Error 40613, State 13

Last updated by | Amie Coleman | Mar 13, 2023 at 8:51 AM PDT

Contents

Issue

Error 40613 state 13 will be encountered when the Gateway process failed to send login data and socket to SQL Instance due to instance health or other performance related issues.

Socket Duplication errors
State 13 represents socket duplication failures on Xdbhost due to one of the following reasons:

- The SQL instance is not healthy
- There is a performance related issue on the SQL side due to excessive resource usage, which makes it hard to process incoming login requests fast enough

Note that socket duplication errors can also surface as 40613 state 14. In state 14, the SQL instance fails **after** receiving the login data and socket due to socket duplication process being stuck OR due to a client side error relating to the DAC (dedicated administrator connection) limit. Follow the DAC related TSG

Error Classification
State 13 - Occurs when failed to send Duplicate data (xdbhost/40613/13 (FailedToSendDuplicateData))

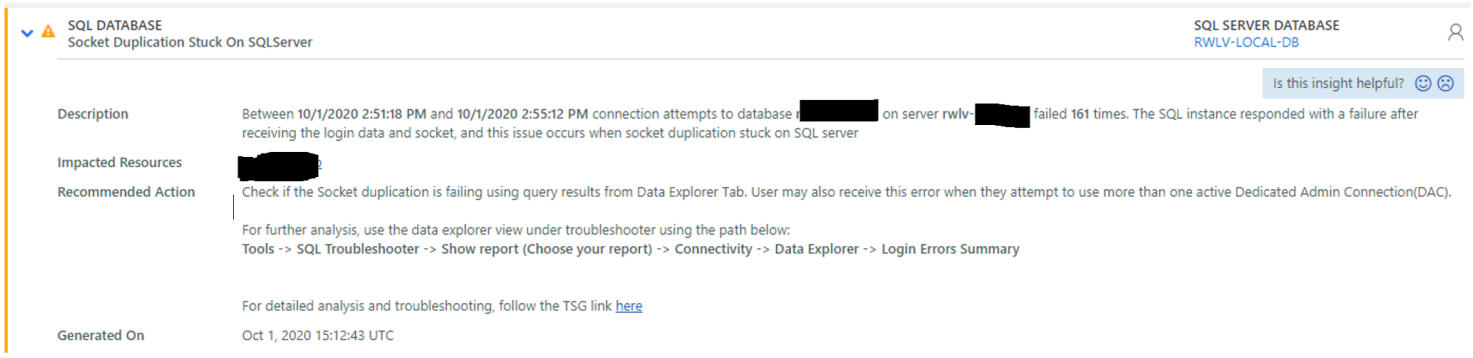
State 14 - Occurs when SQL Instance failed to duplicate data *xdbhost/40613/14 (InstanceFailedToReceiveDuplicatedData)) OR DAC Limitation issue on client side

Troubleshooting

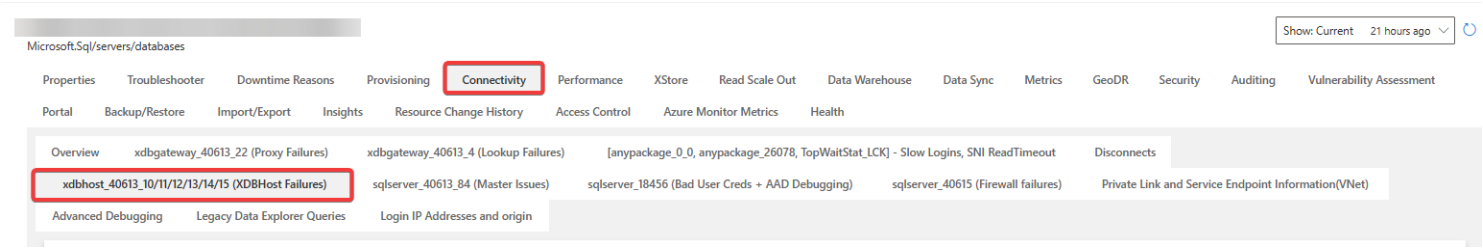
Using Azure Support Center

ASC Insight

We detect this issue in ASC and generate the below insight (Socket Duplication Issue) against the impacted resource.



In addition, you can create a Troubleshooter Report for the impacted resource and review Xdbhost related errors under the Connectivity section:



Using Kusto

Socket Duplication symptoms

Check MonXBHost to confirm if the duplication is failing
You will see an error such as [ERROR]Current socket Duplicate task stuck:{e628ed71e289}

```
let ServerName = "";
let Databasename = "";
MonAnalyticsDBSnapshot
| where logical_server_name == ServerName
| where logical_database_name == Databasename
| where ['state'] =~ "ready"
| summarize by logical_server_name, logical_database_name, sql_instance_name
| join kind = inner
(
    MonXdbhost
    | where text contains "***[ERROR]Current socket Duplicate task stuck:"
    | extend sql_instance_name = extract("task stuck: {([a-f0-9]+)", 1, text)
    | summarize MinTime = min(originalEventTimestamp), MaxTime = max(originalEventTimestamp), Count = count()
)
on sql_instance_name
| project logical_server_name, logical_database_name, sql_instance_name, MinTime, MaxTime, Count
```

Performance Issue

Query MonXdbhost and see if you're noticing the following for the resource : **Text:** ProcessSocketDupAsync ProgressStep::INIT error:1460, {AppName} task Queue”

```
MonXdbhost
| where event == "trace_event"   and NodeName contains "92" and ClusterName contains "tr17."
| where originalEventTimestamp >= datetime({Start_time}) and originalEventTimestamp <=datetime({End_time})
| where text contains "{AppName}"
| project originalEventTimestamp, text
```

Next, you can use this query to check if CPU or Memory load for this instance is high by comparing the load to its respective cap

```
let clusterFilter = "tr11.eastus2-a.worker.database.windows.net";
let nodeFilter = "DB.194";
let appName = "<appname>";
let startTime = datetime(2023-03-01 06:00:00);
let endTime = datetime(2023-03-05 07:00:00);
let MB_In_GB = 1024.0;
MonRgLoad
| where ClusterName == clusterFilter and NodeName contains nodeFilter
| where code_package_name == "Code"
| where application_name contains appName
| where TIMESTAMP > startTime and TIMESTAMP < endTime
| where ClusterName == clusterFilter
| where event == "instance_load"
| extend cpusGroup0 = tostring(parsejson(allocated_cpus).Group0)
| extend cpusGroup1 = tostring(parsejson(allocated_cpus).Group1)
| extend memory_load_gb = memory_load / MB_In_GB, memory_load_cap_gb = memory_load_cap / MB_In_GB
| project TIMESTAMP, application_name, NodeName, code_package_name, cpusGroup0, cpusGroup1, cpu_load,  cpu_cur
| order by TIMESTAMP desc
| render timechart
```

Also, check the wait stats

```
let clusterFilter = "{ClusterName}"; //"tr11.eastus2-a.worker.database.windows.net";
let nodeFilter = "{NodeName}";
let appName = "{AppName}";
let startTime = datetime({StartDate});
let endTime = datetime({EndDate});
MonDmCloudDatabaseWaitStats
| where ClusterName == clusterFilter and NodeName contains nodeFilter
| where AppName contains appName
| where TIMESTAMP > startTime and TIMESTAMP < endTime
| extend waiting_tasks = delta_waiting_tasks_count
| extend signal_time_ms_per_wait = delta_signal_wait_time_ms / waiting_tasks
| extend wait_time_ms_per_wait = delta_wait_time_ms / waiting_tasks
//| where wait_type == "MEMORY_ALLOCATION_EXT"
| project TIMESTAMP, wait_type,wait_time_ms_per_wait , waiting_tasks , signal_time_ms_per_wait, max_wait_time_
```

Mitigation

Scenario 1 - Socket duplication failure
If the customer is experiencing active connectivity issues and you've found that this is due to socket duplication failures/stuck; Open an IcM Incident with the Gateway team for further investigation.

Scenario 2 - Performance
Resource performance should be further analysed as you normally would on a typical performance case.

RCA Template

For on-going issues with socket duplication and you have an IcM, consult with the PG engineer regarding an RCA. If the issue self-mitigated and you intend to utilise the RCA templates, please consult with an xEE or TA prior to sharing with the customer

Summary of Impact
Between <StartTime> and <EndTime> connection attempts to database <> on server <> were failing.

Root Cause
The Azure connectivity architecture uses an internal TDS traffic load balancing component in the data path to your SQL instance. This component is running on the node hosting your SQL instance and its purpose is to forward incoming connections to the SQL instance.

The microservice component will accept the incoming connection and duplicate the socket into the destination SQL Instance. The process of duplicating the socket is done by maintaining a duplication queue per destination SQL process. However, the queue is serviced by one thread per destination process, which makes the duplication queue susceptible to backlog if the connection rate is high and the SQL process exhibits any slowness.

On the microservice, the next login is processed only after the current login has been acknowledged by the SQL engine. Inside the SQL Engine, logins are handled by a fixed number of cores (processors) and all login tasks are assigned to these cores. However, these cores don't exclusively handle login tasks and can handle other workloads within the SQL engine as well. If one or more of the cores allocated to login processing in the SQL Engine have long running tasks scheduled on them, the login tasks can end up waiting to be processed. This delays the acknowledgement back to the microservice and would in turn lead to queuing of logins on the microservice because of head-of-line blocking.

We are aware of this problem and have made improvements to the following:

1. Improve the login queue management of the microservice to make it more scalable and resilient during stress conditions. This work changes the login transfer between the microservice and the SQL engine to make it asynchronous. This work should minimize head-of-line blocking issues.
2. Improve the login task scheduling in the SQL engine. This will help minimize the dependency of login tasks on the completion of other tasks.

Additional RCA Template for separate scenario - consult with xEE or TA before sharing with the customer
Summary of Impact - Between <Starttime> and <EndTime> Database <Database Name> on Server <Server name> was was unavailable because the gateway process was incorrectly directing logins to a node that was not hosting your SQL Server process.

Root Cause - This issue is quite rare, and it can occur after the SQL Server process fails over to a new node, which happens as part of an update SLO operation. There is an automation in-place to automatically detect and mitigate this instance by restarting the gateway process so that it can correctly redirect logins if the process fails to correctly update the redirection. However, in rare scenarios automation may fail to handle the issue and remain database longer until we manually failover. We will continue to investigate how to improve our automation to identify and properly mitigate these extremely rare events. In this case, it appears that our automatic mitigations were not sufficient to mitigate the issue in a timely fashion, and we will continue to investigate and evaluate appropriate improvements to help prevent this issue in the future. We apologize sincerely for the inconvenience that this issue has caused on your business.

Classification

Root Cause: Azure SQL DB v2\Connectivity\ for scenario 1 Root Cause: Azure SQL DB v2\Performance\resourcelimits for scenario 2

How good have you found this content?