

# Error 10054 An existing connection was forcibly closed by the remote host

Last updated by | Peter Hewitt | Mar 29, 2023 at 10:48 PM PDT

## Contents

- [Issue](#)
- [Investigation / Analysis](#)
- [RCA Template - for the prelogin scenario](#)
- [Mitigation](#)
- [More Information](#)
- [Internal Doc Reference](#)
- [Classification](#)

## Issue

The customer has issues to connect to their Azure SQL Database server. The application logs some variation of error 10054:

Error 10054: An existing connection was forcibly closed by the remote host

System.Data.SqlClient.SqlException (0x80131904): A connection was successfully established with the server, but then an error occurred during the login process. (provider: SSL Provider, error: 0 - An existing connection was forcibly closed by the remote host.) ---> System.ComponentModel.Win32Exception (0x80004005): An existing connection was forcibly closed by the remote host

## Investigation / Analysis

In most cases, this type of error is a client-side issue and is not caused by the SQL side:

- Error 10054 can be caused by a network issue, like network latency, routing issues, overloaded network hardware etc.
- It can also be a client-side error during the pre-login phase. The symptoms usually include high application CPU or I/O usage, resulting in the client taking longer than 5 seconds to respond to a pre-login ACK sent by the server. After 5 seconds, the connection is forcibly killed by the service to prevent trickle attacks.
- Error 10054 can occur if a client application is closed unexpectedly, while it was either sending data to the server or waiting for a reply from the server. It's not necessarily that the application itself crashes; it is also possible that the app kills the SQL thread without properly closing the SQL connection. This may indicate that the application has problems with its connection handling.
- If error 10054 is logged in the context of a performance issue, then the error is rather a symptom than the cause.

For troubleshooting, you should check for occurrences of SniReadTimeout that can be related to the disconnects. This can be best achieved through a client-side network packet capture (network trace), which will show you the exact steps and their timing in relation to each other. See the "More Information" section below for further background.

On the telemetry side, you may get further information from MonLogin, especially if the error occurs during the login sequence. Troubleshooting disconnects (dropped connections) however might not be as conclusive, as you'll probably be only able to confirm the disconnect but without further clues to the cause.

When possible, try to obtain call stacks for the failure if you can see the failure from your application. Typically for Web/Worker Role or Web Apps, the customer should be able to provide us with a log of the failures - which has a high chance of containing the affected `connection_id`.

### **Filter for disconnects with errors**

Note that on MonLogin, not every disconnect with error 10054 is an actual error. These might also occur if the client app drops the connection while the query was still running. Depending on how the applications handle their connections, most of the 10054 errors could be "normal" behaviour of the application.

The following Kusto query first filters the `connection_id` on errors 10054 for the server/database, then retrieves detailed information for the identified `connection_id` list:

```

let srv = "servername";
let db = "databasename";
let startTime = datetime(2023-02-14 11:00:00Z);
let endTime = datetime(2023-02-14 12:00:00Z);
let timeRange = ago(1d);
MonLogin
| where TIMESTAMP >= startTime
| where TIMESTAMP <= endTime
//| where TIMESTAMP >= timeRange
| where logical_server_name =~ srv
| where database_name =~ db
| where extra_info contains "10054" or extra_info contains "10060"
| where extra_info contains "error"
//| limit 100
| distinct connection_id
| join kind=inner (
    MonLogin
    | where TIMESTAMP >= startTime
    | where TIMESTAMP <= endTime
    //| where TIMESTAMP >= timeRange
    | extend ProxyOrRedirect = iif( result =~ "e_crContinue", "Redirect", iif( result =~ "e_crContinueSame
    | extend fedauth_library_type_desc =
        case(
            fedauth_library_type == 0, "SQL Server Authentication",
            fedauth_library_type == 2, "Token Base Authentication",
            fedauth_library_type == 3 and fedauth_adal_workflow == 1, "Azure Active Directory - Password A
            fedauth_library_type == 3 and fedauth_adal_workflow == 2, "Azure Active Directory - Integrated
            fedauth_library_type == 3 and fedauth_adal_workflow == 3, "Azure Active Directory - Universal
            strcat(tostring(fedauth_library_type) , "-" , tostring(fedauth_adal_workflow))
        )
    | extend AADUser = iif( fedauth_adal_workflow > 0 or fedauth_library_type > 0, "AAD" , "SQL")
    | extend disconnect_from = iff(extra_info contains "Svr", "TenantRing", "Client")
    | project originalEventTimestamp, type, event, error, state, is_user_error, is_success, os_error, sni_
        lookup_error_code, lookup_state, tds_flags, total_time_ms, package, NodeName, AppName, logic
        host_name, application_name, peer_address, vnet_peer_address, control_ring_address, driver_n
        instance_name, instance_port, is_contained_user, connection_peer_id, connection_id, vnet_lin
        result, ProxyOrRedirect, fedauth_library_type_desc, AADUser, message, session_elapsed_time_m
        kill_reason, extra_info, disconnect_from
) on connection_id

```

## Gateway disconnects

The following query is a variation of the query above. It specifically filters for `os_error == 10054` through these steps:

1. Get all the client IPs that are reaching the customer's logical server
2. Join this set of IPs with all the SNI read timeout errors that were logged
3. Check whether the resulting IPs are being used by other server names and prints out the count (is somebody else sharing the same masked IP?) The more `distinct_servers_same_ip_mask`, the less confidence there is on the customer actually hitting SNI Read timeouts.

```

let srv = "servername";
let startTime = datetime(2023-02-09 04:00:00Z);
let endTime = datetime(2023-02-12 05:00:00Z);
let timeRange = ago(1d);
MonLogin
| where TIMESTAMP >= startTime
| where TIMESTAMP <= endTime
//| where TIMESTAMP >= timeRange
| where logical_server_name =~ srv and database_name <> "master"
| where package == "xdbgateway"
| where event == "process_login_finish"
| summarize count() by peer_address
| extend total_completed_logins_to_server = count_
| join kind=inner (
    MonLogin
    | where sni_consumer_error <> 0 and os_error == 10054
    | summarize count(), min(TIMESTAMP), max(TIMESTAMP) by remote_host
    | extend sni_read_timeout_hits_from_ip = count_
    | extend first_sni_read_timeout_hit = min_TIMESTAMP
    | extend last_sni_read_timeout_hit = max_TIMESTAMP
    | extend peer_address = remote_host
) on peer_address
| join kind = inner (
    MonLogin
    | where event == "process_login_finish" and package == "xdbgateway" and database_name <> "master"
    | summarize dcount(tolower(logical_server_name)) by peer_address
    | extend total_distinct_servers_sharing_ip_mask = dcount_
) on peer_address
| project peer_address, total_completed_logins_to_server, sni_read_timeout_hits_from_ip,
    total_distinct_servers_sharing_ip_mask, first_sni_read_timeout_hit, last_sni_read_timeout_hit
| order by sni_read_timeout_hits_from_ip desc

```

## RCA Template - for the prelogin scenario

As a protection, Azure SQL Database endpoints forcibly terminate connections when the client login packets take several seconds to arrive to our service. We have found a condition where our telemetry is incorrectly classifying this error pattern as database unavailability, however the cause of these errors is not in the Azure SQL Database service itself.

There are 3 typical causes of this error pattern:

1. High resource contention on client VMs (CPU, threads, memory) causing clients to send packets much slower than normal.
2. Intermittent network latency.
3. Session Disconnects during updates/inserts


From our telemetry, we have found that most of the cases where this error condition had occurred happened while the client is processing the TLS handshake. Typically, this pattern indicates high single-core CPU pressure on the client VM.

## Mitigation

1. The customer needs to debug their application. A network packet capture on the client will help understanding if the application is taking a long time to send packets, or if the issue is on the network itself. Involve the Azure Networking or Windows Networking support team for assisting with the packet capture.

2. If the cause is a performance issue of an Azure client: scale up the client environment to mitigate the resource bottleneck.
3. Investigate network latency through a network packet capture. The analysis needs to be repeated on any hops between the client and Azure SQL Database, to understand which one is introducing the latency. A typical source of problems are firewalls or packet inspectors, which may either be dropping packets or not keeping up with the load. Involve the Azure Networking or Windows Networking support team for assisting with the packet capture. If the last hop between customer and SQL Database is proven to not be introducing latency via packet capture analysis, then the issue might come from within Azure.
4. Session disconnects can be reduced by requesting the customer to force the connection policy to redirect.
5. If the application is using a connection pool, it should close the connection as soon as it is no longer actively using it. Unless the application reuses the connection for another operation immediately and without pause, we recommend the following pattern:
  - Open a connection
  - Run one operation through the connection
  - Close the connection

## More Information

Robert Dorr has provided a detailed description of the SQL Server login sequence in this [blog article](#) . Although the article is for on-premise SQL Server, it shows you the individual login steps and aligns them with a sample network trace output.

See Bob's explanation about the cause of 10054 errors:

### 10054 The Direction Matters

Understanding where the RST (TCP closure) occurs is helpful in understanding the problem and how to troubleshoot it.

Each stage in the login process uses asynchronous read and write timeouts. For example, when the SQL Server sends the initial Pre-Login Response the SQL Server posts a read request using a 5 second timeout. If the first Ssl/Tls context packet takes longer than 5 seconds to arrive, from the client, SQL Server closes the TCP connection (RST occurs from the SQL Server) and the next write attempt from the SQL Client fails with a 10054 error, the connection has been reset. The Ssl/Tls reads use similar timeout mechanisms and the SQL client provider honors the 'login timeout.' For example, if the login packet is sent by the client but the SQL Server does not respond before the established login timeout (30 second default) the client closes the TCP connection (RST from the SQL Client provider) and the server records the 10054, connection reset error and terminates the login attempt.

## Internal Doc Reference

- [Closing client connection does not cancel query](#)
- [Error: A network-related or instance-specific error - Troubleshooting steps](#)

## Classification

Root Cause: Azure SQL v3/Connectivity/Network (Client)

**How good have you found this content?**



-