How to prevent conversation endpoint leaks

March 31st, 2014

One of the most common complains about using Service Broker in production is when administrators discover, usually after some months of usage, that `sys.conversations_endpoints` grows out of control with CLOSED conversations that are never cleaned up. I will show how this case occurs and what to do to fix it.

A message exchange pattern that leaks endpoints

I have a SQL Server instance (this happens to be SQL Server 2012) on which I have enabled the Service Broker endpoint. I'm going to set up a loop back conversation, one that is forced to go on the network even though both initiator and target service are local:

source.sql

```
create database [source];
go

use [source];
go

create queue [q];
go

create service [source_service] on queue [q];
go

create route [target] with service_name = N'target_service',
        address = 'tcp://localhost:4022';
go

create procedure usp_source
as
begin
        set nocount on;
        declare @h uniqueidentifier, @mt sysname;

        begin transaction;

        receive top(1) @mt = message_type_name, @h = conversation_handle from q;
        if (@mt = N'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog' or
                @mt = N'http://schemas.microsoft.com/SQL/ServiceBroker/Error')
        begin
                end conversation @h;
        end
        commit
end
go

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

target.sql

```
create database [target];
go

use [target];
go

create queue [q];
go

create service [target_service] on queue [q] ([DEFAULT]);
go

grant send on service::[target_service] to [public];
go

create route [source] with service_name = N'source_service',
        address = 'tcp://localhost:4022';
go

create procedure usp_target
as
begin
        set nocount on;
        declare @h uniqueidentifier, @mt sysname;

        begin transaction;

        receive top(1) @mt = message_type_name, @h = conversation_handle from q;
        if (@mt = N'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog' or
                @mt = N'http://schemas.microsoft.com/SQL/ServiceBroker/Error')
        begin
                end conversation @h;
        end
        commit
end
go

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

This is pretty much as simple as it gets, is pair of services called `source_service` and `target_service` which have set up activation on their queue to simply end conversations when the `EndDialog` or `Error` messages are received (the very minimum requirement any service should handle). I'm using the `DEFAULT` contract. Next I'm going to send a message and immediately end the dialog:

dialog.sql

```
use [source];
go

begin tran;
declare @h uniqueidentifier;
begin dialog conversation @h
        from service [source_service]
        to service N'target_service'
        with encryption = off;
send on conversation @h;
end conversation @h;
commit;
go
```

Lets check the target database conversation endpoints:

check.sql

```
select  lifetime, state_desc, security_timestamp
        from [target].sys.conversation_endpoints;

lifetime                state_desc                                    security_timestamp
----------------------- --------------------------------------------- -----------------------
2082-04-18 11:41:08.987 CLOSED                                        1900-01-01 00:00:00.000

(1 row(s) affected)
```

Fire and Forget will leak CLOSED conversations

The target conversation endpoint is in CLOSED state, but notice that the `security_timestamp` field is unitialized . The `security_timestamp` exists to prevent dialog replays (either as an malicious attack or as a configuration mistake) which would cause the dialog 'resurrect' if the first message is retried (or 'replayed'). The target conversation endpoint cannot be deleted before the datetime in the security timestamp field, which makes it safe in case of retry/replay. This field is initialized when the message from the initiator contains a special flag set that instructs the target that the initiator has seen the acknowledgement of the first message and had deleted the message 0 from its transmission queue, and thus will not re-send it. When the target receives a message with this flag set, it initializes the security timestamp with current time plus 30 minutes. You may have noticed that the message exchange pattern in my example is the dreaded fire-and-forget pattern. In this pattern the initiator never has a chance to send a message with the above mentioned flag set and thus the target never has a chance to initiate the conversation security timestamp. This endpoint will be reclaimed on April 18th 2082, because that is the conversation lifetime. In case you wonder that date comes from adding MAX_INT32 (ie. 2147483647) seconds to the current date.

From an operational point of view this target endpoint is 'leaked'. It will consume DB space and the system will refrain from deleting it for quite some time. Repeat this vicious exchange pattern several thousand times per hour and in a few days your target database will simply run out of space. After frantic investigation you discover the culprit is `sys.conversation_endpoints` and you send me an email asking for solutions. Happens about once every week...

Solution 1: specify a lifetime

Using the very same setup, I'll add a trivial change: I will specify a 60 seconds lifetime for the conversation:

```
use [source];
go

begin tran;
declare @h uniqueidentifier;
begin dialog conversation @h
        from service [source_service]
        to service N'target_service'
        with encryption = off, lifetime = 60;
send on conversation @h;
end conversation @h;
commit;
go
```

After I let the conversation messages to be exchanged, I'm checking again the endpoints:

```
select  lifetime, state_desc, security_timestamp
        from [target].sys.conversation_endpoints;

lifetime                state_desc                                    security_timestamp
----------------------- --------------------------------------------- -----------------------
2082-04-18 11:41:08.987 CLOSED                                        1900-01-01 00:00:00.000
2014-03-31 08:47:37.947 CLOSED                                        1900-01-01 00:00:00.000

(2 row(s) affected)
```

Notice how the second conversation endpoint has a lifetime that is in the near future (one minute). Even though the security timestamp is uninitialized, the conversation endpoint **will be reclaimed in 30 minutes after the lifetime expires**. I just had lunch (chicken noodle soup, grill salmon and wild rice, apple tart), and when I check again, the

endpoint is gone:

```
select  lifetime, state_desc, security_timestamp
        from [target].sys.conversation_endpoints;

select getutcdate()

lifetime                state_desc                                          security_timestamp
----------------------- --------------------------------------------------- -----------------------
2082-04-18 11:41:08.987 CLOSED                                              1900-01-01 00:00:00.000

(1 row(s) affected)

-----------------------
2014-03-31 09:51:25.230

(1 row(s) affected)
```

If you don't want to change the existing message exchange pattern then adding an explicit lifetime to `BEGIN CONVERSATION` is a viable solution. You have to be careful when choosing the lifetime, the "correct" value is very much application dependent. A lifetime declares that your application is no longer interested in delivering the messages sent after the lifetime has expired. Lifetime is per conversation, not per message. If the conversation is not complete (the two endpoints did not exchange `EndDialog` messages before the lifetime has expired) then when the lifetime expires the conversation ends with error, an error message is enqueued for your service. Choosing a lifetime like 60 seconds in my example is quite aggressive, specially in a distributed environment, it does not allow much room for retries if, for example, the target is going through some maintenance downtime. You may want to specify a lifetime of several hours, or maybe one day. Again, is always entirely application specific.

Solution 2: change the message pattern

The next solution I'll propose is to change the actual message exchange pattern. Instead of closing the conversation from the initiator side immediately after the first `SEND`, let the target close the conversation when it receives the `DEFAULT` message. The initiator will in turn close its own endpoint in the activate procedure, as a result of receiving the `EndDialog` message. The setup is almost identical, with one minor different in target's activated procedure:

target.sql

```
create procedure usp_target
as
begin
        set nocount on;
        declare @h uniqueidentifier, @mt sysname;

        begin transaction;

        receive top(1) @mt = message_type_name, @h = conversation_handle from q;
        if (@mt = N'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog' or
                @mt = N'http://schemas.microsoft.com/SQL/ServiceBroker/Error' or
                @mt = N'DEFAULT')
        begin
                end conversation @h;
        end
        commit
end
go
```

And now when we send the message we no longer ending the dialog:

dialog.sql

```
use [source];
go

begin tran;
```

```
declare @h uniqueidentifier;
begin dialog conversation @h
        from service [source_service]
        to service N'target_service'
        with encryption = off;
send on conversation @h;
commit;
go
```

After we let the conversation run its flow, we'll check the target endpoint:

```
select  lifetime, state_desc, security_timestamp
        from [target].sys.conversation_endpoints;

lifetime                state_desc                                  security_timestamp
----------------------- ------------------------------------------- -----------------------
2082-04-18 13:15:39.690 CLOSED                                      2014-03-31 10:32:11.307

(1 row(s) affected)
```

Notice how with this message exchange pattern the target endpoint is in `CLOSED` state and **security timestamp is initialized**. The conversation endpoint will be removed by the system (deleted) when the security timestamp expires. Aside from fixing the endpoint leak problem, this pattern is better because it does have the >error handling related problems of the fire-and-forget pattern.

The bad idea: WITH CLEANUP

All too often the team that notices the conversation leak problem comes up with a horrible 'fix': they add the `WITH CLEANUP` cause to the `END CONVERSATION` statement in the target activated procedure. A short test seems to show the problem was solved, but in a few days the same team ends up in Tier 3 Microsoft customer support calls reporting all sort of weird and strange problems, including multiple application logic processing of the same message (replays) and warnings logged in ERRORLOG complaining about database corruption (DBCC will not report any). This happens because WITH CLEANUP clause is a very rude way to end a conversation. The distributed state embedded in the conversation is abruptly ended, without any attempt to inform the peer endpoint. The complications that can arise are endless and I don't even know all of them. Basically, all bets are off.

Conclusion

The fire-and-forget message exchange pattern is all too compelling, but it is riddled with problems. Ending the conversation from the target side is a much better approach. Additionally consider *always* specifying a conversation lifetime, one that makes sense for whatever the message represents from your business point of views. A lifetime declares "if the messages were not delivered after this time, I'm no longer interested in trying to deliver them", so consider carefully what is a good value for your case. And never ever deploy `WITH CLEANUP` in production code.

If you are already facing a situation in which the conversation endpoints have leaked then you first have to fix the application using one of the proposed solutions above (or both!). Then you can go ahead and cleanup the leaked conversations, and for that you can use `END CONVERSATION ...WITH CLEANUP`. Here is an example cleanup loop that looks for leaked endpoints and cleans them up:

cleanup.sql

```
declare @h uniqueidentifier, @count int = 0;
begin transaction;
while (1=1)
begin
        set @h = null;
        select top(1) @h = conversation_handle
                from sys.conversation_endpoints
                where state_desc = N'CLOSED'
                and security_timestamp = '1900-01-01 00:00:00.000'
                and lifetime > dateadd(month, 1, getutcdate())
```

```
        if (@h is null)
        begin
                break
        end
        end conversation @h with cleanup;
        set @count += 1;
        if (@count > 1000)
        begin
                commit;
                set @count = 0;
                begin transaction;
        end
end
commit
```

Posted in Samples, Troubleshooting

Comments are closed.

Remus Rusanu
I write database engines by day.
I use databases by night.

Recent Posts

- The cost of a transactions that has only applocks
- SQL Server 2014 updateable columnstores Q and A
- WindowsXRay is made public as Media eXperience Analyzer
- How to prevent conversation endpoint leaks
- How to read and interpret the SQL Server log

Entries (RSS)