Tivoli Netcool Supports Guide to Production Triggers

by

Jim Hutchinson

Document release: 2.0

Table of Contents

1 Introduction

1.1 Overview

This document was written to highlight the types of custom triggers that are required to support a production object server. There are two main groups of triggers discussed;

- Object Server Connection Management
- Temporal Trigger Management
- Monitoring Dynamic Tables
- Historical Gateway Resynchronisation

The triggers provided in the SQL scripts are examples triggers. It is expected that the Netcool/OMNIbus systems administrator will customise the triggers to meet the systems specific requirements.

1.2 Concepts

1.2.1 Naming

Trigger naming is important and if planned will improve system administration. Trigger names should contain unique prefixes, so that custom triggers are easy to distinguish from other triggers. It is best to create custom SQL files for use with nco_dbinit, with prefixes added to the standard triggers and procedures, to aid identification. Customisation is therefore made apparent which is useful when upgrading or when triggers/procedures need to be migrated between systems using nco_confpack. If unique prefix identifiers are used in the trigger/procedure names, sorting triggers in nco_config [Administrator] using their name allows them to be identified as customised triggers, or belonging to a specific trigger group without trigger group sorting.

e.g.
v721_generic_clear – Netcool/OMNIbus v7.2.1 Generic Clear trigger
v721_primary – Primary Object Server only trigger group
my_generic_clear – Custom Generic Clear trigger
my_triggers – Customised trigger group
test_event_suppression – Test event suppression trigger
test_triggers – Test trigger group

1.2.2 New Tables

New tables should be held within a new database, so that they are easily distinguishable from the default tables.

Any new table names should be descriptive.

All new tables that have data insert statements used against them need to have a table specific deduplication trigger.

1.2.3 SQL statements

Though comments can be made in triggers, these need to be terse as there is a limitation on the number of characters stored within triggers. Therefore keeping comments to a minimum whilst making the SQL statements readable is the best method to ensure triggers are maintainable. There is a comment/description field that can be used to supplement trigger in-line annotations.

The SQL code should be concise. The ‘where’ clause should be used to filter data, such that the data being acted upon is the smallest possible sub-set of the data within the object server.

The use of local variables should be kept to a minimum. Variables require initialisation at the start of the SQL statements, as well as when the data is being specifically set.

For temporal triggers, it is important to choose timing values that are staggered, using odd and prime numbers around the precise periodicity will achieve better performance than all the triggers running at the same periodicity or periodicities that are factors of each other.

Triggers must never be recursive and should not act upon large portions of the object server’s data.

Selected data should be minimised as the temporary space used to hold this data is limited and is not presently expandable. This is because the object server is designed to be light weight. Processing large text fields takes a lot more time than integer key fields, such as Serial. Using ‘via Identifier’ is useful, although only performance effective where Identifier is a light weight value, rather than a large string, which uses up temporary storage. The object server reports ‘No Space’ when the temporary storage limit is reached, which results in object server failure.

2 Connection Management

2.1 Object Server Connection Overview

The Object Server is limited to the number of connections it allows through the operating system and property MaxConnections. There is also a hard coded default of 512 connections allowed, for object servers before Netcool/OMNIbus 7.3.0. For UNIX the number of open files should be set to at least twice the value of the MaxConnections property (e.g. 1024). The open files size is best set as powers of two;

e.g. 1024, 2048, 4096 etc.

and defined in the 'nco' start-up script;

e.g.
```
vi /etc/init.d/nco
#! /bin/sh
ulimit -n 2048
:wq
```

2.2 Object Server Connection Issues

There are various reasons why connections can become a problem for an object server. These range from security, through to an excessive usage of the systems resources. Typically desktop clients may become a problem in a system using the multi-tier Architecture, due to the limitation of the number of allowed connections. This means that there is a requirement to manage which clients are allowed to connect to a production object server.

Additionally, in a mixed end-user environment, specific object servers may be allocated to specific users, resulting in the need to ensure only machines that are allowed to connect to an object server can connect.

The two main types of trigger used to handle connections are signal and temporal. The signal trigger can disconnect a client immediately, while a temporal trigger is periodic, allowing the client connection to be closed within a given maximum time period. The trigger type is used, depends upon the security requirements of the system.

2.3 Example Triggers

The example connection management triggers use the trigger group `nc_production_triggers`. Along with the tables `nc_triggers.allowed_applications` and `nc_triggers.allowed_hosts`. These tables are used to hold the allowed ‘AppName’ and ‘HostName’ fields, whose column values are found within the catalog.connections table, while the allowed connection is connected to the object server. The field strings are defined by the connecting client’s and may vary from client to client. Therefore it is important to ensure that the correct syntax is used (FQDN, hostname, etc.).

2.3.1 nc_rogue_clients

The `nc_rogue_clients` trigger uses the statistics gathered by the object server to deem that a client should be disconnected. Using the data available in the `catalog.profiles` table, the trigger disconnects clients whose `PeriodSQLTime` exceeds 20 cycles. This is effectively 33% of the cpu resources available to the object server. Additional clauses can be added to allow other clients such as ‘GATEWAY’s to use more resources as required.

2.3.2 nc_drop_unknown_connections

The `nc_drop_unknown_connections` trigger is a temporal trigger that disconnects clients based in tables that define which client connections are allowed to connect to the object server. Any client connection that is not defined within these tables is disconnected.

2.3.3 nc_drop_all_unknown_connections

The `nc_drop_all_unknown_connections` is a signal trigger that checks to see if the client attempting to connect to the object server is allowed. It uses the same tables as the `nc_drop_unknown_connections` trigger, and is a more aggressive replacement for this temporal trigger. In the event of the allowed tables becoming corrupted, the object server would need to be restarted with all the triggers inactive, so that the system could be repaired.

3 Temporal Trigger Management

3.1 Temporal Trigger Monitoring

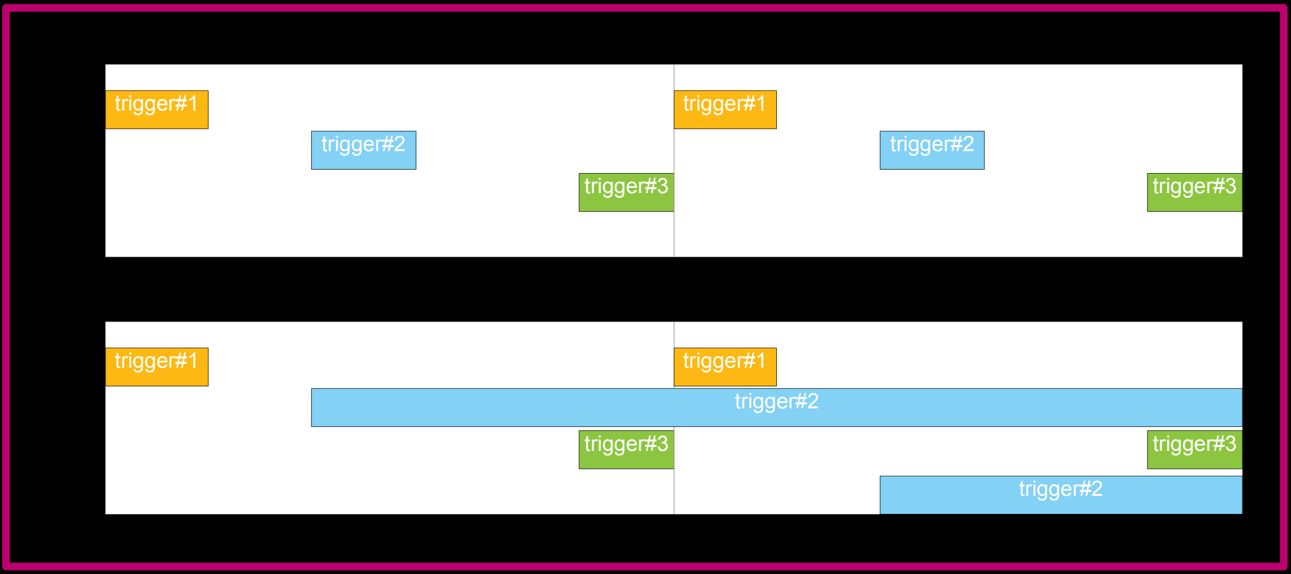
Temporal triggers can overload a system when an unexpected number of events are seen in the object server, or in the table being acted upon. There are many ways to ensure that this situation is avoided and dealt with.

One way is to prevent the temporal trigger from being run if it is already running;

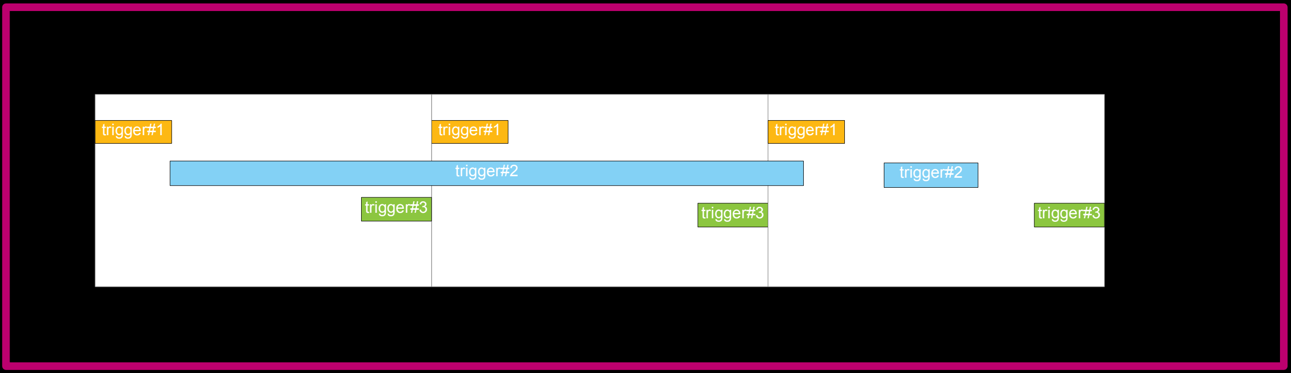
Through the use of a wrapper trigger, it is possible to load a system to maximum capacity without causing it to fail completely. Although, the system is processing at its maximum load, it should remain usable enough to allow the situation to be monitored and administered until the temporal triggers revert to their normal loads.

3.1.1 Design Overview

Consider three temporal triggers, that are normally timed to run outside each others runtimes. Should one trigger exceed its normal runtime, it affects other triggers, and potentially its own next run. When this occurs the system reaches a state whereby it may become unrecoverable, or unusable without deactivating the affected trigger[s].



If the triggers are monitored, and prevented from being run if they have yet to complete their task, the system is kept within its limitations.



This temporal trigger monitoring model can be extended to deal with triggers as groups, though this is not dealt with in the example SQL provided within this text. Such configuration would check to see if a group of triggers had completed their task, allowing dependent triggers to be managed.

3.1.2 Temporal Trigger Monitor Example Design

The example provided uses a table called `nc_triggers.monitor` to store the list of temporal triggers using the wrapper trigger code. This table is populated at start-up using another trigger which the system administrator must maintain called `nc_trigger_system_startup`. In the example the table is populate with the name of the template wrapper trigger `nc_generic_clear`.

The table is populated using insert statements;
e.g.

```
insert into nc_triggers.monitor values ( 'nc_generic_clear', false );
```

The wrapper trigger `nc_generic_clear` then checks the `nc_triggers.monitor` table to see if the trigger is already active. With active being defined through the Boolean field Active. If the trigger is not active, the trigger sets the Active flag to ‘true’ and continues to process the rest of the trigger code, until this is completed. After which point the trigger sets the Active field back to ‘false’. Therefore allowing the trigger to run on the next invocation. If the trigger is still defined as Active when the trigger is run next, then the trigger is exited.

4 Monitoring Dynamic Tables

Within the default object server there are four main dynamic tables;

- alerts.status
- alerts.journal
- alerts.details
- master.stats

These tables are managed using ‘clean’ triggers that can be disabled. If this occurs there is a risk that the volume of data will result in an object server outage. It is possible to be alerted of this situation using a set of monitor triggers and a row count table that defines the maximum row count for each table.

The default settings for the tables are;

- alerts.status < 50k
- alerts.journal < 100k
- alerts.details = 0
- master.stats < 1k

The default action is to insert a critical event into alerts.status. Other actions can be added to the table specific trigger as required.

4.1 Adding a custom table

If custom tables need to be monitored;

1. Add a definition of the custom table to the table nc_rowcount_triggers.dynamic_tables;

```
TableName        ‘custom.mytable’
TimeStamp        1289831823
MaxRowCount      100000
RowCount         0
```

2. Create [copy] a new temporal trigger to monitor the table with the syntax nc_rowcount_<tablename> in the nc_rowcount_triggers trigger group.

3. Modify the *nc_rowcount_mytable* trigger as required

5 Historical Gateway Resynchronisation

Historical gateways can cause a sudden increase in load on the object server during Synchronisation on start-up. In addition there is a risk that events are lost whilst the historical gateway is down, if a historical gateway filter is used. In order to workaround these issues it is possible to set the gateways filter flag to a value that prevents any synchronisation on start-up. Once the historical gateway is running, blocks of events are forwarded to the historical database through the setting of the gateways filter flag.

6 Trigger Installation

6.1 Overview

The triggers discussed in this document are provided as working examples.

System management triggers [triggers_r#]

These can be applied to a default object server created using nco_dbinit and applied using nco_config or nco_sql. It includes a set of SQL files and a README.txt file describing how to apply and use the files provided.

6.2 Systems management triggers

- Determine the hosts you require to access the object server;

  e.g. Connect the require clients to the object server and use users to determine the correct syntax

- Update the rogue_client_tools.sql script to reflect these requirements, before applying the triggers;

```
unzip triggers_r1_0.zip
cd triggers_r1_0
vi rogue_client_tools.sql
-- standard hosts to allow
insert into nc_triggers.allowed_hosts values ( 'a-host', true );
insert into nc_triggers.allowed_hosts values ( 'a-host.domainname.com', true );
insert into nc_triggers.allowed_hosts values ( 'b-host', true );
insert into nc_triggers.allowed_hosts values ( 'b-host.domainname.com', true );
go
-- standard applications to allow
insert into nc_triggers.allowed_applications values ( 'Administrator', true );
insert into nc_triggers.allowed_applications values ( 'isql', true );
insert into nc_triggers.allowed_applications values ( 'GATEWAY', true );
go
:wq


cat temporal_trigger_monitor.sql | \
        nco_sql -server NCOMS -user root -password ‘’
cat rogue_client_tools.sql | \
        nco_sql -server NCOMS -user root -password ‘’
```

- The trigger groups are installed disabled, as is the nc_drop_unknown_connections trigger. These need to be enabled after first visually checking the installation using nco_config, as required

- To fallback, start the object server without the triggers running use the –autoenabled option, and disable the triggers or edit the control tables as required

```
nco_objserv -name NCOMS -autoenabled FALSE
```

- To remove the examples use the remove_triggers.sql script

```
cat rogue_remove_triggers.sql | \
        nco_sql -server NCOMS -user root -password ‘’
```