

SQL Server Technical Article

**Summary:** Describes the idle connection resiliency feature, which allows ODBC and SqlClient data access applications to maintain their connections to SQL Server 2014 or an Azure SQL Database.

**Writer:** Luiz Fernando Santos and Matt Neerincx

**Technical Reviewer:** David Schwartz

**Published:** March 2014

**Applies to:** SQL Server 2014

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious.  No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2014 Microsoft. All rights reserved.

Contents

[Overview 2](#_Toc383091520)

[What does “ idle” means? 2](#_Toc383091521)

[How it works 2](#_Toc383091522)

[Client's idle connection reconnect logic 2](#_Toc383091523)

[Client session state handling and replay logic 2](#_Toc383091524)

[Non-recoverable session states 2](#_Toc383091525)

[General Considerations 2](#_Toc383091526)

[ODBC 2](#_Toc383091527)

[SqlClient 2](#_Toc383091528)

[Error messages 2](#_Toc383091529)

[New Client Settings 2](#_Toc383091530)

[Conclusion 2](#_Toc383091531)

# Overview

This paper describes the idle connection resiliency feature. Developers can write data access applications that enable idle connection resiliency with .NET Framework 4.5.1 (and later) Data Provider for SQL Server (SqlClient) and the Microsoft ODBC Driver 11 for SQL Server on Windows, which ships in SQL Server 2014 (for more information, see [Welcome to the Microsoft ODBC Driver 11 for SQL Server on Windows](http://msdn.microsoft.com/en-us/library/jj730314.aspx) (http://msdn.microsoft.com/en-us/library/jj730314.aspx)). To enable idle connection resiliency, you must connect either to a SQL Server 2014 or Microsoft Azure SQL Database. By reading this paper, you will be able to understand and properly use this feature in order to make the connections of your applications to SQL Server more robust.

Let’s imagine that you are a roaming worker that needs to use an Access application to connect to SQL Server. When you need to move from meeting to meeting, you normally close your notebook’s lid in order to move. In working online, every time this happens, you may end up disconnected either because your notebooks sleeps or due to blind wireless spots in your building. To avoid the hassle of being disconnected, you may choose to avoid certain places (like elevators, for example) and walk with your notebook’s lid open.

Now, imagine if you can close your lid and walk anywhere in your building (even take the elevator) and just arrive to your next meeting, open your lid and find your work there, waiting for you to continue. To address this and other scenarios when an idle connection drops, SQL Server introduced a new feature called Idle Connection Resiliency.

# What does “ idle” means?

In order to understand Idle Connection Resiliency, first we need to understand what “idle” connections are. So, an idle connection is the one that is active (opened, but not in the pool) but it’s not executing a command or waiting for data.

# How it works

The connection flow is presented in the hypothetical UML sequence diagram below:



Figure - High level sequence diagram that illustrates the execution flow

## Client's idle connection reconnect logic

The client provider (or simply client for the sake of this document) can re-establish broken connections transparently and independently of the application in certain cases, as described below:

* In t0, the client opens a new connection to the server and if succeeds, it saves context information after t1;
* In t2, the client first checks if the connection is alive; if not, it just re-establishes the connection using the information saved in the step above and starts sending data;
* Between t2 and t3, and t4 and t5, any connection failure must not be recoverable and the client must fail the same way it does today;
* Now, between t1 and t2, and t3 and t4, any connection failure can recover and the client can transparently re-establish it. It’s important to note that, if for some reason, the client fails to handle the security token, the connection becomes unrecoverable and, in case of a connection break, the client just fails the same way it does today.

Special treatment is given for handling SSL encryption during reconnect. The client and server negotiate if SSL encryption is enabled for the connection during initial connection establishment. In general, the client verifies that the same encryption level (either on or off) is used on the recovered connection, otherwise it fails the recovery.

## Client session state handling and replay logic

When a session state change occurs on the server during the running of a command, the server sends session state information to the client, which reads it and associates the session state with the current connection for later reuse.

A single command can have multiple session state changes, for example **SET ANSI\_NULLS ON** and **SET NOCOUNT ON**. Each individual state change is stored in the client.

When a broken idle connection is detected, the client executes against the following table:

|  |  |
| --- | --- |
| Condition | Action |
| Client is in a non-recoverable state on the current connection | Fails reporting the original connection broken error |
| If the client were performing a BCP write. | Fails fast reporting the original connection broken error |
| If state is recoverable. In this case, the client:* Executes the reconnection algorithm, using up to ConnectRetryCount times to attempt to recover the connection;
* Establishes the session;
* Sends the session state information to the server, in order to replay the session state.
 | Reconnects |

## Non-recoverable session states

When a client connects, it sends any existing session state information to the server, which executes the login process and restores the session state. If restoring fails, the server fails the login.

Now, if the restore succeeds, the server starts tracking and versioning the session state sent to the client in order to keep it in sync with the client. Any session state change can mark the connection as recoverable or not. These changes are calculated and sent to client at the end of command execution. As mentioned above, if the session holds a non-recoverable state and the connection breaks, the client fails and returns the error to the application.

The most common non-recoverable session states are:

|  |  |
| --- | --- |
| Temporary tables | Example: Create table #temp1 (f1 int) |
| Global and local cursors | Example #1: DECLARE CURSORExample #2: sp\_cursor\* APIs used by client drivers such as SQL Server Native Client |
| Transaction context | Example#1: begin transaction |
| Session level transaction locks | These are also transaction scoped, so transaction context covers thisExample #1: begin transaction update table1 set id=5 < – Update creates locks on table |
| Application locks (sp\_getapplock) | These are transaction as well as session scoped, see sp\_setapplock/sp\_getapplock API for details |
| EXECUTE AS/REVERT security context | EXECUTE AS allows setting the session’s security to a specified user and REVERT allows reverting this context back to defaultSupported by Windows Azure SQL Database, including REVERT WITH COOKIE |
| OLE automation handles | sp\_OACreate, etc… |
| Orphan LOB handles | Created by sp\_createorphan and sp\_droporphans stored procedures;This is a legacy API no longer used by current driver stacks |
| Prepared XML handles | sp\_xml\_preparedocument (<http://msdn.microsoft.com/en-us/library/ms187367.aspx>), which allows opening a handle to an XML document parsed by SQL Server |
| Trace flags* Session level trace flags are set per session and are not recovered
* Global trace flag options are server state and are ignored
 | dbcc traceon(-1, 4886) -- globaldbcc traceon(1234) -- session |

## General Considerations

* This capability is on by default in .NET Framework 4.5.1 (and later) Data Provider for SQL Server (SqlClient) and the Microsoft ODBC Driver 11 for SQL Server on Windows, when connecting to SQL Server 2014 or Microsoft Azure SQL Database.
* There is no special treatment for the KILL command. If the DBA kills a session using KILL <spid>, the application may try to reconnect again;
* The Change Password command disables any future reconnection attempt (even if the session state is recoverable).

### ODBC

For specific information about using idle connection resiliency with the Microsoft ODBC Driver 11 for SQL Server on Windows, see [Connection Resiliency in the Windows ODBC Driver](http://msdn.microsoft.com/en-us/library/jj730312.aspx) (http://msdn.microsoft.com/en-us/library/jj730312.aspx).

### SqlClient

It’s important to note that CLR integration Context Connections and SQL Dependency are not supported. In this case, if the connection string contains any of the new keywords introduced here, both the **SqlDependency::Start** method and the **SqlDependency(SqlCommand)** constructor will ignore them.

# Error messages

The client error messages can be found below:

|  |
| --- |
| Text |
| The connection is broken and recovery is not possible. The connection is marked by the server as unrecoverable. No attempt was made to restore the connection. |
| The connection is broken and recovery is not possible. The connection is marked by the client driver as unrecoverable. No attempt was made to restore the connection. |
| The connection is broken and recovery is not possible. The client driver attempted to recover the connection one or moretimes and all attempts failed. Increase the value of **ConnectRetryCount** to increase the number of recovery attempts. |
| The server did not acknowledge a recovery attempt, connection recovery is not possible. |
| The server did not preserve the exact client TDS version requested during a recovery attempt, connection recovery is not possible. |

The server error messages can be found below:

|  |  |  |
| --- | --- | --- |
| Error | Text | Result |
| Input SessionState data is invalid | Session recovery feature data used in login record to open or recover a connection is structurally or semantically invalid; the connection has been closed. Please contact the vendor of the client library.%.\*ls | Connection will be closed. The above error is logged in error log. No error message is sent to client. |
| On a recovered connection, negotiate to the same TDS version requested by client is not possible(resulting in a mismatch TDS version between original connection and recovered connection)CrClientSSLStateNotRecoverable | The Tabular Data Stream (TDS) version 0x%x of the client library used to recover a dead connection is unsupported or unknown. The server could not recover the connection to the requested TDS version. The connection has been closed. %.\*lsThe server did not preserve SSL encryption during a recovery attempt, connection recovery is not possible | Connection will be closed. The above error is logged in errorlog. No error message is sent to client |
| Failed to logon to the database required in SessionRecovery object during login | Reason: Failed to open the database '%.\*ls' configured in the session recovery object while recovering the connection | Connection will be closed. The above error message followed by Logon error will be sent to client. (also logged in error log). |
| Failed to logon to the database required in **SessionRecovery** object (initial copy) when we try to login again when client reuse the recovered connection from pool | Failed to open the database '%.\*ls' configured in the session recovery object while revalidating the login on the recovered connection | Connection will be closed. Logon error with the above reason will be sent to client. (also logged in error log). |
| If the recovered database have different collation than the one requested in the **SessionState** object (same error for both initial and to be copy) | The current collation did not match the database's collation as indicated by the session recovery object in the login | Connection will be closed. Logon error with the above reason will be sent to client. (also logged in error log). |
| Failed to recover **Identity\_Insert** option | An error occurred while recovering the **Identity\_Insert** session state requested by the recovery login | Connection will be closed. Logon error with the above reason will be sent to client. (also logged in error log). |
| Failed to recover @@identity session state | An error occurred while recovering the @@identity session state requested by the recovery login | Connection will be closed. Regular logon error message will be sent to client. The above code is only logged in error log. |
| When recovering session state failed, this error message is sent to the client to client (followed by regular logon failure message). | Cannot recover the session state requested by the login |  |
| @@RowCount is not recoverable by connection resiliency. If @@RowCount is part of the first batch after recovery, an exception must be raised and the request must fail. | The connection was recovered and rowcount in the first query is not available. Please execute another query to get a valid rowcount. | This will be raise when client issues request right after the connection is recovered and the batch request contains @@RowCount or rowcount\_big(). The request will fail but the connection won’t be closed. |

# New Client Settings

Two new client settings exposed as connection string keywords: ConnectRetryCount and ConnectRetryInterval are used to control the reconnect behavior.

* **ConnectRetryCount** controls the amount of reconnection attempts the client will try upon after identifying an idle connection failure. Its valid values range from 0 to 255. The maximum value is enforced to ensure that the client driver does not retry forever. It’s default is set to 1. It’s also important to note that a **ConnectRetryCount** of 0 (zero) means do not attempt to reconnect, giving the option to disable connection resiliency. In this case, ConnectRetryInterval is ignored;
* **ConnectRetryInterval** adds a wait time between each ConnectRetryCount attempt. Its range is from 1 (default) to 60 seconds and it’s applied **after** the first reconnection attempt:
* When a broken connection is detected, the client immediately attempts to reconnect (this is the first reconnection attempt and only occurs if ConnectRetryCount > 0);
* If the first reconnection attempt fails, and ConnectRetryCount > 1, then the client waits ConnectRetryInterval in order to try the second and subsequent reconnection attempts;
* The amount of time needed for the reconnection process cannot exceed the value of the command timeout. So, the command timeout must be greater than or equal to the total reconnection time (ConnectRetryCount \* ConnectRetryInterval).

# Conclusion

Summarize the key points from your document. At the end of your conclusion, place the following More Information and Feedback sections:

**For more information:**

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback.