

USS2

Overview

USS2 is a brand new design and is in many ways fundamentally different from the old USS, although many basic concepts are shared. It is important to understand that USS2 is *not* built on the old USS but is designed ground up as a more generic engine.

Some of the key differences from the old USS are:

- USS2 is a replication and persistence handling framework that allows "pluggable" state machines. There is currently a legacy RADIUS state machine and a cache handling state machine available, where the RADIUS state machine implements the same state handling as the old USS. However, in future releases there may be new types such as Diameter, IS835C, TACACS+ etc, all with their semantics.
- USS2 can handle any number of *instances* of state machines of the same or different types, each with its own database and set of replication hosts.
- USS2 replicates in active-active mode and all hosts in a replication domain can service requests at any time. This means that there is no "role" switch over time at a failure situation, although there is a lag time for entries to replicate (typically in the sub-second range under reasonable load conditions).
- USS2 is using custom serialization and replication mechanisms as opposed to RMI in the old USS.
- USS2 is designed to be enhanced with resource brokering for geographic redundancy.

With the above in mind, there are a number of concepts that are generic for USS2 and there are others that are specific to the state machine.

Terminology

The state handling modules in USS2 are called "models". A specific type of model, such as RADIUS, is called a "model type". Configuring and naming a model makes it a "model instance".

A given release of the server may provide some number of model types, where RADIUS and "general" (provides a generic replicated cache) are the ones currently available. An installed server may in turn have several *instances* of the RADIUS type. So, as an example, could a given installation have a RADIUS model called orange that replicates to A, B and C and another RADIUS model called plum that replicates to B and D.

The servers involved in replication are called "nodes".

Models

From an implementation/conceptual standpoint a model type consists of three major components:

- A Plugin. The Plugin is a fairly "thin" component that serves as an entry point from policy flow into the model's state machine.
- A state machine. This is the heart of the model type and defines and drives what events and states are available and how event changes affect the data maintained in the database associated with the model instance.
- An entry definition. An entry makes up the data "entity" that the state machine works on. Entries are mostly the same across different state machines but typically have some specific fields and methods to deal with the state machine they're implemented for.

Every model instance has its own, private database of entries. The database is generic in nature although the primary key and the indexes in the database are configurable per model instance.

Flow of control

The Plugin gets invoked (called) from policy flow and uses the value of its "model property info" to find the correct model instance. The model instance is then called with the current work-item as a parameter which feeds into the state machine that in turn uses the data in the work item along with its configuration to alter, create or delete an entry based on it.

As the state changing code in the state machine exits, the common USS2 engine determines whether or not the entry needs to be replicated, marked for deletion and/or persisted to disk.

Finally, control is returned back to the Plugin with a disposition that conveys the result of the operation.

Configuration

Model configuration is split up in two groups of parameters; Parameters defined by the USS2 framework and parameters specific to the type of model. Configuration would typically be performed through the SMT, but the actual data is stored in a stanza file. The USS2 defined parameters are valid in every model and are as follows:

Name	Type	Description
MODEL-Name	String	The name of the model instance.
MODEL-Type	Enumeration	The type of instance, currently "GeneralState" and "RadiusState" is available.
MODEL-Nodes	Enumerations	List of node names as they appear in the nodes configuration.
MODEL-ReplicationMap	Map	The replication map code that determines how replicated entries are merged.
MODEL-Index	Strings	Entry field names to index the database on.
MODEL-Persist	Enumeration	One of "Never", "Shutdown" or "RealTime".
MODEL-Obliterate	Duration	The maximum time entries are kept in the "purged" state before they are forcefully obliterated.

Every model instance must have a name which is used to refer to the model from policy flow through the ModelPropertyInfo.
The type of model ties the instance to a specific type of state machine.

Nodes

The Nodes list is a list of names that refer to node names in the separate node configuration (see below). The relation between model instances and nodes is N to N, i.e. the same node may appear under multiple model instances and a model instance may be served by any number of nodes.

The replication engine on every server attempts to establish connections to all configured nodes in a symmetric fashion, meaning that no server is designated to just listen or just connect. A server needs to know what local address and port to listen for connections on in addition to knowing the remote servers. To keep configuration consistent across all servers an identical node configuration is kept on every server and the local address is determined as the node whose name is identical (case insensitive) to the configured Diameter Origin-Host name.

The Origin-Host property is in turn determined from the FQDN of the lowest numbered unicast IP interface but can be overridden by a server property.

The Replication Map

The replication map is a specialized map that provides the user with an ability to determine how the data in entries is joined when replicated entries are received for already existing entries.

Entries typically maintain two internal name value maps, the "meta" map and the "data" map, although it is conceivable that some basic state machine would only need the "meta" map. The meta map, or from a conceptual standpoint, the *meta namespace* contains "housekeeping" data necessary for the specific state machine that the entry is for. So, for example, do the radius entries contain META_RadiusState, META_NasKey and META_SessionId among other items, all necessary for the state machine to do its work.

The user name space on the other hand, is free form data that the state machine doesn't pay any attention to solely intended for arbitrary user variables.

Give the above, the syntax of the replication map name is defined as follows:

```
<name-space>.<condition>.<variable-path>
```

Where the *name-space* is one of "meta" or "data", *condition* is one of the conditions according to the table below and *variable-path* is the normal path to a variable, such as "user-name".

Condition	Meaning
local	Data from the local entry.
remote	Data from the remote entry.
newer	Data from the newest entry.
older	Data from the oldest entry.
localIfNewer	Data from the local entry if the local entry is newer than the remote entry.
localIfOlder	Data from the local entry if the local entry is older than the remote entry.
remoteIfNewer	Data from the remote entry if the remote entry is newer than the local entry.
remoteIfOlder	Data from the remote entry if the remote entry is older than the local entry.

The default map statement in USS2 is:

```
#{meta.local.*} := #{meta.remoteIfNewer.*};  
#{data.local.*} := #{data.remoteIfNewer.*};
```

In effect causing both name spaces to be assigned from the remote (replicated) entry if the remote entry is newer, otherwise left untouched.

The entry deletion related property "Obliterate"

USS2 replication is based on tagging of replication entries with a "bucket number" implemented as the "seconds since the epoch" UNIX style 32-bit integer time-stamp value. Remote nodes acknowledge received replication traffic as counts of entries per bucket number. An acknowledge message may thus in human terms be expressed as something like: "I received 3126 entries tagged 11:27:16, 918 entries tagged 11:27:17 and 1933 entries tagged 11:17:19".

The replicating node keeps track of outstanding buckets and takes them off the list of outstanding replication as their counts reach zero.

Entry deletion is dependent on this mechanism and when a local entry is due to be deleted, e.g. from an inactive time out expiring it is not immediately deleted if the model has replication turned on. Instead the entry is tagged as "purged" and kept on a separate list of entries who are pending for deletion. As acknowledge messages are received from replication nodes with time stamps that exceed the time stamp in the purged entry, the entry can be terminally deleted "obliterated". If, for some reason, acknowledges are not received from one or more nodes the list of purged entries would potentially grow indefinitely leading to memory exhaustion on the server.

The Obliterate property defines a time-out after which unacknowledged entries older than the time-out will be taken off the purge list and obliterated, i.e. removed from the database.

Miscellaneous properties

The strings in the Index property are used as names of fields, normally from the data space, that index tables are built from. This is analogous to the old USS, i.e. if an index of "user-name" is defined data in the "user-name" field is indexed.

The persist property determines when entries are persisted to disk. This property can take on the value "Never" meaning never persist, "Shutdown" meaning to persist when the server is shutdown only (same as old USS) or "RealTime" meaning that entries are persisted every time they have been modified. The real time persistence setting will affect overall performance, potentially significantly.

RADIUS specific properties

This document deals with the USS2 in general, but in order to add some context, RADIUS specific information is documented as well.

The properties specific to the RADIUS model type are:

Name	Type
RadiusState-AuthCounter	Strings
RadiusState-AcctCounter	Strings
RadiusState-AcceptedTimeout	Duration
RadiusState-ActiveTimeout	Duration
RadiusState-InactiveTimeout	Duration

The Auth-Counter and Acct-Counter properties define names of fields in the data name-space that are used as counters. The difference between an Auth-Counter and an Acct-Counter is that the Auth-Counter is incremented at RADIUS auth time whereas the Acct-Counter is incremented at accounting start time. The counters are used to impose limits on resources the same way as in the old USS.

The three last properties establish entry timeouts from the auth event (accepted), the accounting start event (active) and the accounting stop event (inactive) before an entry is deleted, once again analogous to the old USS.

Nodes

A node configuration has two required properties; the name of the node, typically an FQDN, and the network address, given as either an FQDN or an IP address, followed by a port number.

The name of the node is the name referred to from the models configuration and is also the string used to find the local node by comparing it to the Diameter Origin-Host (see Replication basics below).

The address is given in the usual "address:port" representation.

In addition to the two required properties, there are a number of properties that allow fine-tuning of replication for the node. These parameters are picked up from the server properties defaults or specific explicitly given values, but can be overridden on a per-node basis.

Replication basics

When servers come up, the node configuration is read and the nodes defined are examined. The name of each node is compared case insensitive to the Diameter Origin-Host (see end of this section) value, and if the name matches, a local listener is started on the given address and port. If the name does not match, the node is considered remote and a connection state machine as well as a pool of replication threads are started for the node.

The connection state machine's purpose is to attempt to establish a connection to the remote node. Since all nodes try to connect to each other as they come up the state machine has to resolve the not too unlikely condition that two nodes connect to each other at the same time and agree on which of the two connections to keep and which to discard.

The four states the state machine has defined are:

- Attempting: The initial state trying to connect to a remote.
- Proposing: Connection has succeeded and the local node has been identified to the remote but not accepted.
- Expecting: A connection from a remote has been accepted but not acknowledged.
- Established: A connection is up and active. The established state is published as either "EstablishedA" or "EstablishedC" in the command

output depending on whether the connection was established by the local node connecting to the remote (EstablishedC) or the local node accepting a connection from the remote (EstablishedA).

Once a connection is established, the nodes will start to exchange "heartbeats" between each other. The purpose of the heartbeat is twofold: (1) It serves as a way to detect connectivity problems from missing heartbeats and (2) to pass replication acknowledges back to the sender.

The basic mechanism used to acknowledge replication is based on a system of "buckets". The *replicating* server tags all outbound entries with a bucket number. Several entries typically share a bucket number and the replicating server keeps track of how many are sent with the same number in a "bucket". The *receiving* server also counts entries per bucket number and piggy backs a list of how many entries it has encountered for each bucket on the heartbeat message if the count is greater than zero.

When the acknowledge data comes back to the sender, the counts are subtracted from their respective buckets and as buckets are counted down to zero they are discarded.

Using second resolution time stamps as bucket numbers provides a monotonically increasing number space with fairly ideal "clumping" of entries per bucket. The base for the time stamps has been chosen to be "standard" UNIX time, i.e. the number of seconds since 1970.01.01 00:00:00 UTC which also makes them easy to log in human readable time.

Since the bucket numbers are monotonically increasing, any data whose timestamp/bucket-number precedes the oldest non-zero bucket has been acknowledged by the remote node.

This acknowledgement scheme is used to "obliterate" entries (see Obliterate above) as well as to determine when and what to "reconcile" when needed.

Reconciliation

Reconciliation is the act of replicating everything from a given point in time over to another node, typically after determining missing replication. The two reasons for reconciliation are; (1) The link state changes to non-established or (2) non-zero buckets are not getting acknowledged for an extended time.

The heartbeat message carries along a boot-time field that is the time the sending node last thought it had replication up to date.

Every time the link state enters established state each node looks at the boot-time advertised by the remote node. The minimum of the remote boot-time and local nodes notion of when the remote node was last seen becomes the *reconciliation-point*. The entire database of entries is then traversed and all entries whose modification time are equal to or exceed the reconciliation point are replicated to the remote node.

If buckets are getting too old a slightly different approach is taken; when the age limit is reached, replication is turned off all together and reconciliation is made pending while waiting for the remote node to stop acknowledging entries to make sure replication overhead from the local node is idle. At that point reconciliation is started with the reconciliation point set to the oldest outstanding bucket.

The Diameter Origin-Host property

Since the Origin-Host name is used to determine the local USS2 node a few words will be said about it. The value *can* be set through the server property "Origin-Host", but is otherwise picked up automatically as the FQDN of the lowest numbered unicast IP interface in the system. To avoid dependencies on DNS and hardware configuration it is recommended that the server property is set to an appropriate value.

"Advanced" node properties

Property	Description
Uss2-Node-Timeout	The amount of time before the outbound replication queue is drained after the node connection state has dropped out of established state.
Uss2-Heartbeat-Time	How often the heartbeat message is sent.
Uss2-Heartbeat-Skip	How many heartbeats may be missing before the connection id determined to be non-established.
Uss2-Bucket-Load-Factor	How many heartbeat intervals worth of age of the oldest bucket is accepted before reconciliation is initiated.
Uss2-Idle-Ack-Rate	When the number of acknowledges per heartbeat drops below this limit (the "idle" limit) after a reconciliation has been initiated the reconciliation is started.
Uss2-Replicator-Pool-Size	The size of the thread pool used to process outbound replication traffic.

Note that all of the above node properties are backed by server properties that all have reasonable defaults and will only need to be adjusted in environments with special requirements.

USS2 Engine configuration

Since almost all configuration is done per model and some in the node very little information is global in USS2. In fact, there is only one server property that is only for the engine besides the node properties and that is the

`Uss2-Replicator-Pool-Size`. `Uss2-Replicator-Pool-Size` configures the number of threads in the *inbound* replication pool. The optimal number for this property and for the `Uss2-Replicator-Pool-Size` node property is obviously highly dependent on the type of machine the server is run on.

As always with a thread pool size, there are a number of different parameters that determines a reasonable number of threads. Setting the number much higher than the number of CPU's (or cores) is usually a waste. It should be realized, however, that threads that spend some amount of their execution time waiting for resources (typically I/O) may overlap with other threads even on a single CPU.

Giving exact recommendations for the number is beyond the scope of this document and in many instances a good number has to be found by experimentation given a platform and environment. During in-house testing with replication between two Sun T1000 servers connected by a GigE network it was found that any number beyond four didn't show any signs of improved performance.