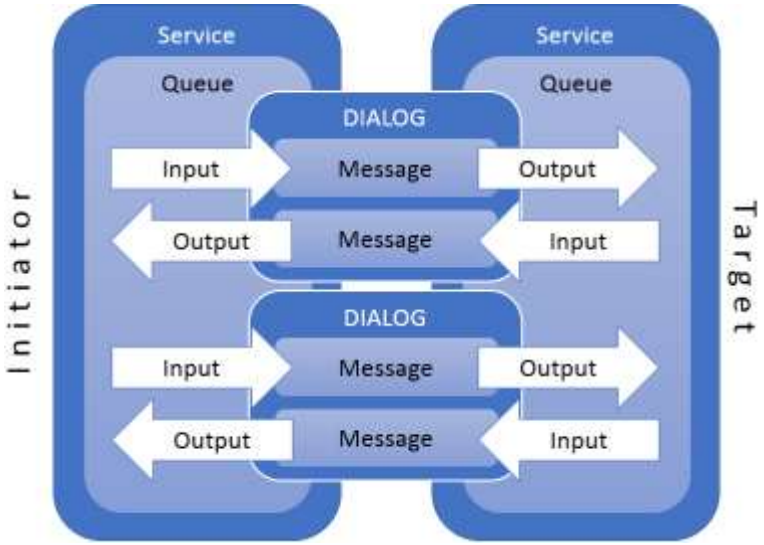
Functionality

Service Broker implements asynchronous and distributed programming on the server side of the database. This allows you to implement end-to-end processes that run on different database servers. It also provides load balancing, allowing you to separate the synchronous execution of critical tasks, and queue tasks that do not need to be performed immediately

Asynchronous programming is based on **messages** that are part of a **dialogue** (distributed unit of work). Message repositories are queues. From the point of view of a particular system, messages can be outgoing - sent to another system, and incoming - received from another system. The system that creates the dialogue is the **initiator**. The other side is the **target**.

> To balance the load, the database can send messages to itself.
>
> For dialogue, **priority** can be set from 1 (low) to 10 (high). This ensures rapid processing of important messages. Default sets the priority of 5.

Each dialogue belongs to a specific conversation group associated with a particular **service**.



On each side, services are defined that are the logical address for sending messages. Each service is associated with a queue. The service on the initiating side is called the **calling service**, and on the target side is called the **target service**.

The basic dialogue scenario:

1. On the initiator side:
    1. The program starts a dialogue.
    2. The program creates a message containing the data necessary to complete the task.
    3. The component sends a message to the target service.
2. On the target side:
    1. The message is placed in the queue associated with the target service.
    2. The program receives a message from the queue and processes the data.

The target can also send a message to the initiator before completing the conversation. Thus, a cyclic dialogue can be formed between different services in the form of messages of different types. Each side after processing the last message closes the conversation.

A distributed task can be performed on several servers in different dialogs, respectively, for messages related to a common task, a common **message group** identifier is set.

You can configure message routing across multiple servers.

When developing distributed solutions, Service Broker provides reliable, secure and orderly messaging between different databases and servers based on the TCP/IP network protocol.

> Messaging is based on transactions and a queuing mechanism. A message is sent only after the transaction is committed on the initiator side. Deleting a message from the queue is also performed only after the transaction is committed on the target side. Thus, the reliability and consistency of data and message processing is ensured.
>
> Messages are delivered in the order they were sent, and only once.

Event notifications that can be configured instead of triggers for asynchronous processing are a built-in data source. They can also be used for built-in performance monitoring - as an analogue of SQL tracing.

Service Broker is used by the SQL Server mail service and for mirror databases.

Design

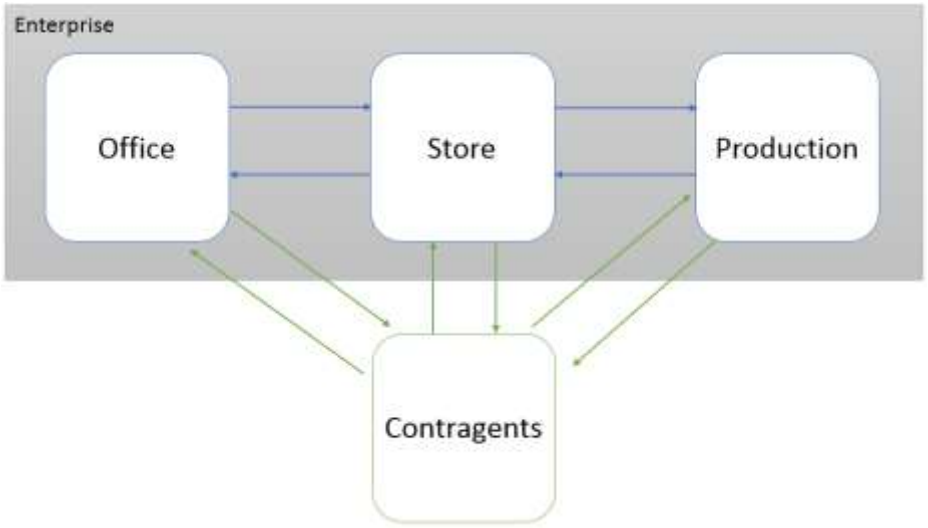
Logically, the solution at Service Broker is a cellular automaton. From the point of view of the domain area, a workflow is executed, local or distributed.

To some extent, programming is similar to developing triggers, which are stored procedures that handle data change events or server-level events.

Service Broker sends and receives messages. Messages can be sent from any stored procedure, and message processing can be implemented according to a schedule or through activation - in this case, a complete analogue occurs with triggers when an event triggers the execution of its handler.

An example of a distributed solution. The company may have several units and, accordingly, applications: trade, warehouse and production. To process orders, you may need to implement a distributed workflow based on the integration of these applications. Order processing includes planning and execution. Example of a planning process:

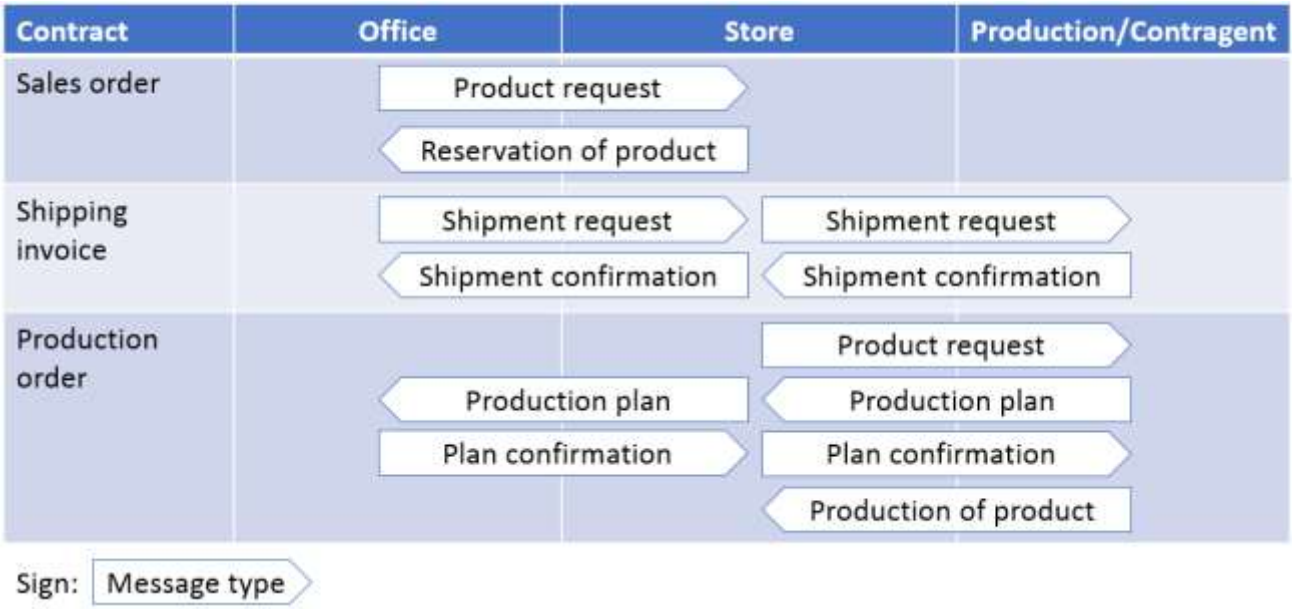
1. The client can choose products, request terms and prices.
2. At the warehouse, the availability of the required products is checked, if necessary, a request is formed to production or contragents (contractors).
3. The production or contractor submits an estimate for the timing and cost, which is then collected and transferred to the client.



When designing a solution based on Service Broker, you need to solve 2 tasks:

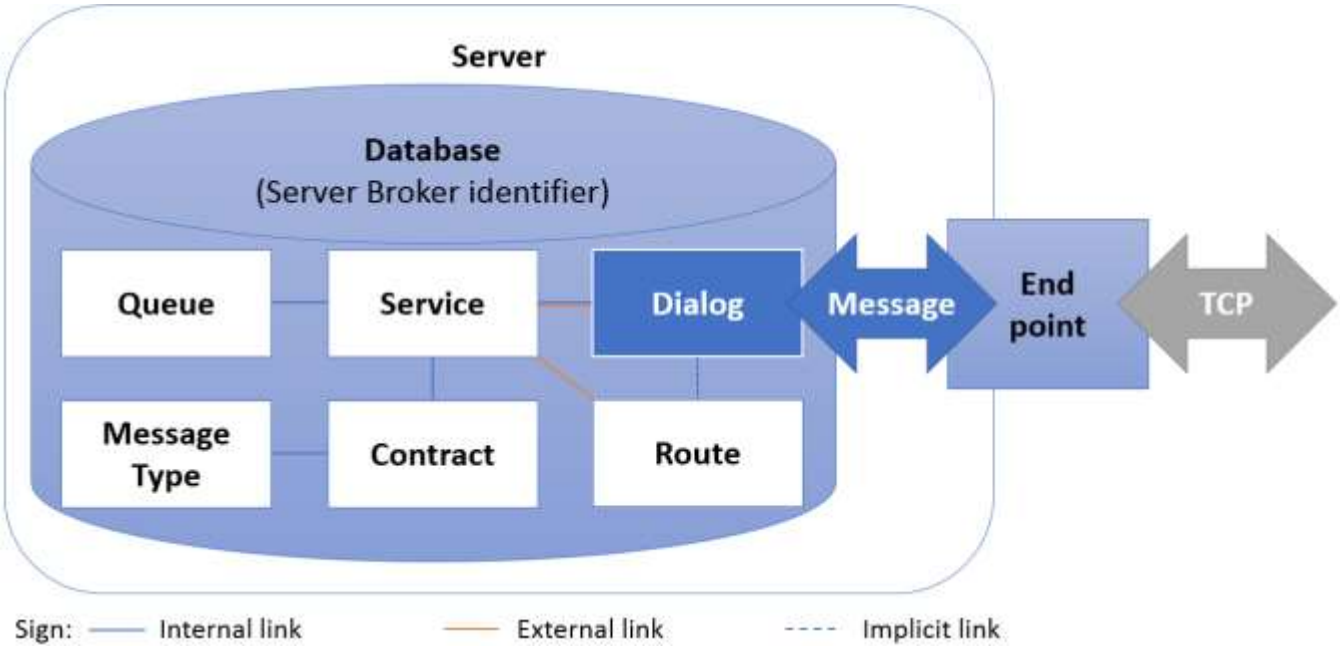
1. Determine the solution topology: which local networks, domains, servers and databases are involved in the workflow.
2. Determine the logic of the decision: what types of messages, data formats and processors need to be developed.

Using contracts, the interaction between the components is described: for example, a sales order is issued at the office, and then reservation and shipment of products are performed. Contracts define message types that describe interaction options.



Each database on SQL Server can be an application participating in a distributed process using Service Broker. Accordingly, the database has a special global unique identifier and a set of objects for generating, sending and processing messages.

To access remote services, you need to define an endpoint for the Service Broker at the database server instance level.

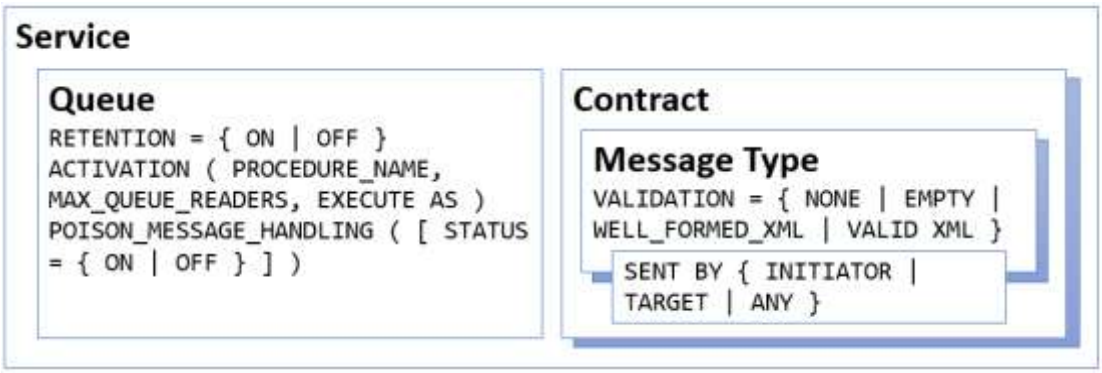


Since data is transmitted via the TCP/IP network protocol, it is necessary to configure the firewall to allow data transfer through the Service Broker specified for the connection point and enable this protocol for the database server instance.

At the database level, remote service bindings are additionally created. This defines the user associated with the certificate from which the public key is used to authenticate messages and to encrypt the session key, which is then used to encrypt the communication session.

> If anonymous authentication is used (ANONYMOUS = ON), then the calling service connects to the target service as a member of the **public** database role. By default, members of this role do not have permission to connect to the database. To successfully send a message, the target database must provide the public role with **CONNECT** permission for the database and **SEND** permission for the target service.

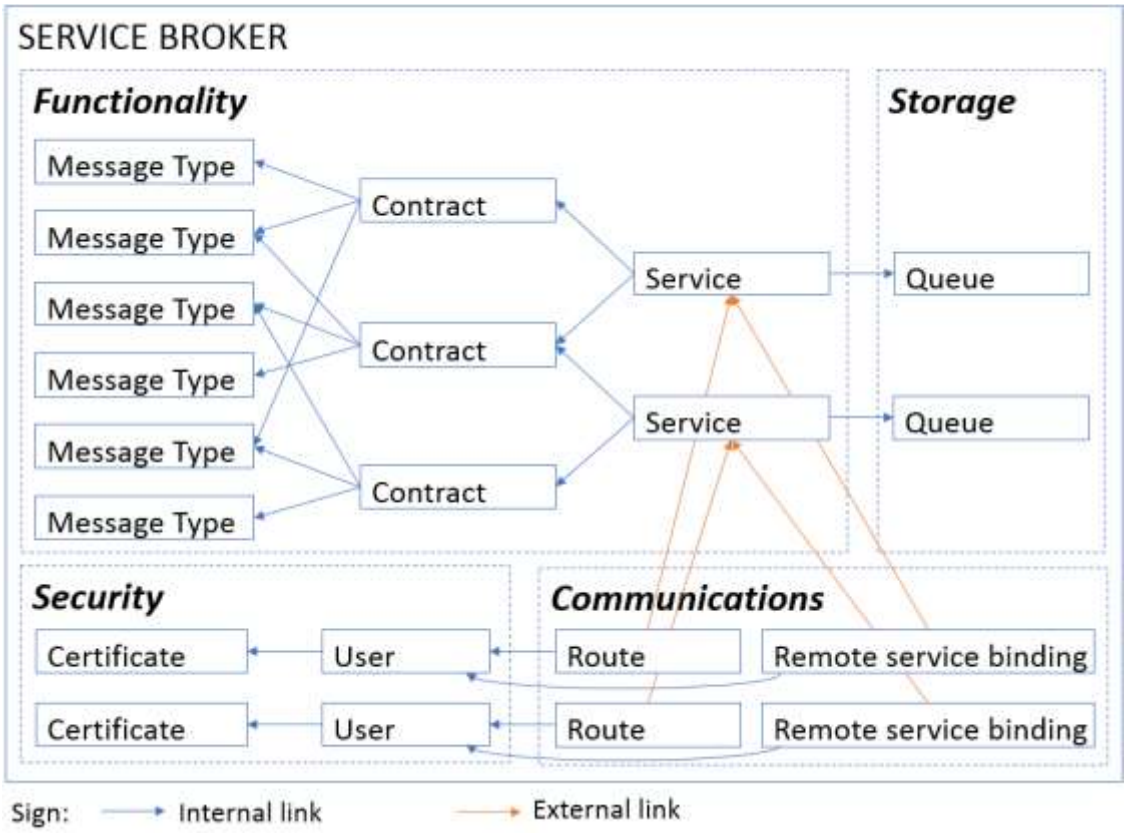
The logical composition of Service Broker objects:



Service Broker object classes:

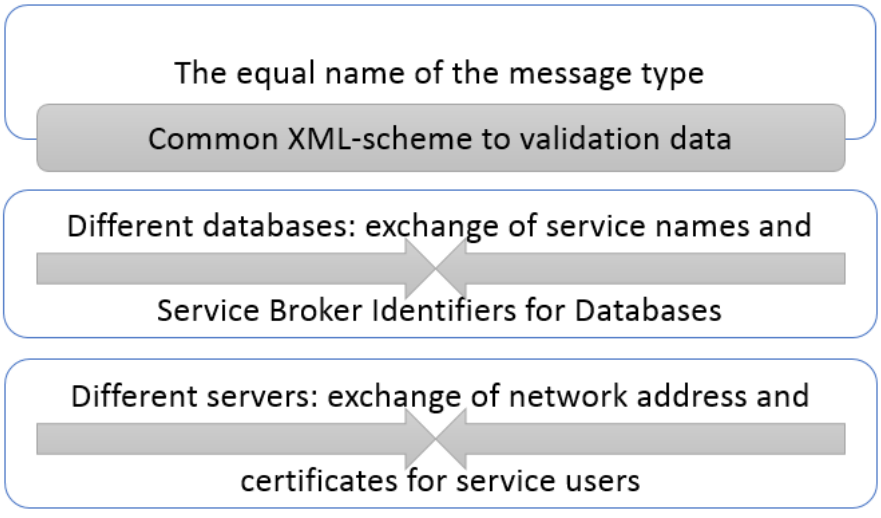
1. **Message Type** - defines the format of the message. It can be empty, arbitrary (usually for binary data transfer), XML or typed XML (based on XML schema).
2. **Queue** of messages - a repository of actual messages sent or received. After processing, messages are deleted.
   - To save messages in the queue until the dialogue is completed, set **RETENTION = ON**. This can be used to audit or perform offsetting transactions.
   - For a queue, a stored procedure can be defined as an incoming message handler (**ACTIVATION (PROCEDURE_NAME)**). You can specify the number of parallel threads (**MAX_QUEUE_READERS**) and the security context (**EXECUTE AS**) for the handler.
   - Standard failover (**POISON_MESSAGE_HANDLING**) is enabled by default. After 5 failures, the queue is disabled. If the application implements user-defined processing of error messages, then this parameter can be disabled.
3. **Contract** - combines message types into a set corresponding to a specific task. Sets the direction for each type of message: incoming (**TARGET**), outgoing (**INITIATOR**) or two-way (**ANY**).
4. **Service** - a logical access point that combines contracts and queues. The service is associated with only one queue; it allows you to specify several contracts whose messages will be stored in the corresponding queue.
   - The service may not contain contracts; in this case, it can only initiate a dialogue.
5. **Route** - connects services for sending certain types of messages. When searching for a service by name, the register is different (binary comparison is performed).

There is a message type and a default contract (**DEFAULT**) that can be used for testing or simple messages. There is also a default **AutoCreatedLocal** route, which is used to send messages inside the database.



When configuring, it is important to synchronize message types and contracts between remote services, and to configure mirroring for the services and routes themselves.

It is important to agree on the name of the message types between the participants in the integration and develop common XML schemes for all that are used to validate the data in the messages.



For different databases, you need to give each other the name of the services that receive messages and Service Broker identifiers for the databases involved in the messaging.

For different servers, you also need to exchange network addresses and certificates for service users, which are used to create connection points and external bindings to services.

Messaging on a single database server

The simplest configuration option for Service Broker runs on a single database server, as topology configuration is not required.

Single Database

To load balancing, you can configure message processing within the same database: i.e. for the application, messages will be sent to themselves. Thus, it is possible to perform some operations synchronously, as part of a user transaction, and perform other operations asynchronously, in a separate queue processing transaction.

In this case, the **AutoCreatedLocal** system route will be used. It is enough to define the types of messages, contracts, queues and services. And you do not need to create your own route.

1. Creation and configuration of the database:

```
CREATE DATABASE SBDemoLocal
ALTER DATABASE SBDemoLocal SET ENABLE_BROKER;

USE SBDemoLocal
CREATE MESSAGE TYPE DemoMessageType VALIDATION = NONE;
CREATE CONTRACT DemoContract (DemoMessageType SENT BY ANY);
CREATE QUEUE DemoQueue;

CREATE SERVICE DemoService ON QUEUE DemoQueue (DemoContract);
```

2. Start a dialogue, send a message and end conversation:

```
USE SBDemoLocal

DECLARE @h UNIQUEIDENTIFIER;
BEGIN DIALOG CONVERSATION @h FROM SERVICE [DemoService] TO SERVICE 'DemoService'
ON CONTRACT [DemoContract] WITH ENCRYPTION = OFF;

SEND ON CONVERSATION @h MESSAGE TYPE [DemoMessageType] ('test message');

END CONVERSATION @h;
```

3. Receive a message and end conversation:

```
DECLARE @h UNIQUEIDENTIFIER, @msg VARCHAR(MAX);
BEGIN TRANSACTION;
  WAITFOR (
    RECEIVE TOP (1) @h = [conversation_handle], @msg = TRY_CONVERT(varchar, [message_body])
    FROM [DemoQueue]
  ), TIMEOUT 1000
  IF @@ROWCOUNT <= 0 BEGIN
    PRINT 'No messages'
    ROLLBACK TRANSACTION;
    RETURN;
  END;
  PRINT @msg
  END CONVERSATION @h;
COMMIT TRANSACTION;
```

Accordingly, we get the opportunity to split task execution between different transactions (asynchronously), and minimize data processing in a user transaction.

Single server, different databases

If different databases are located on the same server, then in addition you need to set the TRUSTWORTHY option for these databases and configure the route. When configuring the route, you need to specify the Service Broker identifier of the target database and the built-in system address **Local**.

Database 1	**Database 2**
1. Creating and configuring databases: need to exchange Service Broker database identifiers	

`CREATE DATABASE SBDemo1`
`ALTER DATABASE SBDemo1 SET ENABLE_BROKER;`
`ALTER DATABASE SBDemo1 SET TRUSTWORTHY ON;`
`USE SBDemo1`
`SELECT [service_broker_guid]`
`FROM sys.databases`
`WHERE database_id = DB_ID();`
`-- CA517779-C8B2-491A-BE73-00BA702DA888`
`CREATE MESSAGE TYPE DemoMessageType`
`  VALIDATION = NONE;`
`CREATE CONTRACT DemoContract (`
`  DemoMessageType SENT BY ANY);`
`CREATE QUEUE Queue1;`
`CREATE SERVICE Service1`
`  ON QUEUE Queue1 (DemoContract);`
	`CREATE DATABASE SBDemo2`
	`ALTER DATABASE SBDemo2 SET ENABLE_BROKER;`
	`ALTER DATABASE SBDemo2 SET TRUSTWORTHY ON;`
	`USE SBDemo2`
	`SELECT [service_broker_guid]`
	`FROM sys.databases`
	`WHERE database_id = DB_ID();`
	`-- B58A5964-3B95-45E3-AA21-55525EB6DD5E`
	`CREATE MESSAGE TYPE DemoMessageType`
	`  VALIDATION = NONE;`
	`CREATE CONTRACT DemoContract (`
	`  DemoMessageType SENT BY ANY);`
	`CREATE QUEUE Queue2;`
	`CREATE SERVICE Service2`
	`  ON QUEUE Queue2 (DemoContract);`

1.1. Setting routes: use Service Broker identifier of the target:	

`CREATE ROUTE Route1 WITH`
`  SERVICE_NAME = 'Service2',`
`  ADDRESS = 'Local',`
`  BROKER_INSTANCE =`
`  'B58A5964-3B95-45E3-AA21-55525EB6DD5E'`
	`CREATE ROUTE Route2 WITH`
	`  SERVICE_NAME = 'Service1',`
	`  ADDRESS = 'Local',`
	`  BROKER_INSTANCE =`
	`  'CA517779-C8B2-491A-BE73-00BA702DA888'`

2. Start a dialogue, send a message and end a conversation:	

`USE SBDemo1`
`DECLARE @h UNIQUEIDENTIFIER;`
`BEGIN DIALOG CONVERSATION @h`
`FROM SERVICE Service1 TO SERVICE 'Service2'`
`ON CONTRACT DemoContract`
`WITH ENCRYPTION = OFF;`
`SEND ON CONVERSATION @h MESSAGE TYPE`
`DemoMessageType ('test message from db 1');`
`END CONVERSATION @h;`	

3. Receive a message and end a conversation:	

	`USE SBDemo2`
	`DECLARE @h UNIQUEIDENTIFIER, @msg`
	`VARCHAR(MAX);`
	`BEGIN TRANSACTION;`
	`  WAITFOR (`
	`  RECEIVE TOP (1)`
	`  @h = [conversation_handle]`
	`  , @msg = TRY_CONVERT(varchar,`
	`[message_body])`
	`  FROM [Queue2]`
	`  ), TIMEOUT 1000`
	`  IF @@ROWCOUNT <= 0 BEGIN`
	`  PRINT 'No messages'`
	`  ROLLBACK TRANSACTION;`
	`  RETURN;`
	`  END;`
	`  PRINT @msg`
	`  END CONVERSATION @h;`
	`COMMIT TRANSACTION;`

Messaging on different database servers

Before configuring, you need to check the firewall for data transmission through the ports specified for the Service Broker connection point and enable TCP/IP for database server instances. If Kerberos is used, then Service Broker setup is simplified. In this example, we are considering a more versatile and complex option for setting up messaging using certificates.

When exchanging messages between databases located on different database servers, you need to configure endpoint. If necessary, you can configure message encryption at the transport or session level. For encryption at the transport level, you must attach a certificate to it when creating a connection point. For encryption at the session level, you need to create service users at the database level and bind certificates to them (see the example below).

Server 1	**Server 2**
1. Creating a master key and certificate for the endpoint:	

`CREATE MASTER KEY ENCRYPTION`
`  BY PASSWORD = 'Pa$$w0rd';`
`CREATE CERTIFICATE Endpoint1Cert WITH`
`  SUBJECT = 'For Service Broker endpoint';`
 	
`CREATE ENDPOINT Endpoint1`
`STATE = STARTED`
`AS TCP (LISTENER_PORT = 4022)`
`FOR SERVICE_BROKER (`
`  AUTHENTICATION = CERTIFICATE Endpoint1Cert);`
	`CREATE MASTER KEY ENCRYPTION`
	`  BY PASSWORD = 'Pa$$w0rd';`
	`CREATE CERTIFICATE Endpoint2Cert WITH`
	`  SUBJECT = 'For Service Broker endpoint';`
	
	`CREATE ENDPOINT Endpoint2`
	`STATE = STARTED`
	`AS TCP (LISTENER_PORT = 4022)`
	`FOR SERVICE_BROKER (`
	`  AUTHENTICATION = CERTIFICATE Endpoint2Cert);`

2A. Setting for anonymous dialogue:	
`GRANT CONNECT ON ENDPOINT::Endpoint1 TO public;`	`GRANT CONNECT ON ENDPOINT::Endpoint2 TO public;`
2B. Setting for dialogue encryption: need to exchange certificate files:	

`BACKUP CERTIFICATE Endpoint1Cert`
`TO FILE = 'X\Endpoint1Cert.cert';`
 	
`CREATE LOGIN SBLogin2 WITH PASSWORD =`
`'Pa$$w0rd';`
`CREATE USER SBUser2 FOR LOGIN SBLogin2;`
`CREATE CERTIFICATE Endpoint2Cert`
`AUTHORIZATION SBUser2`
`FROM FILE = 'X\Endpoint2Cert.cert';`
 	
`GRANT CONNECT ON ENDPOINT::Endpoint1 TO`
`SBLogin2;`
	`BACKUP CERTIFICATE Endpoint2Cert`
	`TO FILE = 'X\Endpoint2Cert.cert';`
	
	`CREATE LOGIN SBLogin1 WITH PASSWORD =`
	`'Pa$$w0rd';`
	`CREATE USER SBUser1 FOR LOGIN SBLogin1;`
	`CREATE CERTIFICATE Endpoint1Cert`
	`AUTHORIZATION SBUser1`
	`FROM FILE = 'X\Endpoint1Cert.cert';`
	
	`GRANT CONNECT ON ENDPOINT::Endpoint2 TO`
	`SBLogin1`

3. Creating and configuring databases: need to exchange the identifiers of Service Broker databases	

`CREATE DATABASE SBDemo1;`
`ALTER DATABASE SBDemo1 SET ENABLE_BROKER;`
`ALTER DATABASE SBDemo1 SET TRUSTWORTHY ON;`
 	
`USE SBDemo1`
`SELECT service_broker_guid FROM sys.databases`
`WHERE database_id = DB_ID()`
`-- D6D29F2F-AAA0-493C-92B0-701E29B10EB7`
 	
`CREATE MESSAGE TYPE DemoMessageType VALIDATION`
`= NONE;`
 	
`CREATE CONTRACT DemoContract (`
`  DemoMessageType SENT BY ANY);`
`CREATE QUEUE Queue1;`
`CREATE SERVICE Service1`
`  ON QUEUE Queue1 (DemoContract);`
	`CREATE DATABASE SBDemo2`
	`ALTER DATABASE SBDemo2 SET ENABLE_BROKER;`
	`ALTER DATABASE SBDemo2 SET TRUSTWORTHY ON;`
	
	`USE SBDemo2`
	`SELECT service_broker_guid from sys.databases`
	`WHERE database_id = DB_ID()`
	`-- C6938686-BAA3-41A3-8186-72D57278FAC9`
	
	`CREATE MESSAGE TYPE DemoMessageType VALIDATION`
	`= NONE;`
	
	`CREATE CONTRACT DemoContract (`
	`  DemoMessageType SENT BY ANY);`
	`CREATE QUEUE Queue2;`
	`CREATE SERVICE Service2`
	`  ON QUEUE Queue2 (DemoContract);`

Server 1	Server 2
3.1. Setting routes: use Service Broker identity of the target	
CREATE ROUTE Service2Route WITH SERVICE_NAME = 'Service2', ADDRESS = 'TCP://0.0.0.0:4022', BROKER_INSTANCE = 'C6938686-BAA3-41A3-8186-72D57278FAC9';	CREATE ROUTE Service1Route WITH SERVICE_NAME = 'Service1', ADDRESS = 'TCP://0.0.0.0:4022', BROKER_INSTANCE = 'D6D29F2F-AAA0-493C-92B0-701E29B10EB7';
4A. Setting up anonymous access to the database and local service	
GRANT CONNECT TO public GRANT SEND ON SERVICE::Service1 TO public	GRANT CONNECT TO public GRANT SEND ON SERVICE::Service2 TO public
4B. Setting access to the local service: need to exchange certificate files	

CREATE USER DemoUser1 WITHOUT LOGIN;
GRANT CONTROL ON SERVICE::Service1
  TO DemoUser1;
 	
CREATE MASTER KEY ENCRYPTION
  BY PASSWORD = 'Pa$$w0rd';
 	
CREATE CERTIFICATE DemoUser1Cert
  AUTHORIZATION DemoUser1
  WITH SUBJECT = 'For SB Service ';
BACKUP CERTIFICATE DemoUser1Cert
  TO FILE = 'X\DemoUser1Cert.cert';
	CREATE USER DemoUser2 WITHOUT LOGIN;
	GRANT CONTROL ON SERVICE::Service2
	  TO DemoUser2;
	
	CREATE MASTER KEY ENCRYPTION
	  BY PASSWORD = 'Pa$$w0rd';
	
	CREATE CERTIFICATE DemoUser2Cert
	  AUTHORIZATION DemoUser2
	  WITH SUBJECT = 'For SB Service';
	BACKUP CERTIFICATE DemoUser2Cert
	  TO FILE = 'X\DemoUser2Cert.cert';

5. Configuring access to a remote service	

CREATE USER DemoUser2 WITHOUT LOGIN;
CREATE CERTIFICATE DemoUser2Cert
  AUTHORIZATION DemoUser2
  FROM FILE = 'X\DemoUser2Cert.cert';
CREATE REMOTE SERVICE BINDING Service2Binding
  TO SERVICE 'Service2'
  WITH USER = DemoUser2, ANONYMOUS = ON / OFF;
	CREATE USER DemoUser1 WITHOUT LOGIN;
	CREATE CERTIFICATE DemoUser1Cert
	  AUTHORIZATION DemoUser1
	  FROM FILE = 'X\DemoUser1Cert.cert';
	CREATE REMOTE SERVICE BINDING Service1Binding
	  TO SERVICE 'Service1'
	  WITH USER = DemoUser1, ANONYMOUS = ON / OFF;

6. Starting a dialogue, sending a message and ending a conversation:	
USE SBDemo1 DECLARE @h UNIQUEIDENTIFIER; BEGIN DIALOG CONVERSATION @h FROM SERVICE Service1 TO SERVICE 'Service2' ON CONTRACT DemoContract WITH ENCRYPTION = OFF; SEND ON CONVERSATION @h MESSAGE TYPE DemoMessageType ('message from server 1'); END CONVERSATION @h;	
7. Receiving a message and ending a conversation:	

	USE SBDemo2
	DECLARE @h UNIQUEIDENTIFIER, @msg VARCHAR(MAX);
	BEGIN TRANSACTION;
	  WAITFOR (
	  RECEIVE TOP (1) @h = [conversation_handle]
	  , @msg=TRY_CONVERT(varchar,[message_body])
	  FROM [Queue2]
	  ), TIMEOUT 1000
	  IF @@ROWCOUNT <= 0 BEGIN
	  PRINT 'No messages'
	  ROLLBACK TRANSACTION;
	  RETURN;
	  END;
	  PRINT @msg
	  END CONVERSATION @h;
	COMMIT TRANSACTION;

Diagnostics

Diagnostic methods for Service Broker:
1. View system views or functions.
2. View events in SQL Server Profiler or Extended Events.
3. Testing with the **ssbdiagnose** utility.

When analyzing the causes of failures, first of all, you need to check the correctness of the security settings (see Security and Protection (Service Broker)) and the correspondence of the names of message types and services.

Also: Troubleshooting Concepts (Service Broker).

System Views and Functions

The following views are used to view Service Broker objects:
- Connection points for Service Broker: sys.service_broker_endpoints.
- Message types: sys.service_message_types.
  - XML schema collections for checking message format: sys.message_type_xml_schema_collection_usages.
- Contracts: sys.service_contracts.
  - Using message types for contracts: sys.service_contract_message_usages.
- Queues: sys.service_queues.
- Services: sys.services.
  - Using queues for services: sys.service_queue_usages.
  - Using contracts for services: sys.service_contract_usages.
- Routes: sys.routes.
- Remote service bindings: sys.remote_service_bindings.
- Service priorities: sys.conversation_priorities.

To analyze the operation and current status of Service Broker, the following views are used:
- Messages in the transmission queue: sys.service_broker_endpoints.
  May contain general information about data transmission problems in the **transmission_status** field. There is also an **is_conversation_error** field for filtering error messages.
- Active dialogs and conversation group: sys.conversation_endpoints.
  Allows you to analyze the current activity on the initiator or target. The **state** and **state_desc** fields describe the state of the dialog, including the presence of errors (values 'ER' and 'ERROR', respectively).

Additionally, you can use dynamic administrative views:
- Stored procedures as incoming message handlers: sys.dm_broker_activated_tasks.
- List of queue monitors that activate stored procedures to process incoming messages: sys.dm_broker_queue_monitors.
- Service Broker network connections used: sys.dm_broker_connections.
- Messages currently being forwarded: sys.dm_broker_forwarded_messages.

These views provide general information about settings and status that can be used for auditing. But they do not contain detailed information about the errors necessary to correct them.

Based on the views, you can develop audit requests or scripts as tools for monitoring and managing dialogs. Example report on services and contracts:

```
SELECT sv.name as [Service], sc.name as [Contract]
FROM sys.services sv
INNER JOIN sys.service_contract_usages scu ON scu.service_id = sv.service_id
INNER JOIN sys.service_contracts sc ON sc.service_contract_id = scu.service_contract_id
```

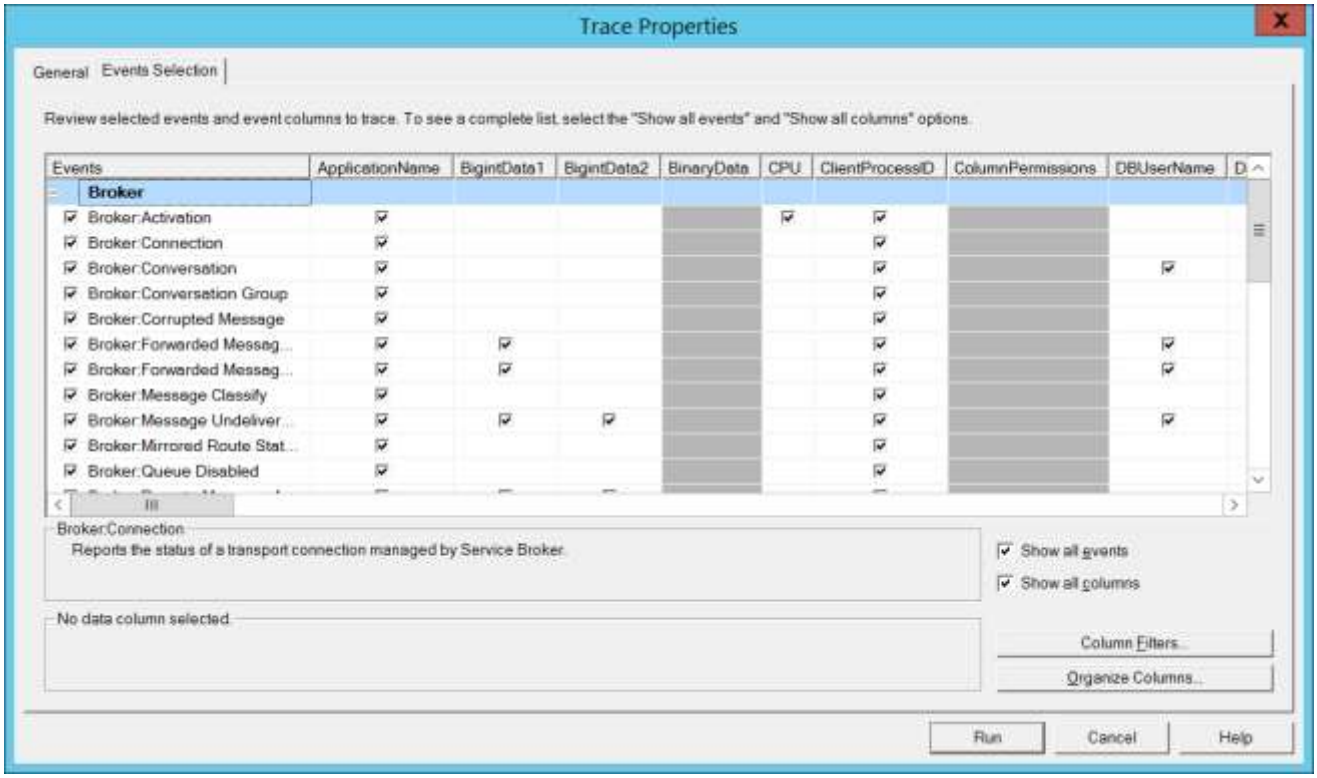
An example of a stored procedure to complete all dialogs:

```
CREATE PROCEDURE dbo.usp_EndAllConversations AS
BEGIN
    DECLARE @sql NVARCHAR(MAX) = N''
    SELECT @sql = @sql + REPLACE('END CONVERSATION "' + CONVERT(NVARCHAR(36),
[conversation_handle]) + '" WITH CLEANUP', NCHAR(34), NCHAR(39)) + NCHAR(59) + NCHAR(13) +
NCHAR(10)
    FROM sys.conversation_endpoints WHERE [state] <> 'CD';
    PRINT @sql;
    EXEC (@sql);
END
```

Using the system scalar function GET_TRANSMISSION_STATUS, you can get information about the state of the last transmission for each side of the dialogue, including error description useful for diagnosis.

Events view

Viewing events using SQL Server Profiler or Extended Events is the most useful tool for finding the cause of failures. And they need to run on each side. Usually on one side you can get a specific error message that helps to understand and localize the cause of the failure.



To view events, you need to run, for example, SQL Server Profiler on each side, connect to the server, select an empty template, and then select Service Broker events.

Diagnostic Utility

The **ssbdiagnose** command line utility is also a useful tool for testing settings and provides specific descriptions of problems. By default, displays information on the screen in the console. Using the **-XML** option, you can configure the verification result to be saved to a file.

Example command for checking settings with anonymous access:

```
ssbdiagnose CONFIGURATION FROM SERVICE Service1 -S <Computer1> -E -d SBDemo1 TO
SERVICE Service2 -S <Computer2> -E -d SBDemo2 ON CONTRACT DemoContract ENCRYPTION
ANONYMOUS
```

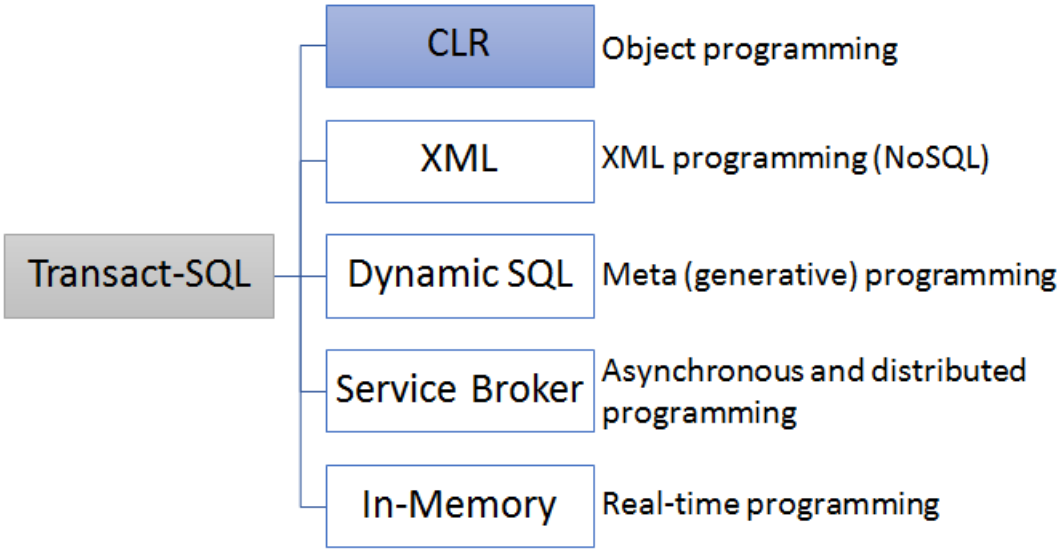
The **-SHOWEVENTS** option also allows you to include SQL Server Profiler events in the result.

Programming model

The advantage of Service Broker as an integration technology is the ability to work directly with the database and the use of Transact-SQL as a programming language. SQL is a very compact and expressive language for business logic. Modern databases offer developers a hybrid programming model that fundamentally extends traditional queries and modular algorithms.

Microsoft offers all of these programming techniques right out of the box, without the need to purchase and deploy additional products, like other companies have.

> In-memory programming is available in all editions of SQL Server, starting from version 2016, or only in commercial (paid) editions in older versions, starting from 2012, when it appeared.



At the same time, the difference between datalogical and object thinking remains: if we need to use program objects in the database to expand standard features, then we must use a separate .NET programming technology. Then the object components integrate very well into the database, and we can use these objects when programming in Transact-SQL.

> Thus, based on Microsoft SQL Server, we can develop a full-fledged application server, or integration server. The only question is load balancing, our experience and approach to solving problems.

When developing integration solutions, we can choose several approaches (Choosing a Startup Strategy):

- Batch data processing - simple logic for exchanging data with an acceptable delay in data processing.

- Event-driven data processing: handler, stored procedure, or external program.
- One-time start (at the start of SQL Server) and continuous execution for the fastest processing of messages.

If we want to use the solution primarily for data transfer, and we do not need to process it immediately, then we can configure tasks for the SQL Server agent to send or receive messages periodically. In this case, as an alternative, you can also consider setting up linked servers.

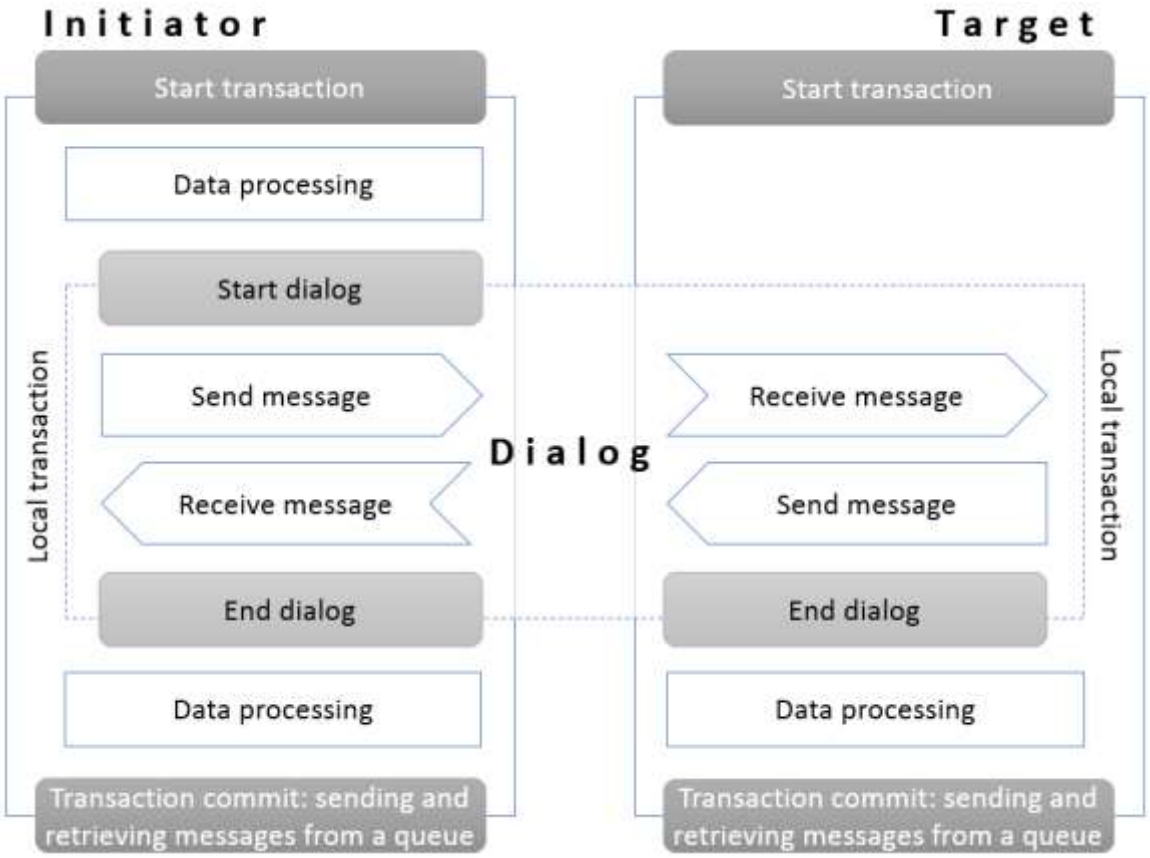
If we are developing a distributed solution, or if we want to use asynchrony to balance the load, then in this case the commands for processing messages are embedded in stored procedures or triggers.

For message queues, you can configure the automatic launch of message handlers — activation. With internal activation, a stored procedure is triggered when a message arrives. External activation generates an event for a program that runs independently of SQL Server.

Transactions and Messaging

Sending and receiving messages is logically a distributed task performed by various participants. As we said earlier, there is always an **Initiator** who starts a **Dialog**, and, on the other side, a **Target**, which continues this dialogue. When exchanging messages, all parties are equal. After completing the task, each side completes the dialogue.

For reliable data processing, each of the parties executes all the commands, including for messages in a local transaction. Accordingly, until the transaction is completed, the messages sent are accumulated in the buffer, and only when committed are sent to the recipient. If an error occurs while processing data or messages, then all commands executed within the transaction will be canceled.

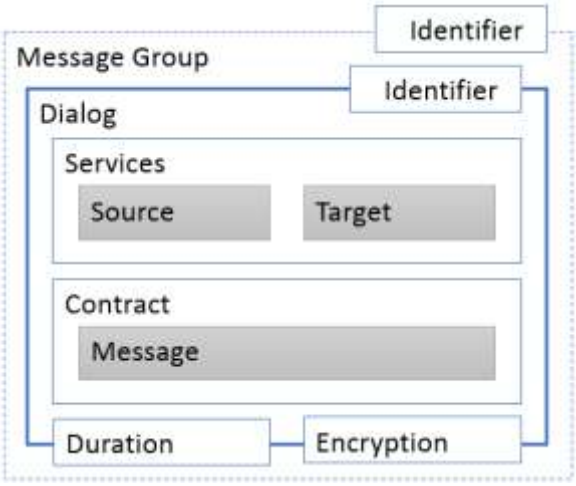


If you process data and messages in different transactions, you cannot guarantee data integrity. Unlike distributed transactions that occur, for example, when using linked servers, each side executes commands independently of the other side, which increases reliability.

Instructions for dialogs and messages

The main logical component is a **Dialog** as a task distributed among several participants. Only after creating a dialogue can you send messages and form groups of messages as a complex task that combines several distributed tasks (dialogs).

> By default, each dialog forms its own group of messages. To use a common message group in different dialogs, it is necessary to transfer the message group identifier between transactions, for example, storing it in a user table.



A dialog for the target service is created automatically when a message is received.

Dialogue management instructions:

- BEGIN DIALOG CONVERSATION - creation of a communication session between two services for the delivery of messages under a specific contract.

  For a dialog, in the **RELATED_CONVERSATION** option, you can specify the identifier of another dialog, thereby expanding the number of participants and transactions in the message exchange of the specified dialog. Also, in the **RELATED_CONVERSATION_GROUP** option, specify the identifier of the message group.

  By default, the dialog is not limited in time (the maximum value for int is set in seconds). If you use the **LIFETIME** option to specify the duration, then after a period of time if the dialog is not completed, a program error will be generated.

- BEGIN CONVERSATION TIMER - a more flexible way to control the execution time: when a specified interval has elapsed, a message of the system type http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer is placed in the local conversation queue.
- END CONVERSATION - termination of the communication session must be performed on each side. If necessary, you can specify the error code and description in the **ERROR** and **DESCRIPTION** options. There is also the **CLEANUP** option to clear the dialog when it fails.

Instructions for message groups:

- GET CONVERSATION GROUP - Get the message group identifier for a specific queue. Returns value if the queue contains messages. If there are messages from different groups, the group of messages with the highest priority is returned. An interval can be set (**TIMEOUT** option) to wait for messages to arrive.
- MOVE CONVERSATION - moves the specified dialog to a specific group of messages.

Instructions for processing messages:

- SEND - within the framework of an open dialogue, sends a message of the specified type.
- RECEIVE - within an open dialog, retrieves a message from the specified queue. On the recipient side, automatically creates a dialogue if there are messages. An interval can be set (**TIMEOUT** option) to wait for messages to arrive.

The **SELECT** statement also allows you to read messages from the queue, but does not extract them from it. Depending on the queue settings, the **RECEIVE** instruction retrieves messages or changes the status (**status = 0** field) if the queue has the **RETENTION = ON** option set.

The queue has a set of fields:

- `service_id, service_name` - identifier and name of the service;
- `service_contract_id, service_contract_name` - identifier and name of the contract;
- `message_type_id, message_type_name` - identifier and name of the message type;
- `message_body` - message content;
- `validation` - message verification: **E** = empty, **N** = no, **X** = XML;
- `status` - message status: 0 = ready, 1 = message received, 2 = not yet completed, 3 = sent message saved;
- `priority` - dialogue priority;
- `queuing_order` - message sequence number.

Programming templates

The processing of messages inside the database (internal activation) is considered.

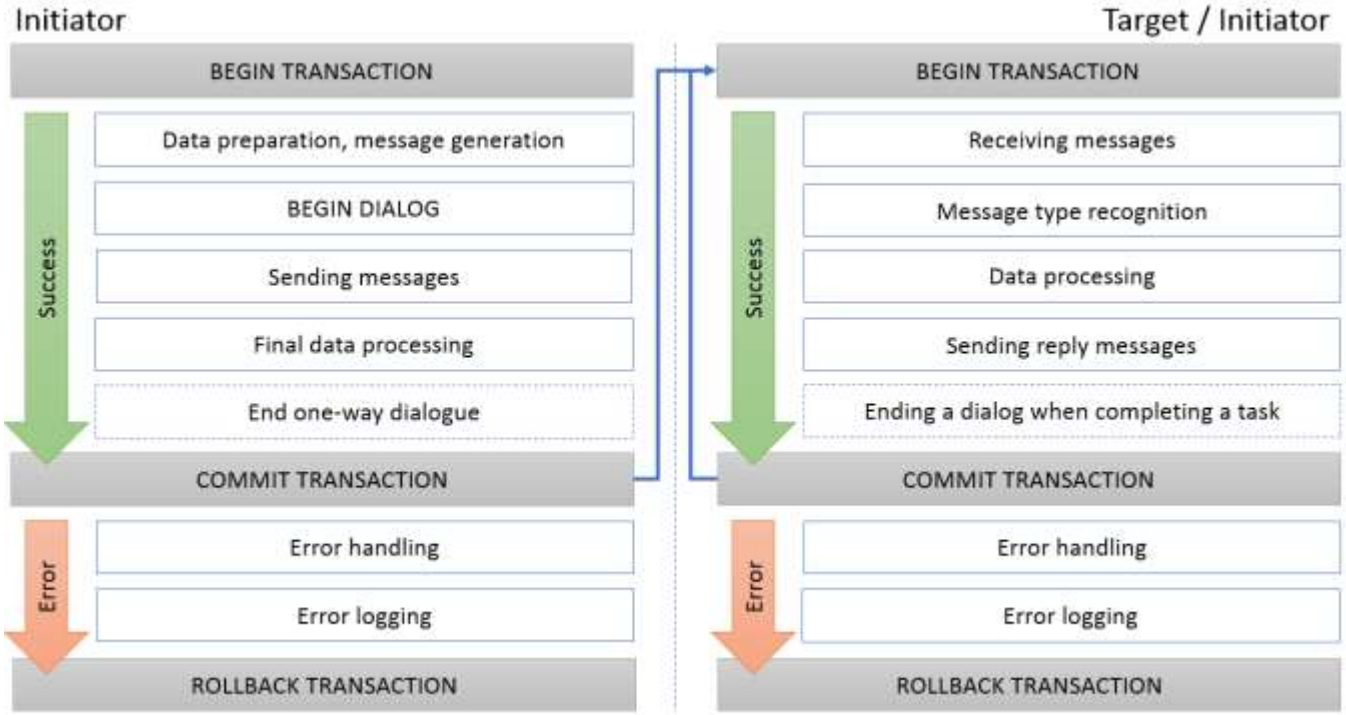
Launching a dialog and processing messages can be performed in a stored procedure or trigger, as well as in the task of the SQL agent. You cannot follow the corresponding instructions in user-defined functions.

> You can also execute instructions in manual mode, or from application code, for example, using ADO.NET.

Sending messages is a relatively simple task for which you need to compose a message, and, if necessary, specify a group of messages or connect to an existing dialog.

For error handling, the TRY..CATCH construct is recommended.

Then the transaction is launched and the instructions for managing the dialogue, processing messages and data are executed.



Script template for creating a dialogue and sending a message:

```
DECLARE @h uniqueidentifier
BEGIN TRY
  BEGIN TRAN
    BEGIN DIALOG CONVERSATION @h
    FROM SERVICE <initiator_service> TO SERVICE '<target_service>'
    ON CONTRACT <service_contract>;
    -- Data and message preparation
    SEND ON CONVERSATION @h MESSAGE TYPE <message_type_name> (<message_body>);
    -- Final data processing
    /* End one-way dialogue
    END CONVERSATION @h; */
  COMMIT
END TRY
BEGIN CATCH
  -- Error recognition and logging
  ROLLBACK
END CATCH
```

Receiving and processing messages is a more complicated procedure. Using the **RECEIVE** command, you can extract the necessary fields from the queue (see the previous section).

Messages can be retrieved from the queue in several ways:

- Extract one message (usually in a loop) into variables.
- Extract multiple messages into a table variable.
- Processing messages using the cursor.

Next, the first processing option is considered.

When a message is received using the **RECEIVE** statement, the dialogue identifier is retrieved from the queue for the response or its closure, the type of message and the message itself. Using the system function **@@ROWCOUNT**, the availability of data is checked, and then the type of message is checked, depending on which further processing is performed.

Script template for receiving a message:

```
DECLARE @h UNIQUEIDENTIFIER
DECLARE @messagetypename NVARCHAR(256)
DECLARE @messagebody XML

BEGIN TRY
  BEGIN TRANSACTION
    WAITFOR (
      RECEIVE TOP (1)
        @h = conversation_handle,
        @messagetypename = message_type_name,
        @messagebody = CAST(message_body AS XML)
      FROM TargetQueue
    ), TIMEOUT <interval>;
    IF (@@ROWCOUNT > 0) BEGIN
      IF (@messagetypename = '<message_type_name>') BEGIN
        -- Message processing
        -- Formation of a response message
        SEND ON CONVERSATION @h MESSAGE TYPE <message_type> (<message_body>);
        END CONVERSATION @h; -- End of dialogue
      END
    END
  COMMIT
END TRY
BEGIN CATCH
  ROLLBACK TRANSACTION
END CATCH
```

In the example, the message is converted immediately to XML, in more complex cases it depends on the type of message.

In addition to the messages defined in the contract, system messages may also come, for example, about an error, the end of a dialogue, or a timer event.

To handle the error after reading the message and before processing it, you can set the save point of the transaction **SAVE TRANSACTION**. In this case, if necessary, reprocessing (for example, when deadlocked), the entire transaction is canceled completely. When processing an incorrect message, error logging and transaction rollback to the recovery point are performed.

Script template for receiving messages in a loop and error handling:

```
DECLARE @h UNIQUEIDENTIFIER
DECLARE @messagetypename NVARCHAR(256)
DECLARE @messagebody XML

WHILE (1=1) BEGIN
  BEGIN TRANSACTION
    WAITFOR (
      RECEIVE TOP (1)
        @h = conversation_handle,
        @messagetypename = message_type_name,
        @messagebody = CAST(message_body AS XML)
      FROM TargetQueue
    ), TIMEOUT <interval>
    IF (@@ROWCOUNT = 0) BEGIN
      ROLLBACK TRANSACTION
      BREAK -- End processing if there are no messages in the queue
    END
    SAVE TRANSACTION MessageReceivedSavepoint
    IF (@messagetypename = '<message_type_name>') BEGIN
      BEGIN TRY
        -- Message processing
        -- Formation of a response message
        SEND ON CONVERSATION @h MESSAGE TYPE <message_type> (<message_body>);
        END CONVERSATION @h; -- End of dialogue
      END TRY
      BEGIN CATCH
      IF (ERROR_NUMBER() = 1205) BEGIN -- Deadlock
        ROLLBACK TRANSACTION
        CONTINUE -- Continuing event processing
      END ELSE BEGIN
        ROLLBACK TRANSACTION MessageReceivedSavepoint
        PRINT 'Error occurred: ' + CAST(@messagebody AS NVARCHAR(MAX))
      END
    END CATCH
  END
  IF (@messagetypename = 'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog') BEGIN
    END CONVERSATION @h;
  END
  COMMIT TRANSACTION
END
```

Message group processing

To include a dialogue in a group, you need to transfer the group identifier to the corresponding parameter (**RELATED_CONVERSATION_GROUP**):

```
DECLARE @h uniqueidentifier
DECLARE @gc uniqueidentifier = NEWID()

BEGIN TRY
  BEGIN TRAN
    BEGIN DIALOG CONVERSATION @h
    FROM SERVICE <initiator_service> TO SERVICE '<target_service>'
    ON CONTRACT <service_contract>
    WITH RELATED_CONVERSATION_GROUP = @gc;
    -- Saving a message group identifier
```

To include other dialogs in the same group, you must save the group identifier and reuse it.

To process a group of messages, you first need to request a group identifier from the queue, and then in a nested loop, request messages from the group:

```
DECLARE @cg UNIQUEIDENTIFIER;
WHILE (1 = 1) BEGIN
  BEGIN TRANSACTION;
    WAITFOR (
      GET CONVERSATION GROUP @cg FROM <queue>
    ), TIMEOUT <interval>
    IF (@@ROWCOUNT = 0) BEGIN
      ROLLBACK
      BREAK
    END

    DECLARE @h UNIQUEIDENTIFIER;
    DECLARE @messageTypeName NVARCHAR(256);
    DECLARE @messageBody XML;
    WHILE (1 = 1) BEGIN
      WAITFOR (
        RECEIVE TOP (1)
          @messageTypeName = message_type_name,
          @messageBody = CAST(message_body AS XML),
          @h = conversation_handle
        FROM <queue>
        WHERE conversation_group_id = @cg
      ), TIMEOUT <interval>
```

Notification of events about problem messages

If the queue handling option **POISON_MESSAGE_HANDLING (STATUS = ON)** is enabled for the queue, then after 5 consecutive rollbacks of the transaction, the queue will be disabled. To handle such situations, you can create a notification of events about turning off the queues and create a special queue for system messages:

```
CREATE QUEUE PoisonMessageNotifyQueue
GO
CREATE SERVICE PoisonMessageNotifyService ON QUEUE PoisonMessageNotifyQueue (
  [http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]);
GO
CREATE EVENT NOTIFICATION PoisonMessageNotification ON QUEUE TargetQueue
FOR Broker_Queue_Disabled
TO SERVICE 'PoisonMessageNotifyService', 'current database'
GO
```

After analyzing and solving problems with messages, you need to enable the message queue:

```
ALTER QUEUE TargetQueue WITH STATUS = ON
```

Error Handling

To improve diagnostics, you can create and process application errors. To send an error message, you need to use a special type of message or end the dialog with the code and description of the error:

```
END CONVERSATION @h
WITH ERROR = 4242
DESCRIPTION = 'Error message'
```

The error message will be transmitted as XML:

```
<Error xmlns="http://schemas.microsoft.com/SQL/ServiceBroker/Error">
<Code>4242</Code>
<Description>Error message</Description>
</Error>
```

When checking the type of message, you need to use the system type of error and extract the code and description of the error from the message:

```
IF (@messagetypename = 'http://schemas.microsoft.com/SQL/ServiceBroker/Error') BEGIN
  -- Extract code and error description
  SET @errorcode = (SELECT @messagebody.value(N'declare namespace
brokerns="http://schemas.microsoft.com/SQL/ServiceBroker/Error";(/brokerns:Error/brokerns:Code)[
1]', 'int'));
  SET @errormessage = (SELECT @messagebody.value('declare namespace
brokerns="http://schemas.microsoft.com/SQL/ServiceBroker/Error";(/brokerns:Error/brokerns:Descri
ption)[1]', 'nvarchar(3000)'));
  -- Logging and error handling
  END CONVERSATION @h; -- End of dialogue
END
```

References

1. Pro SQL Server 2008 Service Broker. Klaus Aschenbrenner. APRESS.
2. https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-service-broker?view=sql-server-2017
3. https://docs.microsoft.com/en-us/sql/relational-databases/service-broker/event-notifications?view=sql-server-2017