List Filters in ConfD

tail-f a Cisco company

Table of Contents

1. Introduction

In order to minimize the amount of data that needs to cross the network, most northbound management interfaces include support to receive a query when data is requested. The device will use the query to filter its data and will only send response data that matches the query. Not only is network traffic reduced, but processing time at the controller or orchestrator is also reduced by working with a smaller data set.

Prior to ConfD 7.1, query filters were only applied in the ConfD core engine. This means that the ConfD core engine would have to retrieve the full data set from a Data Provider prior to filtering. It was recognized that for very large data sets, this is inefficient and can introduce latency in the response. Also, in some systems, it can be an expensive operation for the Data Provider to gather all the data.

This problem was addressed in ConfD 7.1 with the introduction of the "List Filters" feature. This feature is part of the Data Provider API (DPAPI) and enables ConfD to pass query criteria to a Data Provider and have the backend code filter for matching data in a YANG list prior to sending it to the ConfD core engine.

This application note provides an introduction to the List Filters feature; what it is, how it works, and how to implement it in your Data Provider code.

2. Filtering

When a user requests data from a northbound interface, ConfD will call all the data providers responsible for the requested data and will return that data to the user. By default, filtering happens in the ConfD core engine. If a query filter was present in the request, the ConfD core engine applies the filter. i.e All the data has to move from the Data Provider to the ConfD core engine.

The List Filters feature, which was introduced in ConfD 7.1, enables the passing of filter information down to Data Providers. This feature is only for filtering YANG lists. All other filtering continues to be done in the ConfD core engine. The List Filters feature is of most use for lists which can scale up to a large amount of data or in those implementations where gathering the data in the backend is an expensive operation. This allows for the list data to be returned to the ConfD core engine already filtered thus saving time and increasing performance in most cases. Less data has to be moved between the Data Provider and the ConfD core engine which also means the ConfD core engine has to invoke fewer callbacks to the Data Provider.

Note that List Filters is a Data Provider API feature. It can be either configuration data or operational data for which a Data Provider application, using the DPAPI, is responsible.

3. The List Filters Feature

In order to support List Filters, a new mechanism has been introduced in order to allow a Data Provider to receive and parse the filter information which is passed in by ConfD during the get_next(), get_next_object(), find_next(), or find_next_object() callback invocations for lists.

During the first call to these callbacks, i.e., next == -1, ConfD will pass the filter information that should be applied to every list instance returned by the Data Provider.

More details about List Filters can be found in the ConfD User Guide in Chapter 8, section 8.13, "Using List Filters."

4. How List Filters Work

A Data Provider application needs to tell ConfD to pass in the filter information using the flag CONFD_DATA_WANT_FILTER during the registration of data callbacks.

For example:

```
…
struct confd_data_cbs data;
…
data.get_elem = get_elem;
data.get_next = get_next;
data.flags = CONFD_DATA_WANT_FILTER;
strcpy(data.callpoint, arpe__callpointid_arpe);
…
if (confd_register_data_cb(dctx, &data) == CONFD_ERR)
        confd_fatal("Failed to register data cb \n");
…
```

The first call to get_next() will contain the filter information that needs to be read and stored somewhere for future use by the Data Provider.

Subsequent calls to get_next() will not have this filter information set. A Data Provider application could decide to persist this filter information anywhere in the code or it could make use of the cb_opaque field of the transaction to persist this data across multiple calls to get_next() or to the other callbacks mentioned above where the filter information can be present.

Filter information is stored in a new data structure confd_list_filter which is defined in confd_lib.h:

```
enum confd_list_filter_type {
  CONFD_LF_OR     = 0,
  CONFD_LF_AND    = 1,
  CONFD_LF_NOT    = 2,
  CONFD_LF_CMP    = 3,
  CONFD_LF_EXISTS = 4,
  CONFD_LF_EXEC   = 5
};

enum confd_expr_op {
  CONFD_CMP_NOP                  = 0,
  CONFD_CMP_EQ                   = 1,
  CONFD_CMP_NEQ                  = 2,
  CONFD_CMP_GT                   = 3,
  CONFD_CMP_GTE                  = 4,
  CONFD_CMP_LT                   = 5,
  CONFD_CMP_LTE                  = 6,
  /* functions below */
  CONFD_EXEC_STARTS_WITH          = 7,
  CONFD_EXEC_RE_MATCH             = 8,
  CONFD_EXEC_DERIVED_FROM         = 9,
  CONFD_EXEC_DERIVED_FROM_OR_SELF = 10
};

struct confd_list_filter {
  enum confd_list_filter_type type;

  struct confd_list_filter *expr1; // OR, AND, NOT
  struct confd_list_filter *expr2; // OR, AND

  enum confd_expr_op op;           // CMP, EXEC
  struct xml_tag *node;            // CMP, EXEC, EXISTS
  int nodelen;                     // CMP, EXEC, EXISTS
  confd_value_t *val;              // CMP, EXEC
};
```

When the get_next() callback is called for the first time, i.e. next == -1, the filter information can be read using the following API:

```
int confd_data_get_list_filter(struct confd_trans_ctx *tctx,
                               struct confd_list_filter **filter);
```

When the Data Provider is done using the filter information, it is the Data Provider's responsibility to free the memory allocated by this API by using:

```
void confd_free_list_filter(struct confd_list_filter *filter);
```

Here is a snippet of code making use of these two APIs:

```
static int get_next(struct confd_trans_ctx *tctx, confd_hkeypath_t *keypath,
                    long next)
{
    struct arpdata *dp = tctx->t_opaque;
    struct aentry *curr;
    confd_value_t v[2];
    struct confd_list_filter *filter=NULL;
    int ret=0;

    if (next == -1) {  /* first call */
        if (need_arp(dp)) {
            if (run_arp(dp) == CONFD_ERR)
                return CONFD_ERR;
        }

        /* Get the filter data */

        ret = confd_data_get_list_filter(tctx, &filter);

        if (filter != NULL) {
            /* Here the application parses the filter information*/
            confd_free_list_filter(filter);
        }

        /* End of filter handling */

        curr = dp->arp_entries;
    }
…
```

A filter is a Boolean expression following the informal syntax:

<expr> = <expr> 'or' <expr>
/<expr> 'and' <expr>
/'not' <expr>
/'cmp' <op> <node> <value>
/'exec' <func> <node> <value>
/'exists' <node>

<op> stands for operation and can have the following values:
<op> = 'eq' / 'gt' / 'lt' / 'gte' / 'lte' / 'neq'

<func> maps to XPath functions:
<func> = 're-match' / 'starts-with' / 'derived-from' / 'derived-from-or-self'

<node> is a tagpath representing the path from the list node to the child node being filtered. The child node can be a leaf, a leaf-list, or, if we're checking for existence, it can also be a presence container or a list node (in which case the filter is checking if the list is empty or not).

Example:

```
list arpe {
      key "ip ifname";
      max-elements 1024;
      leaf ip {
        type inet:ip-address;
      }
      leaf ifname {
        type string;
      }
      leaf hwaddr {
        type string;
        mandatory true;
      }
      leaf permanent {
        type boolean;
        mandatory true;
      }
      leaf published {
        type boolean;
        mandatory true;
      }
    }
```

```
Filter examples:



To return all the instances with
'permanent' equals 'true':
permanent = "true"

To return all the instances with
'ip' value starting with '10.' And
'published' equals 'true':
starts-with(ip, "10.") and pub-
lished = "true"
```

5. NETCONF Filter Support

The NETCONF protocol, defined in RFC 6241, supports filtering of data using the <filter> element.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <get>
    <filter>
      <arpentries xmlns="http://tail-f.com/ns/example/arpe">
         <arpe>
            <permanent>false</permanent>
         </arpe>
      </arpentries>
    </filter>
  </get>
</rpc>
```

This is called subtree filtering and is the default type of filtering.

XPath filtering can also be used if the :xpath capability is supported by the NETCONF server.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <get>
    <filter xmlns:t="http://tail-f.com/ns/example/arpe" type="xpath"
        select="/t:arpentries/t:arpe[t:permanent='false']" />
  </get>
</rpc>
```

With the XPath filter, a user has to specify an XPath expression in the 'select' attribute to be able to specify the filter.

Here is another example of the XPath filter type, making use of the “starts-with()" XPath function:

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <get>
    <filter xmlns:t="http://tail-f.com/ns/example/arpe" type="xpath"
        select="/t:arpentries/t:arpe/t:hwaddr[starts-with(.,'1')]" />
  </get>
</rpc>
```

The NETCONF RFC provides more details on XPath filtering as well as subtree filtering.

By default, these filters are applied by the ConfD core engine once the Data Providers have returned the data. If a Data Provider has implemented List Filters, the ConfD core engine will provide these filter requests to the Data Provider application.

6. Considerations

When enabling this feature in Data Provider code, the developer should decide whether ConfD should still perform filtering after it receives the filtered data. This is unnecessary if the Data Provider can fulfill the filtering task entirely before returning the data to ConfD.

If the Data Provider can completely apply the filter, it should instruct ConfD not to do any additional filtering of the returned data using the flag CONFD_TRANS_CB_FLAG_FILTERED before returning the reply in the get_next() callback or in the other callbacks where the List Filters feature can be used. This decision can be made programmatically if the filter information read can't be fulfilled entirely in some cases.

```
/* Tells ConfD the data is already filtered and no need to filter */
tctx->cb_flags = CONFD_TRANS_CB_FLAG_FILTERED;

confd_data_reply_next_key(tctx, &v[0], 2, (long)curr->next);
```

Even if the Data Provider only does partial filtering, performance can still be improved because the data set that the ConfD core engine needs to filter has already been filtered, i.e., reduced, to some extent.

An example can be found in your ConfD installation in the directory $CONFD_DIR/examples.confd/dp/filters.

7. Summary

This application note has provided an introduction to the List Filters feature; what it is, how it works, and how to implement it in your Data Provider code.

By making use of this feature in your Data Provider application code, you can increase performance as a result of moving less data between your Data Provider and the ConfD core engine as well as potentially reduce the compute load inside your network device.

8. For More Information

For more information about ConfD, visit https://www.tail-f.com

ConfD User Guide, Chapter 8, section 8.13: "Using List Filters"

RFC 6241 "Network Configuration Protocol (NETCONF 1.1)":
https://tools.ietf.org/html/rfc6241

tail-f a Cisco company

www.tail-f.com
info@tail-f.com

Corporate Headquarters

Sveavagen 25
111 34 Stockholm
Sweden
+46 8 21 37 40