ForgeRock Core Token Service Affinity Based Load Balancing

Written by Darinder S. Shokar

Problem Description

The ForgeRock Core Token Service (CTS) is a specific ForgeRock Directory Services repository used to store and persist ForgeRock Access Management (AM) SSO, OAuth2/OIDC and SAML tokens. In terms of architecture, typical primary/secondary aka active/standby or all active 1:1 AM topologies are adopted for AM to CTS connections (described in more detail below). Both of these approach have inherent limitations, namely:

1. Active/standby topologies do not maximise available hardware (only the primary node services requests) and horizontal scaling of the CTS pool of servers is not possible.

2. Under high load active/standby topologies require expensive vertically scaled instances.

3. 1:1 all active CTS topologies require stickiness for all session requests to the server which created the token or replication delay becomes a functional issue.

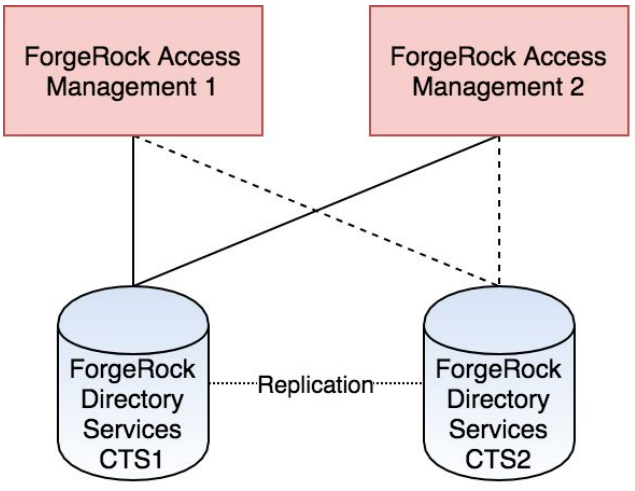
Affinity based load balancing for CTS elegantly resolves these problems.

Traditional AM - CTS Topology

Assuming CTS connections strings are used; the recommended approach to avoid the need for a load balancer and allow AM to manage its own connections, typically one of two AM - CTS topologies are used - Primary/Secondary (aka Active/Standby) or 1:1 all active. These are described below.

Mode 1 - Primary/Secondary

All AMs in the pool communicate with a single CTS server. If this server fails all connections failover to the secondary CTS server and so on. The following topology depicts this architecture.



All requests from AM1 and AM2 to CTS1. CTS2 in standby in case of primary failure

In this mode replication delay is not an issue as all traffic hits the primary and as long as replication happens successfully over time; functionality is unaffected.

A connection string for this mode would look like this:
`cts1.example.com:1636,cts2.example.com:1636`

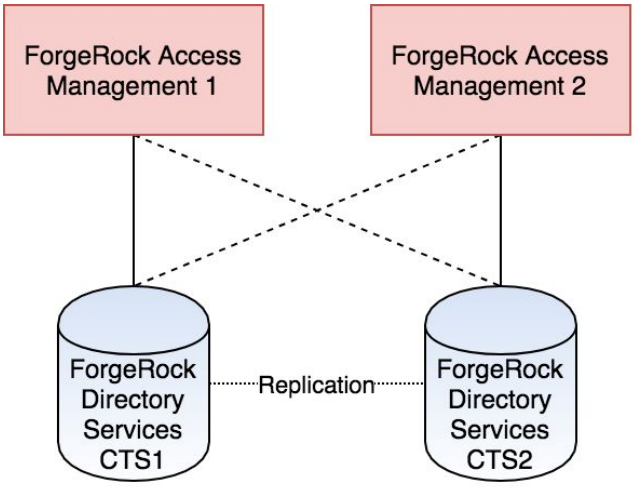
Issues:

- In an high load environment, CTS needs to be vertically scaled to ensure a single server can handle all requests. This is extremely costly for both on-prem and cloud based deployments.

- Does not make efficient use of the available hardware; only one server is handling traffic the others are sat idle awaiting primary failure.

Mode 2 - All Active CTS with 1:1 Mapping between AM and CTS nodes

Each AM communicates with its own CTS server, if this node fails it connects to its failover CTS. For example AM1 connects to CTS1 as its primary with CTS2 as it secondary, AM2 connects to CTS2 as its primary and CTS1 as its secondary. The following topology depicts this architecture.



All requests from AM1 to CTS1 and AM2 to CTS1 . AM1 failover to CTS2 and AM2 to CTS1 in case of primary failure

This mode allow an all active CTS farm to maximise hardware usage. However, this approach requires end to end stickiness to the AM node which created a given token to avoid functional issues as a result of replication delay.

The connection string would look like the following where the serverID for AM1=01 and AM2=02.
`cts1.example.com:1636|01,cts2.example.com:1636|01,cts2.example.com:1636|02, cts1.example.com:1636|02`

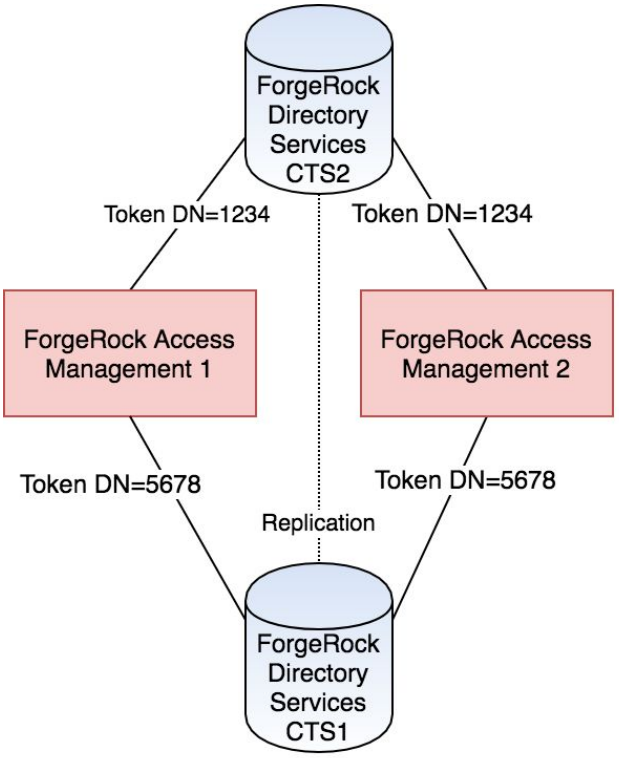
Issues:

- An all active architecture like this is highly susceptible to replication delay if stickiness to the AM node which created a given session cannot be guaranteed. DJ replication is based on a “loose” algorithm where all CTS nodes will fully converge over time and not instantly. This means under high load a token create may hit AM1 and CTS1, however AM2 which is connected to CTS2 may be hit for the token validate request. If replication has not happened fast enough the request would error.

What is CTS Affinity Based Load Balancing?

Affinity based load balancing for CTS addresses the key issues with the more traditional approaches; namely, unlimited horizontal scaling of the CTS pool using smaller, cheaper instances and eliminating functional issues as a result of replication delay.

It does this by making use of the new affinity based load balancing algorithm built into the DJ SDK deployed with AM. For each and every inbound token DN, the SDK creates a hash and allocates the result to a specific CTS DJ instance in the CTS connection string. All AM servers in the pool compute the same hash and thus send the request to the same CTS instance. The next request for another token DN is hashed and may be sent to another CTS instance in the pool and so on. The end result of this is, for each and every token a create occurs on a specific CTS instance (the token origin server) and from that point on all read, update and delete operations will be sent to this same origin server from any AM node. The following topology depicts this architecture:



All requests from AM1 and AM2 to CTS1 for a specific token DN . All requests from AM1 and AM2 to CTS2 for a different token DN

The DJ SDK also makes sure token creates are spread close to evenly across all CTS nodes in the pool to ensure one CTS server is not overloaded with requests, while the others remain idle. Finally, the DJ SDK is instance aware; if the origin server goes down, the request goes to another in the pool and remains sticky to that CTS instance from them on. Assuming replication has happened there will be no functional impact. When the original CTS server comes back online, requests fails back to that server for any requests where it is the origin servers.

The connection string for affinity simply lists all CTS instances as a comma separate list:
`cts1.example.com:1636,cts2.example.com:1636`

It is imperative that all AM servers share the same connection string, affinity will fail if each AM server changes the ordering of the connection string.

What Are the Advantages of CTS Affinity Based Load Balancing?

The primary advantages of CTS Affinity based Load balancing over the traditional topologies are:

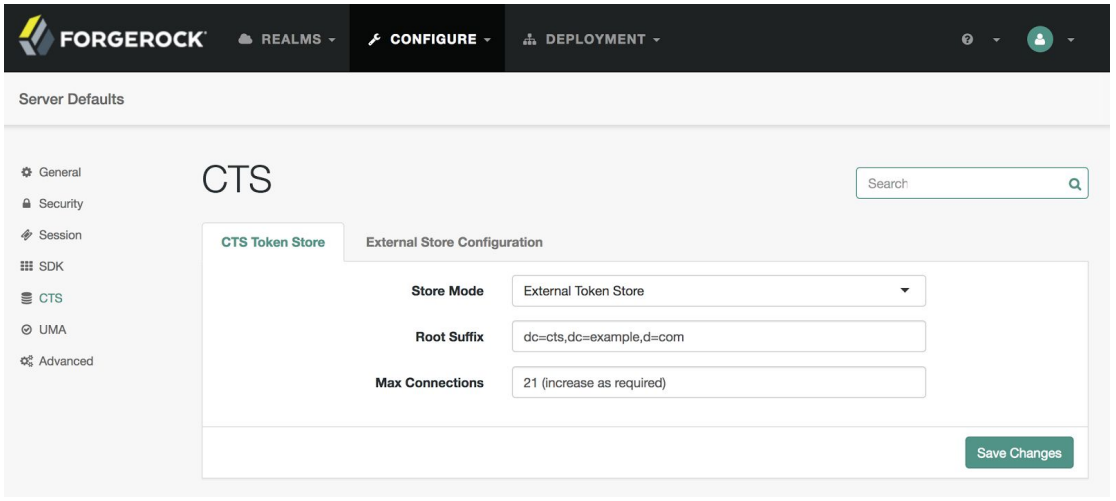
1. Allows horizontal scaling of the CTS server farm, this is not possible with the traditional topologies.
2. Maximises available resources by introducing an all active CTS pool of servers.
3. The all active CTS architecture allows smaller, cheaper nodes for handling CTS traffic, rather than expensive vertically scaled nodes.
4. Combats replication delay between CTS nodes.
5. Removes the need for end-to-end stickiness to AM for session activities (note still required for AuthN).

How do you configure CTS Affinity Based Load Balancing?

CTS Affinity based load balancing can be configured using amSter, ssoadm and the GUI. The GUI approach is described below.

1. Build an external CTS repository as per the following guides: https://backstage.forgerock.com/docs/am/5.1/install-guide/#cts-deployment-steps and https://backstage.forgerock.com/knowledge/kb/article/a46985800
2. As the connection string for CTS Affinity needs to be **exactly** the same across all AM nodes, it is recommended to make the necessary CTS changes at the “Server Default” level rather than at an instance level to ensure all AM nodes inherit the settings from these defaults. So from the AM Administration GUI goto: Configure -> Server Defaults -> CTS.
3. Change the Store mode to “External Token Store”, set the Root Suffix to that configured in step 1 and set Max Connections as appropriate.

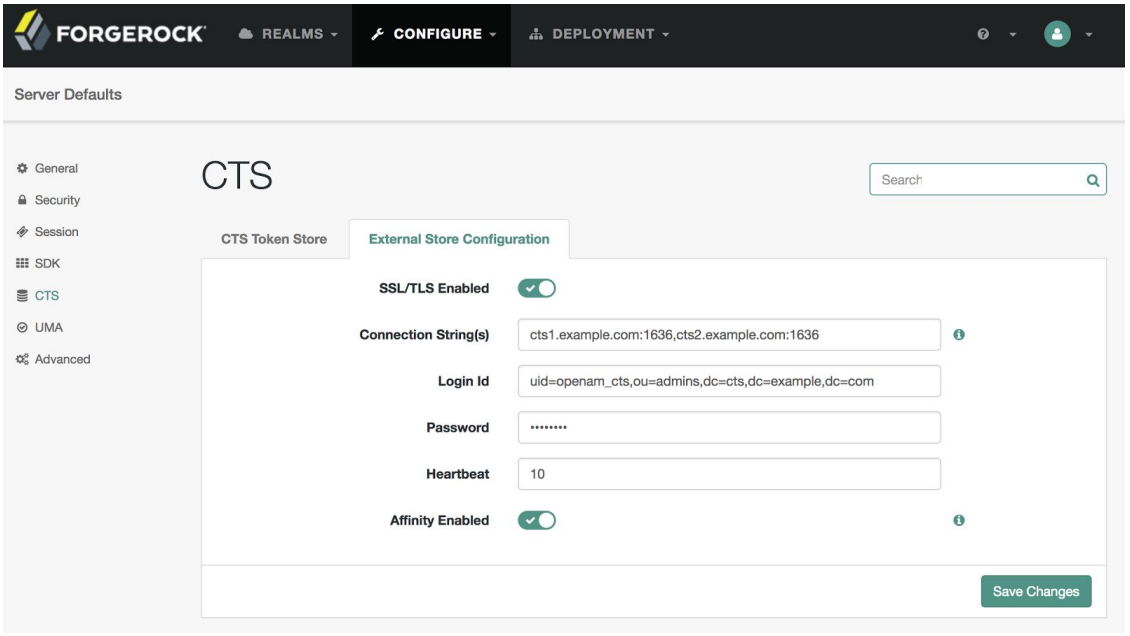
   The Max connections is shared as follows; 1 connection for CTS cleanup Reaper and the rest shared equally between the nodes specified in the connection string. For example if there are 2 CTS nodes and the connection string is set to 21; 10 connections will be allocated to CTS1; 10 to CTS2 and 1 reserved for the reaper. Load testing will determine optimal value - increase as appropriate.



4. Click Save Changes

5. Select the External Store Configuration tab. Enable SSL/TLS if CTS is LDAPS enabled, define the connection string with a comma separated list in the format <server FQDN>:<port>, set the Login ID and password as appropriate. An example connection string is:

   ```
   cts1.example.com:1636,cts2.example.com:1636
   ```



6. Click Save Changes.

7. Restart the AM node(s). Configuration complete :)

   Note if there are problems be sure to check out AM's debug logs - usually it something like the bind credentials to connect to CTS are wrong.