# Asynchronous procedure execution

August 5th, 2009

Code on GitHub: rusanu/async_tsql

**Update:** a version of this sample that accepts parameters is available in the post Passing Parameters to a Background Procedure

Recently an user on StackOverflow raised the question Execute a stored procedure from a windows form asynchronously and then disconnect?. This is a known problem, how to invoke a long running procedure on SQL Server without constraining the client to wait for the procedure execution to terminate. Most times I've seen this question raised in the context of web applications when waiting for a result means delaying the response to the client browser. On Web apps the time constraint is even more drastic, the developer often desires to launch the procedure and immediately return the page even when the execution lasts only few seconds. The application will retrieve the execution result later, usually via an Ajax call driven by the returned page script.

Frankly I was a bit surprised to see that the responses gravitated either around the SqlClient asynchronous methods (BeginExecute…) or around having a dedicated process with the sole pupose of maintaining the client connection alive for the duration of the long running procedure.

This problem is perfectly addressed by Service Broker Activation. Since I wanted to preserve the solution for further reference, I decided to put it in as a blog entry, with additional comments. For many of you Service Broker aficionados that read my blog regularly, this article is not innovative as is mostly a rehash of well known techniques I've been talking about on forums for many years now.

I'm going to use a table to store the result of the procedure execution. In this version I'll keep things simple by not allowing for any parameters to be passed to the procedure, nor collecting any execution result set data. So the table will only contain the procedure start time, the execution finish time and any error that occurred during the procedure execution:

```
create table [AsyncExecResults] (
        [token] uniqueidentifier primary key
        , [submit_time] datetime not null
        , [start_time] datetime null
        , [finish_time] datetime null
        , [error_number]       int null
        , [error_message] nvarchar(2048) null);
go
```

Next we're going to create the service and queue we need. I will use one single service for both roles (initiator and target) and I won't create an explicit contract, relying instead on the predefined DEFAULT contract:

```
create queue [AsyncExecQueue];
go

create service [AsyncExecService] on queue [AsyncExecQueue] ([DEFAULT]);
go
```

Next is the core of our asynchronous execution: the activated procedure. The procedure has to dequeue the message that specifies the user procedure, run the procedure and write the result in the results table. I will also deploy the error handling template I elaborated on my previous article Exception handling and nested transactions:

```
create procedure usp_AsyncExecActivated
as
begin
    set nocount on;
    declare @h uniqueidentifier
        , @messageTypeName sysname
        , @messageBody varbinary(max)
        , @xmlBody xml
        , @procedureName sysname
        , @startTime datetime
        , @finishTime datetime
        , @execErrorNumber int
        , @execErrorMessage nvarchar(2048)
        , @xactState smallint
        , @token uniqueidentifier;

    begin transaction;
    begin try;
        receive top(1)
            @h = [conversation_handle]
            , @messageTypeName = [message_type_name]
            , @messageBody = [message_body]
            from [AsyncExecQueue];
        if (@h is not null)
        begin
            if (@messageTypeName = N'DEFAULT')
            begin
                -- The DEFAULT message type is a procedure invocation.
                -- Extract the name of the procedure from the message body.
                --
                select @xmlBody = CAST(@messageBody as xml);
                select @procedureName = @xmlBody.value(
                    '(//procedure/name)[1]'
                    , 'sysname');

                save transaction usp_AsyncExec_procedure;
                select @startTime = GETUTCDATE();
                begin try
                    exec @procedureName;
                end try
                begin catch
```

```
                -- This catch block tries to deal with failures of the
procedure execution
                -- If possible it rolls back to the savepoint created
earlier, allowing
                -- the activated procedure to continue. If the executed
procedure
                -- raises an error with severity 16 or higher, it will doom
the transaction
                -- and thus rollback the RECEIVE. Such case will be a poison
message,
                -- resulting in the queue disabling.
                --
                select @execErrorNumber = ERROR_NUMBER(),
                    @execErrorMessage = ERROR_MESSAGE(),
                    @xactState = XACT_STATE();
                if (@xactState = -1)
                begin
                    rollback;
                    raiserror(N'Unrecoverable error in procedure %s: %i: %s',
16, 10,
                        @procedureName, @execErrorNumber, @execErrorMessage);
                end
                else if (@xactState = 1)
                begin
                    rollback transaction usp_AsyncExec_procedure;
                end
                end catch

                select @finishTime = GETUTCDATE();
                select @token = [conversation_id]
                    from sys.conversation_endpoints
                    where [conversation_handle] = @h;
                if (@token is null)
                begin
                    raiserror(N'Internal consistency error: conversation not
found', 16, 20);
                end
                update [AsyncExecResults] set
                    [start_time] = @starttime
                    , [finish_time] = @finishTime
                    , [error_number] = @execErrorNumber
                    , [error_message] = @execErrorMessage
                    where [token] = @token;
                if (0 = @@ROWCOUNT)
                begin
                    raiserror(N'Internal consistency error: token not found',
16, 30);
                end
                end conversation @h;
            end
            else if (@messageTypeName =
N'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog')
            begin
                end conversation @h;
            end
            else if (@messageTypeName =
N'http://schemas.microsoft.com/SQL/ServiceBroker/Error')
```

```
            begin
                declare @errorNumber int
                    , @errorMessage nvarchar(4000);
                select @xmlBody = CAST(@messageBody as xml);
                with xmlnamespaces (DEFAULT
N'http://schemas.microsoft.com/SQL/ServiceBroker/Error')
                select @errorNumber = @xmlBody.value ('(/Error/Code)[1]',
'INT'),
                    @errorMessage = @xmlBody.value
('(/Error/Description)[1]', 'NVARCHAR(4000)');
                -- Update the request with the received error
                select @token = [conversation_id]
                    from sys.conversation_endpoints
                    where [conversation_handle] = @h;
                update [AsyncExecResults] set
                    [error_number] = @errorNumber
                    , [error_message] = @errorMessage
                    where [token] = @token;
                end conversation @h;
             end
           else
           begin
                raiserror(N'Received unexpected message type: %s', 16, 50,
@messageTypeName);
           end
        end
        commit;
    end try
    begin catch
        declare @error int
            , @message nvarchar(2048);
        select @error = ERROR_NUMBER()
            , @message = ERROR_MESSAGE()
            , @xactState = XACT_STATE();
        if (@xactState <> 0)
        begin
            rollback;
        end;
        raiserror(N'Error: %i, %s', 1, 60,  @error, @message) with log;
    end catch
end
go
```

To make the procedure activated we need to attach it to our service queue. This will ensure this procedure is run whenever a message arrives to our [AsyncExecService]:

```
alter queue [AsyncExecQueue]
    with activation (
    procedure_name = [usp_AsyncExecActivated]
    , max_queue_readers = 1
    , execute as owner
    , status = on);
go
```

And finaly the last piece of the puzzle: the procedure that submits the message to invoke the desired asyncronous executed procedure. This procedure resturns an output parameter ‘token’ than can be used to lookup the asynchronous execution result.

```
create procedure [usp_AsyncExecInvoke]
    @procedureName sysname
    , @token uniqueidentifier output
as
begin
    declare @h uniqueidentifier
            , @xmlBody xml
        , @trancount int;
    set nocount on;

        set @trancount = @@trancount;
    if @trancount = 0
        begin transaction
    else
        save transaction usp_AsyncExecInvoke;
    begin try
        begin dialog conversation @h
            from service [AsyncExecService]
            to service N'AsyncExecService', 'current database'
            with encryption = off;
        select @token = [conversation_id]
            from sys.conversation_endpoints
            where [conversation_handle] = @h;
        select @xmlBody = (
            select @procedureName as [name]
            for xml path('procedure'), type);
        send on conversation @h (@xmlBody);
        insert into [AsyncExecResults]
            ([token], [submit_time])
            values
            (@token, getutcdate());
    if @trancount = 0
        commit;
    end try
    begin catch
        declare @error int
            , @message nvarchar(2048)
            , @xactState smallint;
        select @error = ERROR_NUMBER()
            , @message = ERROR_MESSAGE()
            , @xactState = XACT_STATE();
        if @xactState = -1
            rollback;
        if @xactState = 1 and @trancount = 0
            rollback
        if @xactState = 1 and @trancount > 0
            rollback transaction usp_my_procedure_name;

        raiserror(N'Error: %i, %s', 16, 1, @error, @message);
    end catch
end
```

```
go
```

To test our asynchronous execution infrastructure we create a test procedure and invoke it asynchronously. I will create two test procedures, one that simply waits for 5 seconds to simulate a ‘long’ running procedure and one that produces intentionally a primary key violation, to simulate a fault in the asynchronously executed procedure:

```
create procedure [usp_MyLongRunningProcedure]
as
begin
    waitfor delay '00:00:05';
end
go

create procedure [usp_MyFaultyProcedure]
as
begin
    set nocount on;
    declare @t table (id int primary key);
    insert into @t (id) values (1);
    insert into @t (id) values (1);
end
go


declare @token uniqueidentifier;
exec usp_AsyncExecInvoke N'usp_MyLongRunningProcedure', @token output;
select * from [AsyncExecResults] where [token] = @token;
go

declare @token uniqueidentifier;
exec usp_AsyncExecInvoke N'usp_MyFaultyProcedure', @token output;
select * from [AsyncExecResults] where [token] = @token;
go

waitfor delay '00:00:10';
select * from [AsyncExecResults];
go
```

## Activation Context

If you check the start time of the second asynchronosuly executed procedure you will notice that it started right after the first one finished. This is because we declare a `max_queue_readers` value of 1 when we set up activation on the queue. This restricts that at most one activated procedure to run at any time, effectively serializing all the asynchronously executed procedures. Whether this is desired or not depends a lot on the actual usage scenario. The limit can be increased as necessary.

If you start playing around with this method of invoking procedures asynchronously you will notice that sometimes the asynchrnously executed procedure is misteriously denied access to other databases or to server scoped objects. When the same procedure is run manually from a

query window in SSMS, it executes fine. This is caused by the EXECUTE AS context under which activation occurs. the details are explained in MSDN's Extending Database Impersonation by Using EXECUTE AS and myself I had covered this subject repeatedly in this blog. The best solution is to simply turn the trustworthy bit on on the database where the activated procedure runs. When this is not desired, or not allowed by your hosting environment, the solution is to code sign the activated procedure: Signing an activated procedure.

Using Service Broker Activation to invoke procedures asynchronously may look daunting at beginning. It sure is significantly more complex than just calling BeginExecuteNonQuery. But what needs to be understood is that this is a **reliable** way to invoke the procedure. The client is free to disconnect as soon as it commited the call to `usp_AsyncExecInvoke`. The procedure invoked *will* run, even if the server is stopped and restarted, even if a mirroring or clustering failover occurs. The server may even crash and be completely rebuilt. As soon as the database is back online, that queue will activate and invoke the asynchronous execution. Such level of reliability is difficult, if not impossible, to guarantee by using a client process.

http://rusanu.com/2009/08/05/asynchronous-procedure-execution/