CONFDO OPERATIONAL DATA AND EXTERNAL DATABASES
## Table of Contents

1. Introduction ............................................................................................................ 3

2. ConfD CDB Database and Operational State Data ............. 3

3. A Simple YANG Model Example ............................................................. 6

4. Ways of Filtering Data over NETCONF .................................................... 7

5. A Pull Mode Operational Data Provider Example .......... 10

6. ConfD CDB Operational Datastore .................................................. 18

7. Integrating the Improved 5-c_stats Example with an External Database ...... 24

8. PostgreSQL Database Integration with a ConfD Data Provider Application .................. 26

9. Apache Cassandra Database ............................................................. 38

10. Summary ............................................................................................................ 48

11. For More Information .................................................................................. 48
1. Introduction

ConfD's main strength is to quickly provide end user operators and higher-level management systems a set of full featured, high-quality, programmable northbound interfaces to manage the configuration of network elements using transactions to provision services. Because operational state data is also important, ConfD provides various APIs to integrate with components of the network element's application architecture for both configuration and operational state data. This application note describes and explains some advanced concepts of the Data Provider API (DP API) related to operational state data for the “pull” approach (as described by the previously released “Inside the ConfD Operational Data Provider API” application note) when the data provider is an external database. i.e. Implementing an operational data provider application to service on-demand operational state data retrieval by fetching the requested data from, for example, a PostgreSQL or Cassandra database. The “push” approach, where operational state data is written to CDB in ConfD, is the alternative approach and will be covered in the scope of the example that we use in this application note as an alternative to consider when investigating whether your particular use-case requires external operational data or if the ConfD CDB operational datastore will address your requirements.

We therefore highly recommended complementing this application note with the previously released “Inside the ConfD Operational Data Provider API” application note and the ConfD User Guide chapter on operational state data, external database, and the DP API, CDB API, and Management Agent API (MAAPI) documentation.

The goal of this application note is to give some guidance to the many use cases and features supported by ConfD for retrieving operational data using, for example, NETCONF and when and how to provide the requested operational data from ConfD’s CDB, a cache, or an external database such as PostgreSQL or Apache Cassandra.

2. ConfD CDB Database and Operational State Data

For configuration data the transactional, ACID compliant built-in ConfD CDB database is used in the large majority of use cases. CDB is built around ACID transactions. Therefore, any configuration change performed by, for example, a NETCONF client either happens completely or nothing is changed in the data store.

For operational data, i.e. status information, the applications themselves register as data providers most of the time. There is usually no point in storing this kind of potentially rapidly changing data in a database. The values are simply computed when an operator wants to see them. For certain types of operational state data, e.g. performance and alarm data, it may make sense to store it in a database. The CDB database has a separate operational datastore available for this purpose, should you wish to use it. The CDB operational datastore can be updated by backend applications with multiple operations.
ConfD Operational Data and External Databases

over the CDB API without atomicity or by using the ConfD Management Agent API (MAAPI) which will apply all changes at once using the ConfD transaction manager even though the operational datastore is not transactional thereby fulfilling the ACID properties 100% as the ConfD CDB configuration datastores do. For example, using the CDB API or MAAPI to update the data or storing the data persistently is optional for the CDB operational datastore.

As soon as a YANG model describing what the device or virtual instance can do has been created, it can be compiled and loaded into ConfD. ConfD will then automatically render all of the described elements in all the management interfaces, e.g. NETCONF, RESTCONF, CLI, etc. The CDB database schema is composed of the YANG models. CDB picks up any changes to the model automatically and translates any existing contents in the database to the new schema. Every list and leaf in the YANG models are also mapped to a provider, either CDB or data provider applications. Subscriber applications are notified of changes to relevant parts of the configuration. Feeder applications inject operational data, e.g. performance data, into the CDB operational data store.

The task of an operational data provider in push mode (figure 2) a.k.a. a feeder application (figure 1) is to repeatedly store operational data, e.g. performance readings, into the CDB operational datastore using the CDB API and/or MAAPI. The operational data then resides in CDB and all the northbound agents, including NETCONF, can read the operational data directly from CDB.

On the other hand, a data provider application in pull mode knows the current value of something in the data model. ConfD will contact this application through the DP API to get the current value when requested by an operator through any of the northbound agents, e.g. show CPU temperature. This data is not stored anywhere in CDB, but may optionally be cached for a configurable time by ConfD’s core engine.

Figure 1 YANG models describing both configuration and status information are mounted at the root and on top of each other.
An external database to ConfD which holds only operational data, e.g. status data, is just another pull mode data provider to ConfD. We will in this application note take a look at how two popular databases can be integrated with ConfD using a pull mode data provider application.
3. A Simple YANG Model Example

In this application note, we will be using, from the ConfD examples set, the examples. confd/intro/5-c_stats ARP table YANG model as a YANG reference example model for the data provider, operational datastore, and external DB data provider examples.

module arpe {
    namespace "http://tail-f.com/ns/example/arpe"
    prefix arpe;

    import ietf-inet-types {
        prefix inet;
    }
    import tailf-common {
        prefix tailf;
    }

    container arpentries {
        config false;
        tailf:callpoint arpe;
        list arpe {
            key "ip ifname";
            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;
            }
        }
    }
}
Now let’s say we have a SSL VPN tunnel setup on our laptop to get secure access to our workplace infrastructure. This results in an ARP table that looks something like this when shown from ConfD’s CLI:

<table>
<thead>
<tr>
<th>IP</th>
<th>IFNAME</th>
<th>HWADDR</th>
<th>PERMANENT</th>
<th>PUBLISHED</th>
</tr>
</thead>
<tbody>
<tr>
<td>92.168.98.1</td>
<td>en0</td>
<td>a0:a9:a5:ab:a0:9a</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>224.0.0.251</td>
<td>IP</td>
<td>a1:a0:5e:a0:a0:fb</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>239.255.255.250</td>
<td>IP</td>
<td>a1:a0:5e:7f:ff:fa</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

4. Ways of Filtering Data over NETCONF

First, let’s briefly cover ways to retrieve data from a table over NETCONF. In this case, operational state data tagged with “config false” in the YANG model. Further, let’s say we want to filter out everything except for this row containing only IP address, interface name, and MAC address:

<table>
<thead>
<tr>
<th>IP</th>
<th>IFNAME</th>
<th>HWADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.98.1</td>
<td>en0</td>
<td>e0:b9:e5:eb:d0:9a</td>
</tr>
</tbody>
</table>

The IETF NETCONF RFC 6241 defines a <get> base protocol operation RPC to retrieve state information. There are two ways to get the data above using the RFC 6241 NETCONF <get> operation, where only one is mandatory for NETCONF support, i.e. with a subtree filter type RPC:

```xml
<rpc message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <arpentries xmlns="http://tail-f.com/ns/example/arpe">
        <arpe>
          <ip>192.168.98.248</ip>
          <ifname/>
          <hwaddr/>
        </arpe>
      </arpentries>
    </filter>
  </get>
</rpc>
```
The RPC reply with the data from the ConfD NETCONF server:

```xml
<rpc-reply message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <arpentries xmlns="http://tail-f.com/ns/example/arpe">
      <arpe>
        <ip>192.168.98.248</ip>
        <ifname>en0</ifname>
      </arpe>
    </arpentries>
  </data>
</rpc-reply>
```

Since ConfD supports all optional NETCONF features, including the :xpath capability in RFC 6241, you can also filter data using the XPath filter type. To get the same result as the above subtree filter with an XPath expression:

```xml
<rpc message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter select=' /arpentries/arpe[ip="192.168.98.248"]/hwaddr' type="xpath"/>
  </get>
</rpc>
```

The xpath filter type RPC reply will be identical to the one returned from the ConfD NETCONF server for the above subtree filter type RPC.

In addition to the RFC 6241 <get> operation, you can also use the ConfD Query API that is announced in the NETCONF <hello> message capability exchange if you have enabled it in the ConfD configuration, e.g. in the confd.conf file:

```xml
<confdConfig>
  ...
  <netconf>
    ...
    <capabilities>
      ...
      <query>
        <enabled>true</enabled>
      </query>
    </capabilities>
  </netconf>
</confdConfig>
```
The ConfD Query capability provides RPC operations to run advanced search queries to retrieve data over NETCONF, RESTCONF, MAAP, and JSON-RPC similar to the query functionality using the SELECT command in the PostgreSQL and Cassandra database interfaces.

Using the Query API, if we want to send a query and get the entire result with only one request RPC, we can do this by using the <immediate-query> extension RPC. The simplest form of query to filter out the same data as with the <get> requests:

```xml
<rpc message-id="1" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <immediate-query xmlns="http://tail-f.com/ns/netconf/query">
    <foreach>
      /arpentries/arpe[ip='192.168.98.248']
    </foreach>
    <select>
      <expression>ip</expression>
      <result-type>string</result-type>
    </select>
    <select>
      <expression>ifname</expression>
      <result-type>string</result-type>
    </select>
    <select>
      <expression>hwaddr</expression>
      <result-type>string</result-type>
    </select>
    <sort-by>name</sort-by>
  </immediate-query>
</rpc>
```

The RPC reply with the query-result from the ConfD NETCONF server:

```xml
<rpc-reply message-id="1"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <query-result xmlns="http://tail-f.com/ns/netconf/query">
    <result>
      <select>
        <value>192.168.98.248</value>
      </select>
      <select>
        <value>en0</value>
      </select>
      <select>
        <value>80:a6:50:26:d4:94</value>
      </select>
    </result>
  </query-result>
</rpc-reply>
```
5. A Pull Mode Operational Data Provider Example

The examples/confd/intro/5-c_stats ConfD pull mode operational data provider example is described in detail in the ConfD User Guide and by the example itself. Therefore, we will not dive into details for that example. Instead, we will dive right into showing how we can extend that example to efficiently support retrieving the data from ConfD over, for example, NETCONF in the ways to filter out the data we want that we just described.

To allow ConfD to use only one data provider callback to retrieve all or selected parts of our ARP table/list example using the fewest callback invocations and replies as possible, we need to implement the ConfD DP API get_next_object() and find_next_object() callbacks and reply with the confd_data_reply_next_object_arrays() function as defined by the libconfd C-API. The ConfD API is also available in Python, Erlang and Java language bindings. We will see a Java example of later in this application note.

Figure 4 Overview of the NETCONF get and query operations handled by ConfD where ConfD invokes the find_next_object() callback and the application reply by calling confd_data_reply_next_object_arrays().
When a request for data is received, the ConfD NETCONF server passes the request through the query, XPath, and transaction back plane in the ConfD core depending on what type of data request it is. All requests to be handled by a data provider application outside of ConfD will pass through the C API in the ConfD core engine which will then invoke the callbacks that ConfD needs in order to retrieve the requested data, get the reply from the data provider application, and pass it to the requestor in a NETCONF <rpc-reply>.

In order to not have to count the number of rows/list-entries in the ARP table/list, if ConfD needs to get the number of entries in the list/table, we can in this example add a counter “n_ae” to the arpdata struct where the table/list is defined and stored after the ARP table has been updated.

```c
struct arpdata {
    struct aentry *arp_entries;
    struct timeval lastparse;
    int n_ae;
};
```

This is then updated in a few places where we initialize and handle the ARP table:

```c
static void free_arp(struct arpdata *dp) {
    struct aentry *ae = dp->arp_entries;
    while (ae) {
        struct aentry *next = ae->next;
        if(ae->hwaddr) free(ae->hwaddr);
        if(ae->iface) free(ae->iface);
        free(ae);
        ae = next;
    }
    dp->arp_entries = NULL;
    dp->n_ae = 0;
}

/* parse output from arp -an for this transaction */
static int run_arp(struct arpdata *dp) {
    ...
    add_aentry(&dp->arp_entries, ae);
    dp->n_ae++;
    ...
}
Now, we can use this counter when we add our num_instances() callback implementation to the arpstat.c file in the examples.confdir_examples.confdir examples example.

```c
static int num_instances(struct confd_trans_ctx *tctx,
                         confd_hkeypath_t *keypath)
{
    confd_value_t v;
    struct arpdata *dp = tctx->t_opaque;

    CONFD_SET_INT32(&v, dp->n_ae);
    confd_data_reply_value(tctx, &v);
    return CONFD_OK;
}
```

Next, let’s take a look at the get_next_object() implementation:

```c
static int max_nobjs = 100; // Chunk size hardcoded here to simplify

static int get_next_object(struct confd_trans_ctx *tctx,
                            confd_hkeypath_t *keypath, long next)
{
    int i;
    confd_value_t *v;
    struct confd_next_object *obj;
    struct arpdata *dp = tctx->t_opaque;
    struct aentry *curr;
    if (next == -1) { /* first call */
        if (need_arp(dp)) {
            if (run_arp(dp) == CONFD_ERR)
                return CONFD_ERR;
        }
        curr = dp->arp_entries;
    } else {
        curr = (struct aentry *)next;
    }
    if (curr == NULL) {
        confd_data_reply_next_key(tctx, NULL, -1, -1);
        return CONFD_OK;
    }

    obj = malloc(sizeof(struct confd_next_object) * (max_nobjs + 2));
    v = (confd_value_t *) malloc(sizeof(confd_value_t) * max_nobjs * navals);

    for (i = 0; curr != NULL && i < max_nobjs; curr = curr->next, i++)
    {
        /* Collect max_nobjs or as many as there is rows in the table */
```
The get_next_object() callback function begins with checking the next value and getting the current ARP row/list-entry accordingly.

We then allocate enough memory to cover the hardcoded max chunk of list entries that we want to return to ConfD. For this type of simple list, returning 100 list entries at a time will give you at least close to optimal performance pushing these chunks in to the ConfD object cache while ConfD passes them to the client over, for example, the NETCONF interface.

We construct an array of objects, which in turn is an array of values. An object is here one list-entry/row in the list/table. Each object consists of values which are all leafs in this example. Only one leaf, the MAC address, may not always exist in the system ARP table when we populate it. If it doesn’t, we just use a CONFD_NO_EXIST() value to indicate that to ConfD.

We set the next value each time we add an entry to ensure that subsequent calls will pass us the pointer to the next value in the ARP entry array/list that we, as mentioned earlier, populate at startup and when a timer expires.

If we did reach the end of the list before we reached our max chunk, we set the last list entry/row to NULL to indicate to ConfD that this is the end of the list.
Lastly, we return the list entries to ConfD using confd_data_reply_next_object_arrays(). We could have used confd_data_reply_next_object_tag_value_arrays() here where we would have added tags together with the values. This means slightly more data to fill in, but the performance is about the same and it would have been a bit easier to see what’s going on when filling in the values into the list entries.

Next, let’s take a look at the find_next_object() implementation:

```c
cstatic int find_next_object(struct confd_trans_ctx *tctx, 
    confd_hkeypath_t *kp, 
    enum confd_find_next_type type, 
    confd_value_t *keys, int nkeys)
{
    int pos = -1;
    int ipos;
    confd_value_t *v;
    struct confd_next_object *obj;
    int i;
    struct arpdata *dp = tctx->t_opaque;
    int n_list_entries, nobjs;
    struct aentry *ae;
    struct in_addr ip;
    char *iface;

    if (need_arp(dp)) {
        if (run_arp(dp) == CONFD_ERR)
            return CONFD_ERR;
    }

    n_list_entries = dp->n_iae;
    ae = dp->arp_entries;

    switch (nkeys) {
    case 0:
        if(n_list_entries > 0) {
            pos = 0; /* No keys provided => the first entry will always be
                "after" */
        }
        break;
    case 1:
        /* one key provided => find the first entry "after"
        - since there can never be a "same" entry, 'type' is not
        relevant,
        and an entry with same first key is also "after" */
        ip = CONFD_GET_IPV4(&keys[0]);
        ipos = 0;
        for (; ae != NULL; ae = ae->next) {
            if (memcmp(&ae->ip4, &ip, sizeof(struct in_addr)) >= 0) {
```
pos = ipos;
break;
}
  ipos++;
}
break;
case 2:
  /* both keys provided => find first entry "after" or "same",
   depending on 'type' */
  ip = CONFD_GET_IPV4(&keys[0]);
  iface = (char*)CONFD_GET_BUFPTR(&keys[1]);

  switch (type) {
  case CONFD_FIND_NEXT:
    /* entry must be "after" */
    ipos = 0;
    for (; ae != NULL; ae = ae->next) {
      if (memcmp(&ae->ip4, &ip, sizeof(struct in_addr)) > 0 ||
          (memcmp(&ae->ip4, &ip, sizeof(struct in_addr)) == 0 &&
           strcmp(iface, ae->iface) > 0)) {
        pos = ipos;
        break;
      }
      ipos++;
    }
    break;
  case CONFD_FINDSAME_OR_NEXT:
    /* entry must be "same" or "after" */
    ipos = 0;
    for (; ae != NULL; ae = ae->next) {
      if (memcmp(&ae->ip4, &ip, sizeof(struct in_addr)) > 0 ||
          (memcmp(&ae->ip4, &ip, sizeof(struct in_addr)) == 0 &&
           strcmp(iface, ae->iface) >= 0)) {
        pos = ipos;
        break;
      }
      ipos++;
    }
    break;
  default:
    confd_trans_seterr(tctx, "invalid number of keys: %d", nkeys);
    return CONFD_ERR;
  }

  if (n_list_entries - pos > max_nobjs) {
    nobjs = max_nobjs + pos;
ConfD Operational Data and External Databases

```c
} else {
    nobjs = n_list_entries;
}

v = (confd_value_t *) malloc(sizeof(confd_value_t) * nobjs * navals);
obj = malloc(sizeof(struct confd_next_object) * (nobjs + 2));

if (pos != -1) {
    for (i = 0; pos + i < nobjs; i++) {
        obj[i].v = &v[i * navals];

        CONFD_SET_IPV4(&(obj[i].v[0]), ae->ip4);
        CONFD_SET_STR(&(obj[i].v[1]), ae->iface);
        if (ae->hwaddr == NULL) {
            CONFD_SET_NOEXISTS(&(obj[i].v[2]));
        } else {
            CONFD_SET_STR(&(obj[i].v[2]), ae->hwaddr);
        }
        CONFD_SET_BOOL(&(obj[i].v[3]), ae->perm);
        CONFD_SET_BOOL(&(obj[i].v[4]), ae->pub);

        obj[i].n = navals;
        obj[i].next = (long)ae->next;//-1;
        ae = ae->next;
    }
    if (pos + i == n_list_entries)
        obj[i++].v = NULL;
    confd_data_reply_next_object_array(tctx, obj, i, 0);
} else {
    confd_data_reply_next_object_array(tctx, NULL, -1, -1);
}
free(v);
free(obj);
return CONFD_OK;
}
```

The `find_next_object()` callback is the most complex callback for this type of list. Here, we begin with checking if we need to refresh the linked list of representation of the ARP table. We then check the keys provided and go through the linked list searching for a match. If there is no match, we will end up calling `confd_data_reply_next_object_array(tctx, NULL, -1, -1)`.

Since we here know the number of list entries/objects/rows that we will return to ConfD, we can be a bit smarter (even though it is likely insignificantly smarter...) than `get_next_object()` and allocate only exactly as much memory as we need to cover the number of list entries that we want to return to ConfD.

Lastly, we return the list entries to ConfD using `confd_data_reply_next_object_array()` just as we did from the `get_next_object()` callback.
There is also a “navals” or number or ARP values/leafs global variable in that you may have noticed that the get/find_next_object() implementations use. This is the number of leafs in the list entry, i.e. columns in the table. To avoid hardcoding that number in our code, we use the schema derived from the YANG model to find out how many columns the table has when we start the data provider application. This is done right after we have registered our callbacks, including our get/find_next_object() + num_instances() callbacks, and loaded the schema to be able to access it:

```c
static int navals;

int main(int argc, char *argv[]) {
...
    struct confd_cs_node *object;
...
    memset(&data, 0, sizeof (struct confd_data_cbs));
    data.get_elem = get_elem;
    data.get_next = get_next;
    data.num_instances = num_instances;
    data.get_next_object = get_next_object;
    data.find_next_object = find_next_object;
...
    object = confd_cs_node_cd(NULL, "/arp:arpentries/arp");
    navals = confd_max_object_size(object);
...
```

The 5-c_stats example in ConfD could be implemented in a different way. In ConfD 6.6, the current version as of this writing, the example caches the ARP table in a linked list for 2 seconds. When the get_next() callback is invoked with next = -1, the code checks if the ARP table cached in the linked list is over 2 seconds old. If it is, the ARP table is re-read from the system. Instead of reading the system’s ARP table into our own linked list cache, we could utilize the ConfD Data Provider operational data cache to hold the latest read for 2 seconds before our data provider is called again. (The operational data cache is a feature which was added to ConfD after the 5-c_stats example was originally created.) In this way, we could just re-read the system’s ARP table every time our DP is called and let ConfD do the caching instead.

See the “Inside the ConfD Operational Data Provider API” application note and ConfD’s documentation for details on the ConfD operational data cache.
6. ConfD CDB Operational Datastore

As you may have already noticed with the 5-c_stats example arpstat.c application, we take the ARP table from our system and cache it in our own linked list with a 2 second staleness check. So, why don’t we store it in CDB instead of some linked list cache in our “pull mode” data provider application? Good question. For this use case, it would indeed be a good option with less code and effort needed to just enable the CDB operational data store and “push” the data into CDB periodically. This example will, for the sake of demonstration purposes, do so every 10 seconds.

This way, a data provider “pull mode” application can be replaced with a far simpler “push mode” application and the performance of the manager reading the operational status data over, for example, NETCONF will enjoy improved performance as the data is read directly from the CDB operational data store instead of calling a data provider running outside of ConfD.

![Figure 5 Pushing the system ARP table into the CDB operational datastore](image)
The entire arpstat.c application will for this use case be replaced by a simpler “push mode” application that re-reads the system’s ARP table every 10 seconds and writes the table into the CDB operational datastore using a single call to maapi_set_values() in MAAPI.

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/poll.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>

#include <confd_lib.h>
#include <confd_dp.h>
#include <confd_maapi.h>
#include "arpe.h"

#ifndef __QNX__
/* too hard to figure out which #defines are needed for this to get included */
extern char *strdup(const char *string);
#endif

#include <confd_lib.h>
#include <confd_dp.h>
#include <confd_maapi.h>
#include "arpe.h"

#if !defined(Linux) && !defined(__NetBSD__) && !defined(__FreeBSD__) && 
!defined(__QNX__) && !defined(Darwin)
#warning "arpstat: Not tested on this OS"
#endif

#define INTERVAL 10 // How often in seconds we read the ARP table into CDB
static int max_nobjs = 100; // Chunk size hardcoded here to simplify
static int navals;
static int msock, th;

/*******************************/
static volatile int do_run_arp = 0;

static void catch_alarm(int sig)
{
    do_run_arp++;
}

/*******************************/
/* parse output from arp -an for this transaction */
static int run_arp()
{
  char *sep = " (?<\n"
  FILE *fp;
  char buf[BUFSIZ];
  confd_tag_value_t *v = (confd_tag_value_t *)
  malloc(sizeof(confd_tag_value_t) * (navals+2) * max_nobjs);
  int i = 0;
  struct in_addr ip4;
  char *hwaddr;
  char *ifname;
  int perm, pub, ret;

  if ((fp = popen("arp -an", "r")) == NULL)
    return CONFD_ERR;

  if ((th = maapi_start_trans(msock, CONFD_OPERATIONAL, CONFD_READ_WRITE)) < 0) {
    confd_fatal("failed to start trans\n");
  }
  maapi_set_namespace(msock, th, arpe.ns);

  while (fgets(&buf[0], BUFSIZ, fp) != NULL) {
    char *cp = strtok(&buf[0], sep);

    /* Now lazy parse lines like */
    /* ? (192.168.1.1) at 00:0F:B5:EF:11:00 [ether] on eth0 */
    /* slightly different arp output on Linux and BSD */
    ip4.s_addr = inet_addr(cp);
    /* skip "at" */
    assert(strcmp(strtok(NULL, sep), "at") == 0);
    cp = strtok(NULL, sep);

    if ((strcmp(cp, "incomplete") == 0)) {
      assert(strcmp(strtok(NULL, sep), "on") == 0);
      cp = strtok(NULL, sep);
    } else if ((strcmp(cp, "<from _interface>") == 0)) {
      cp = strtok(NULL, sep);
      while (cp) {
        if (strcmp(cp, "on") == 0) {
          cp = strtok(NULL, sep);
          break;
        }
        cp = strtok(NULL, sep);
      }
    } else {
      /* some common error cases handled, get real hw addr */
      hwaddr = strdup(cp);
    }
  }
}
while (1) {
    cp = strtok(NULL, sep);
    if (cp == NULL)
        break;
    else if (strcmp(cp, "PERM") == 0)
        perm = 1;
    else if (strcmp(cp, "PUB") == 0)
        pub = 1;
    else if (strcmp(cp, "[ether"]") == 0)
    ;
    else if (strcmp(cp, "on") == 0) {
        cp = strtok(NULL, sep);
        break;
    }
}

/* cp should now point to the interface name
   - this is required since it is a key */
if (cp) {
    CONFD_SET_TAG_XMLBEGIN(&v[i], arpe_arpe, arpe_ns); i++;
    CONFD_SET_TAG_IPV4(&v[i], arpe_ip, ip4); i++;
    ifname = strdup(cp);
    CONFD_SET_TAG_STR(&v[i], arpe_ifname, ifname); i++;
    CONFD_SET_TAG_STR(&v[i], arpe_hwaddr, hwaddr); i++;

    /* Some OSes have perm/pub after interface name */
    while ((cp = strtok(NULL, sep)) != NULL) {
        if (strcmp(cp, "permanent") == 0)
            perm = 1;
        else if (strcmp(cp, "published") == 0)
            pub = 1;
    }

    CONFD_SET_TAG_BOOL(&v[i], arpePermanent, perm); i++;
    CONFD_SET_TAG_BOOL(&v[i], arpePublished, pub); i++;

    CONFD_SET_TAG_XMLEND(&v[i], arpe_arpe, arpe_ns); i++;
    if(i == 100) {
        if ((ret = maapi_delete(msock, th, "arpentries/arpe")) < 0) {
            confd_fatal("maapi_delete() failed");
        }
        if ((ret = maapi_set_values(msock, th, v, i, "/arpentries") < 0) {
            confd_fatal("maapi_set_values() failed");
        }
        if ((ret = maapi_apply_trans(msock, th, 0)) < 0) {
            confd_fatal("maapi_apply_trans() failed");
        }
        maapi_finish_trans(msock, th);
        if ((th = maapi_start_trans(msock, CONFD_OPERATIONAL, CONFD_READ_WRITE)) < 0) {
            confd_fatal("failed to start trans\n");
        }
    }
}
```c
if(i > 0) {
    if ((ret = maapi_set_namespace(msock, th, arpe_ns)) < 0) {
        confd_fatal("maapi_set_namespace() failed");
    }
    if ((ret = maapi_set_values(msock, th, v, i, "/arpentries")) < 0) {
        confd_fatal("maapi_set_values() failed");
    }
    if ((ret = maapi_apply_trans(msock, th, 0)) < 0) {
        confd_fatal("maapi_apply_trans() failed");
    }
}
maapi_finish_trans(msock, th);
free(v);
pclose(fp);
return CONFD_OK;
}

/*****************************************************************************/
int main(int argc, char *argv[]) {
    struct sockaddr_in addr;
    int debuglevel = CONFD_TRACE;
    int interval = INTERVAL;
    time_t now;
    struct tm *tm;
    struct itimerval timer;
    struct confd_cs_node *object;
    int c;
    struct confd_ip ip;
    const char *groups[] = { "admin" };
    char *context = "system";

    while ((c = getopt(argc, argv, "idpts")) != -1) {
        switch(c) {
          case 'i':
            interval = atoi(argv[1]);
            break;
```
case 'd':
    debuglevel = CONFD_DEBUG;
    break;
  case 'p':
    debuglevel = CONFD_PROTO_TRACE;
    break;
  case 't':
    debuglevel = CONFD_TRACE;
    break;
  case 's':
    debuglevel = CONFD_SILENT;
    break;
}
} /* initialize confd library */
confd_init("arpe_daemon", stderr, debuglevel);

addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_family = AF_INET;
addr.sin_port = htons(CONFD_PORT);

if (confd_load_schemas((struct sockaddr*)addr, sizeof(struct sockaddr_in)) != CONFD_OK)
    confd_fatal("Failed to load schemas from confd\n");

object = confd_cs_node_cd(NULL, "/arpe:arpentries/arpe");
navals = confd_max_object_size(object);

if ((msock = socket(PF_INET, SOCK_STREAM, 0)) < 0 ) {
    confd_fatal("failed to create socket\n");
}

if (maapi_connect(msock, (struct sockaddr*)addr, sizeof(addr)) < 0)
{
    confd_fatal("failed to connect to confd\n");
}

ip.af = AF_INET;
inet_pton(AF_INET, "127.0.0.1", &ip.ip.v4);

if ((maapi_start_user_session(msock, "admin", context, groups, sizeof(groups) / sizeof(*groups), &ip, CONFD_PROTO_TCP) != CONFD_OK)) {
    confd_fatal("failed to start user session");
}

signal(SIGALRM, catch_alarm);
Why are we using the ConfD MAAPI and not the CDB API to push the data into CDB? While there is an overhead of using MAAPI and going through the ConfD transaction manager to set operational data in CDB compared to going directly into CDB using the CDB API, the advantages of MAAPI in this use case is that we can easily delete the entire old list and write the new list in chunks of say 100 entries at a time into CDB without exposing our deletes and writes until we apply the data to the CDB operational datastore. This way, the delete is simpler and the manager, for example, a NETCONF client, will be confused by reading the data in the middle of when our push mode provider is deleting and writing chunks to CDB to replace the old table/list with the new one.

Just as with the operational data cache, the ability to write operational data to CDB using MAAPI in addition to the CDB API was introduced as a feature in ConfD after the 5-c_stats example was originally written.

7. Integrating the Improved 5-c_stats Example with an External Database

Finally, we are ready to move on to what the title of this application note promises. Integrating operational data as indicated by “config false” and “tailf:callpoint xyz” statements in the YANG model with an external database.

Why do we want to do this? Isn’t the CDB operational datastore sufficient? If you are looking to expose all of your state data over for NETCONF, RESTCONF or any other northbound APIs, then yes, use ConfD’s operational datastore for the data that you want to cache in a database. Compared to going outside of ConfD for the data, the
performance for the manager reading the data over NETCONF will be better and the schema and northbound interfaces, e.g. NETCONF, RESTCONF, will be automatically rendered from the YANG model. No application to maintain and update as the schema/YANG model evolves. Persistency and high availability replication of data? That can be done with CDB’s operational datastore as well. You just need to enable that feature in the YANG model and in the ConfD configuration file, confd.conf.

That being said, maybe you just need expose a small part of your massive amount of state data over the northbound management interfaces. Perhaps you do want to expose some data to the manager, but the rest will just be used for purposes other than managing the application/service. There are three use cases where you might want to use an external database for operational state data:

1. You might be passing large amounts and/or very frequently changing application state data around between application instances. The manager will never see this data as it is just used for passing state between your applications. You may also need to replicate the data to standby nodes to maintain high availability. This replication needs to be throttled or use ring mechanisms, eventual consistency schemes, etc.

2. You have a need for multiple indexes, joins on the fly, etc., to present your data in real time directly to, for example, a web interface and not over NETCONF. ConfD’s CDB provides the basic key indexing functionality and a schema derived from the YANG model, but you may need the tools and features provided by databases that specialize in searching and organizing massive amounts of state data in an efficient manner.

3. Legacy. Your project manager is somehow able to convince you that changes and new things are bad and very costly. You should stay safe and not change any applications that are currently integrated with a legacy database to adapt to the ConfD APIs. Not even the possibility of using a wrapper is safe enough for your project manager. Then doing an external operational state database integration may be your only remaining option.

Below we will show a suggested way of integrating two of the more popular external operational state databases used with ConfD. There are other databases out there as well, but hopefully the examples below will guide you if you see a need to integrate the database of your liking with ConfD in order to, for example, provide parts of the data over ConfD’s northbound interfaces or APIs such as MAAPI.

We will not dig into the details of the advantages of the two databases that we show an example integration with; they do a good job explaining this in their documentation themselves. Rather, the purpose of the example integration is to show a best practices integration implementation for ConfD users that need to do it due to any of the use cases listed above.
8. PostgreSQL Database Integration with a ConfD Data Provider Application

In the PostgreSQL 9.6.6 based example integration code below, we use the ConfD libconfd C API library. Since Postgres is written in C, it makes sense, together with a pull mode application that perform queries when data is requested by ConfD to use the Postgres libpq C Library. The example implements the relevant callback functions for the ARP table use case, including get_next_object() and find_next_object().

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/poll.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>

#include <confd_lib.h>
#include <confd_dp.h>
#include <confd_maapi.h>
#include "arpe.h"
```
#include <libpq-fe.h>

#ifndef Linux
  #endif
#endif

/* Our daemon context as a global variable */
static struct confd_dctx *dctx;
static int ctlsock;
static int workersock;
static int navals;

/* PostgreSQL connection */
static PGconn *conn;

static int s_init(struct confd_trans_ctx *tctx)
{
  confd_trans_set_fd(tctx, workersock);
  return CONFD_OK;
}

static int s_finish(struct confd_trans_ctx *tctx)
{
  return CONFD_OK;
}

static int get_next(struct confd_trans_ctx *tctx,
                     confd_hkeypath_t *keypath,
                     long next)
{
  confd_value_t v[2];

  int n_list_entries, pos;
  PGresult *res;
  char *ip, *iface;
  char str[BUFSIZ];

  if (next == -1) { /* ConfD want to get the whole table / list from the beginning */
    sprintf(str, "SELECT * FROM arpstats ORDER BY ip LIMIT 1");
    pos = 0;
  } else {
    /* ConfD wants the table / list from a specific index */
    pos = next;
    sprintf(str, "SELECT * FROM arpstats ORDER BY ip LIMIT 1 OFFSET

    res = (PGresult*) PQexec(conn, str);
`%d", (int)next);
}

res = PQexec(conn, str);

if (PQresultStatus(res) != PGRES_TUPLES_OK) {
    perror(PQresultErrorMessage(res));
}

n_list_entries = PQntuples(res);

if(n_list_entries == 0) {
    confd_data_reply_next_key(tctx, NULL, -1, -1);
} else {
    ip = PQgetvalue(res, 0, 0);
    CONFD_SET_STR(&v[0], ip);
    iface = PQgetvalue(res, 0, 1);
    CONFD_SET_STR(&v[1], iface);
    confd_data_reply_next_key(tctx, v, 2, pos+1);
}

PQclear(res);
return CONFD_OK;
}

/* Keypath example */
/* /arpentries/arpe{192.168.1.1 eth0} */
/*    2         1         0           */

static int get_object(struct confd_trans_ctx *tctx, confd_hkeypath_t *keypath)
{
    confd_value_t v[navals];
    char *ip = (char*)CONFDF_GET_BUFPTR(&keypath->v[0][0]);
    char *iface = (char*)CONFDF_GET_BUFPTR(&keypath->v[0][1]);
    char *hwaddr;
    PGresult *res = NULL;
    char str[BUFSIZ];

    sprintf(str, "SELECT * FROM arpstats WHERE ip = '%s' AND ifname = '%s'", ip, iface);
    res = PQexec(conn, str);

    if (PQresultStatus(res) == PGRES_TUPLES_OK && PQntuples(res)) {
        CONFD_SET_STR(&v[0], ip);
        CONFD_SET_STR(&v[1], iface);
        hwaddr = PQgetvalue(res, 0, 2);
CONF D _ SET _ STR(&v[2], hwaddr);
if (strcmp(PQgetvalue(res, 0, 3), "f") == 0)
    CONFD _ SET _ STR(&v[3], "0");
else
    CONFD _ SET _ STR(&v[3], "1");
if (strcmp(PQgetvalue(res, 0, 4), "f") == 0)
    CONFD _ SET _ STR(&v[4], "0");
else
    CONFD _ SET _ STR(&v[4], "1");
confd _ data _ reply _ value _ array(tctx, v, navals);
} else {
    confd _ data _ reply _ not _ found(tctx);
}
PQclear(res);
return CONFD _ OK;
}

static int num _ instances(struct confd _ trans _ ctx *tctx,
    confd _ hkeypath _ t *keypath)
{
    confd _ value _ t v;
PQresult *res = NULL;
char *count;

    res = PQexec(conn, "SELECT COUNT(*) FROM arpstats");
    if (PQresultStatus(res) != PGRES _ TUPLES _ OK) {
        perror(PQresultErrorMessage(res));
    }

    count = PQgetvalue(res, 0, 0);
    CONFD _ SET _ INT32(&v, atoi(count));
    confd _ data _ reply _ value(tctx, &v);
    return CONFD _ OK;
}

/* Keypath example */
/* /arpentries/arpe */
/* 2 1 */

static int max _ nobjs = 100; // Chunk size hardcoded here to simplify

static int get _ next _ object(struct confd _ trans _ ctx *tctx,
    confd _ hkeypath _ t *keypath, long next)
{
    int pos = 0;
    char *ip, *iface, *hwaddr;
PQresult *res = NULL;
char str[BUFSIZ];
int n _ list _ entries = 0;


confd_value_t *v;
struct confd_next_object *obj;
int i;

if (next == -1) {
    /* ConfD want to get the whole table / list from the beginning */
    sprintf(str, “SELECT * FROM arpstats ORDER BY ip LIMIT %d”, max_nobjs);
} else {
    /* ConfD wants the table / list from a specific index */
    pos = next;
    sprintf(str, “SELECT * FROM arpstats ORDER BY ip LIMIT %d OFFSET %d”, max_nobjs, (int)next);
}

res = PQexec(conn, str);

if (PQresultStatus(res) != PGRES_TUPLES_OK) {
    perror(PQresultErrorMessage(res));
}

n_list_entries = PQntuples(res);

if(n_list_entries == 0) {
    confd_data_reply_next_object_array(tctx, NULL, -1, -1);
    return CONFD_OK;
}

obj = malloc(sizeof(struct confd_next_object) * (n_list_entries + 2));
v = (confd_value_t *) malloc(sizeof(confd_value_t) * n_list_entries * navals);

for (i = 0; i < n_list_entries; i++) {
    /* Collect all the rows in the table / list entries */
    obj[i].v = &v[i * navals];
    ip = PQgetvalue(res, i, 0);
    CONFD_SET_STR(&(obj[i].v[0]), ip);
    iface = PQgetvalue(res, i, 1);
    CONFD_SET_STR(&(obj[i].v[1]), iface);
    hwaddr = PQgetvalue(res, i, 2);
    if (hwaddr == NULL) {
        CONFD_SET_NOEXISTS(&(obj[i].v[2]));
    } else {
        CONFD_SET_STR(&(obj[i].v[2]), hwaddr);
    }
    if (strcmp(PQgetvalue(res, i, 3), “f”) == 0)
        CONFD_SET_STR(&(obj[i].v[3]), “0”);
else
     CONFD_SET_STR(&(obj[i].v[3]), "1");
if (strcmp(PQgetvalue(res, i, 4), "f") == 0)
     CONFD_SET_STR(&(obj[i].v[4]), "0");
else
     CONFD_SET_STR(&(obj[i].v[4]), "1");

obj[i].n = navals;
/* If we pass -1 here we will invoke find_next_object. If we
instead pass next object ConfD will invoke get_next_object if there
are more objects to be fetched */
     obj[i].next = pos+i+1; //-1;
}
if (n_list_entries < max_nobjs)
     obj[i++].v = NULL;

confd_data_reply_next_object_arrays(tctx, obj, i, 0);
free(v);
free(obj);
PQclear(res);
return CONFD_OK;
}

static int find_next_object(struct confd_trans_ctx *tctx,
     confd_hkeypath_t *kp,
     enum confd_find_next_type type,
     confd_value_t *keys, int nkeys)
{
     PGresult *res = NULL;
     char str[BUFSIZ];
     int n_list_entries;

     confd_value_t *v;
     struct confd_next_object *obj;
     int i;

     switch (nkeys) {
     case 0:
     sprintf(str, "SELECT * FROM arpstats ORDER BY ip LIMIT %d", max_nobjs);
     res = PQexec(conn, str);
     if (PQresultStatus(res) != PGRES_TUPLES_OK)
         perror(PQresultErrorMesage(res));
     n_list_entries = PQntuples(res);
     if(n_list_entries > 0) {
     /* No keys provided => the first entry will always be "after" */
     ip_out = ip_in = PQgetvalue(res, 0, 0);
     iface_out = iface_in = PQgetvalue(res, 0, 1);
     }
break;
case 1:
    /* one key provided => find the first entry "after"
    - since there can never be a "same" entry, 'type' is not
    relevant,
    and an entry with same first key is also "after" */
ip_out = (char *)CONFD_GET_BUFPTR(&keys[0]);

    sprintf(str, "SELECT * FROM arpstats WHERE ip >= '%s' LIMIT %d",
ip_out, max_nobjs);
    res = PQexec(conn, str);
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
        perror(PQresultErrorMessage(res));
    n_list_entries = PQntuples(res);
    if(n_list_entries > 0) {
        ip_in = PQgetvalue(res, 0, 0);
        iface_out = iface_in = PQgetvalue(res, 0, 1);
    }
break;
case 2:
    /* both keys provided => find first entry "after" or "same",
     depending on 'type' */
ip_out = (char *)CONFD_GET_BUFPTR(&keys[0]);
iface_out = (char *)CONFD_GET_BUFPTR(&keys[1]);

    switch (type) {
    case CONFD_FIND_NEXT:
        /* entry must be "after" */
        sprintf(str, "SELECT * FROM arpstats WHERE ip > inet '%s' OR (ip
= '%s' AND lower(ifname) > lower('%s')) LIMIT %d", ip_out, ip_out,
iface_out, max_nobjs);
        break;
    case CONFD_FINDSAME_OR_NEXT:
        /* entry must be "same" or "after" */
        sprintf(str, "SELECT * FROM arpstats WHERE ip > inet '%s' OR (ip
= '%s' AND lower(ifname) >= lower('%s')) LIMIT %d", ip_out, ip_out,
iface_out, max_nobjs);
        break;
    }
    res = PQexec(conn, str);
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
        perror(PQresultErrorMessage(res));
    n_list_entries = PQntuples(res);
    if(n_list_entries > 0) {
        ip_in = PQgetvalue(res, 0, 0);
        iface_in = PQgetvalue(res, 0, 1);
    }
break;
default:
confd_trans_seterr(tctx, "invalid number of keys: %d", nkeys);
    return CONFD_ERR;
}

v = (confd_value_t *) malloc(sizeof(confd_value_t) * n_list_entries * navals);
o = malloc(sizeof(struct confd_next_object) * (n_list_entries + 2));

if (n_list_entries > 0) {
    for (i = 0; i < n_list_entries; i++) {
        obj[i].v = &v[i * navals];

        ip_in = PQgetvalue(res, i, 0);
        CONFDT_SET_STR(&(obj[i].v[0]), ip_in);
        iface_in = PQgetvalue(res, i, 1);
        CONFDT_SET_STR(&(obj[i].v[1]), iface_in);
        hwaddr = PQgetvalue(res, i, 2);
        CONFDT_SET_STR(&(obj[i].v[2]), hwaddr);
        if (strcmp(PQgetvalue(res, i, 3), "f") == 0)
           CONFDT_SET_STR(&(obj[i].v[3]), "0");
        else
           CONFDT_SET_STR(&(obj[i].v[3]), "1");

        if (strcmp(PQgetvalue(res, i, 4), "f") == 0)
           CONFDT_SET_STR(&(obj[i].v[4]), "0");
        else
           CONFDT_SET_STR(&(obj[i].v[4]), "1");

        obj[i].n = navals;
        obj[i].next = -1;
    }
    if (i == n_list_entries && n_list_entries < max_nobjs)
       obj[i++].v = NULL;
    confd_data_reply_next_object_arrays(tctx, obj, i, 0);
} else {
    confd_data_reply_next_object_array(tctx, NULL, -1, -1);
}
free(v);
free(obj);
PQclear(res);
return CONFD_OK;
}

int main(int argc, char *argv[])
{
    struct sockaddr_in addr;
    int debuglevel = CONFD_DEBUG;
    struct confd_trans_cbs trans;
    struct confd_data_cbs data;
struct confd_cs_node *object;
int c, msock;

while ((c = getopt(argc, argv, "dpts")) != -1) {
    switch(c) {
    case 'd':
        debuglevel = CONFD_DEBUG;
        break;
    case 'p':
        debuglevel = CONFD_PROTO_TRACE;
        break;
    case 't':
        debuglevel = CONFD_TRACE;
        break;
    case 's':
        debuglevel = CONFD_SILENT;
        break;
    }
}

/* connect to the postgresql db */
conn = PQconnectdb("dbname=confdoperdb user=postgres");
if (PQstatus(conn) != CONNECTION_OK) {
    perror(PQerrorMessage(conn));
}

memset(&trans, 0, sizeof(struct confd_trans_cbs));
trans.init = s_init;
trans.finish = s_finish;

memset(&data, 0, sizeof(struct confd_data_cbs));
data.get_next = get_next;
data.get_object = get_object;
data.num_instances = num_instances;
data.get_next_object = get_next_object;
data.find_next_object = find_next_object;

strcpy(data.callpoint, arpe_callpointid_arpe);

/* initialize confd library */
confd_init("arpe_daemon", stderr, debuglevel);

addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr.sin_family = AF_INET;
addr.sin_port = htons(CONFD_PORT);

if (confd_load_schemas((struct sockaddr*)&addr,
        sizeof (struct sockaddr_in)) != CONFD_OK)
    confd_fatal("Failed to load schemas from confd\n");
object = confd_cs_node_cd(NULL, "/arpe:arpentries/arpe");
navals = confd_max_object_size(object);

if ((dctx = confd_init_daemon("arpe_daemon")) == NULL)
confd_fatal("Failed to initialize confdlib\n");

confd_set_daemon_flags(dctx, CONFD_DAEMON_FLAG_STRINGSONLY);

/* Create the first control socket, all requests to */
/* create new transactions arrive here */

if ((ctlsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
confd_fatal("Failed to open ctlsocket\n");
if (confd_connect(dctx, ctlsock, CONTROL_SOCKET, (struct
sockaddr*)&addr,
    sizeof (struct sockaddr_in)) < 0)
confd_fatal("Failed to confd_connect() to confd");

/* Also establish a workersocket, this is the most simple */
/* case where we have just one ctlsock and one workersock */

if ((workersock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
confd_fatal("Failed to open workersocket\n");
if (confd_connect(dctx, workersock, WORKER_SOCKET,(struct
sockaddr*)&addr,
    sizeof (struct sockaddr_in)) < 0)
confd_fatal("Failed to confd_connect() to ConfD");

if (confd_register_trans_cb(dctx, &trans) == CONFD_ERR)
confd_fatal("Failed to register trans cb");

if (confd_register_data_cb(dctx, &data) == CONFD_ERR)
confd_fatal("Failed to register data cb");

if (confd_register_done(dctx) != CONFD_OK)
confd_fatal("Failed to complete registration");

if ((msock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
confd_fatal("Failed to connect MAAPI socket");

if (maapi_connect(msock, (struct sockaddr*)&addr, sizeof (struct
sockaddr_in)) < 0)
confd_fatal("Failed to connect maapi_connect() to ConfD");

maapi_start_phase(msock, 2, 0);
close(msock);

while(1) {
    struct pollfd set[2];
int ret;

set[0].fd = ctlsock;
set[0].events = POLLIN;
set[0].revents = 0;

set[1].fd = workersock;
set[1].events = POLLIN;
set[1].revents = 0;

if (poll(set, sizeof(set)/sizeof(*set), -1) < 0) {
    perror("Poll failed: ");
    continue;
}

/* Check for I/O */
if (set[0].revents & POLLIN) {
    if ((ret = confd_fd_ready(dctx, ctlsock)) == CONFD_EOF) {
        confd_fatal("Control socket closed\n");
    } else if (ret == CONFD_ERR && confd_errno != CONFD_ERR_EXTERNAL) {
        confd_fatal("Error on control socket request: %s (%d): %s\n",
                confd_strerror(confd_errno), confd_errno, confd_lasterr());
    }
}

if (set[1].revents & POLLIN) {
    if ((ret = confd_fd_ready(dctx, workersock)) == CONFD_EOF) {
        confd_fatal("Worker socket closed\n");
    } else if (ret == CONFD_ERR && confd_errno != CONFD_ERR_EXTERNAL) {
        confd_fatal("Error on worker socket request: %s (%d): %s\n",
                confd_strerror(confd_errno), confd_errno, confd_lasterr());
    }
}
/**************************************************************************/

The additional main() function initialization needed is just to create the connection. Then we are up and ready to serve any requests arriving from the northbound interfaces once they come up. By the way, we delay moving to ConfD’s start phase 2 where the northbound interfaces, e.g. NETCONF, are brought up until we have fully initialized our database connection and registered our callbacks with ConfD so that no requests will fail once ConfD is opened up for managers to request data. See the call to maapi_start_phase(msock,2,0) just before we enter our main loop.
You may also have noticed that we omitted implementing the get_elem() callback. If a manager requests a single leaf from a list, ConfD will just call the get_object() callback and get all five leaf entries in the list. The performance overhead associated with that is insignificant; so, why bother maintaining a get_elem() callback too?

The simplest callback to implement here is num_instances(). It just does a “SELECT COUNT(*) FROM arpstats” and we return that value as an int32 to ConfD. If you want an easy way to see it in action you can call cdb_num_instances(). The simplest way to do that from a terminal is to use the confd_cmd tool.

$ confd_cmd -dd -o -c ‘num_instances “/arpentries/arpe”’

The strategy used for passing the next row/list-entry from the get_next_object() variant we use here differs a bit from the one we used with the 5-c_stats example above for good reasons. We use an index number instead of pointers to our cached array when indicating to our next call where to continue from. We use “LIMIT” to get no more rows than the desired max chunk size which we want to pass to ConfD. We use OFFSET to continue from the row index indicated by the next variable. However, in the find_next_object() callback, we need to pass -1 back to ConfD for the next row/list-entry in the table unless we want to call Postgres again to find out which index our row is on. There is no need to do so because passing -1 will just result in another find_next_object() call to continue reading the list if there are more entries to be read. The overhead in comparison to jumping to get_next_object() and doing it from there is insignificant.

Not shown in the code above is how we create the database schema and populate the database similar to how the original ConfD 5-c_stats example reads the data from the ARP table to populate a linked list holding it. Here is a quick summary for the purpose of being able to follow the above example:

Drop the old table if it exists:

“DROP TABLE arpstats”

Create the new table:

“CREATE TABLE arpstats (ip inet, ifname text, hwaddr macaddr, permanent boolean, published boolean, PRIMARY KEY (ip, ifname))”

Insert the ARP table after parsing it similar to what is done in the 5-c_stats example:

“INSERT INTO arpstats VALUES ("<my_ip4_val>","<my_iface_val>","<my_hwaddr_val>",<my_perm_val>::BOOLEAN,<my_pub_val>::BOOLEAN)"

If you would rather to copy the contents from a file for test purposes, PostgreSQL comes with a quite convenient non-SQL standard COPY command that can be used similarly to how the maapi_load_config() function can be used:

“COPY arpstats FROM '/my/file'”
9. Apache Cassandra Database

In the Cassandra 3.11.2 based example integration code below, we use the ConfD Java API together with a pull mode application that performs queries when data is requested by ConfD using the Datastax Java Driver for Apache Cassandra. The reasoning for using Java here is that Cassandra is implemented in Java so we will provide a Java example. The example implements most of the relevant callback functions for the ARP table use case, including GET_ELEM, GET_NEXT, GET_OBJECT, GET_NEXT_OBJECT. See examples.confidd/intro/java/4-stats for a Java variant of the ARP table data provider example that this Cassandra integration example draws inspiration from.

```java
import com.tailf.conf.*;
import com.tailf.dp.Dp;
import com.tailf.dp.DpCallbackException;
import com.tailf.dp.DpTrans;
import com.tailf.dp.annotations.DataCallback;
import com.tailf.dp.proto.DataCBType;
import com.tailf.dp.proto.NextObjectArrayList;
import com.tailf.maapi.Maapi;
import org.apache.log4j.Logger;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.FileInputStream;
import java.net.Socket;
import java.util.*;
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;

public class ArpStat {
    private static final String HOST = "127.0.0.1";
    private static Logger log = Logger.getLogger(ArpStat.class);
    private static final String CONTACT_POINT = "127.0.0.1";
    private static final int PORT = 9042;
    private static Cluster cluster;
    private static Session session;

    public static class StatsCb {
        private ArrayList entries = new ArrayList();

        /**
         * @see DpDataCallback
         */
        @DataCallback(callPoint = arpe.callpoint_arpe,
                      callType = {DataCBType.ITERATOR})
        public Iterator<Object> iterator(DpTrans trans, ConfObject[])
```
kp)
    throws DpCallbackException {
    log.trace("==> iterator");
    ResultSet rs = session.execute("SELECT * FROM arp.arpstats;"巡查);
    log.trace("<== iterator");
    // iterate through the rows
    Row row;
    while((row = rs.one()) != null){
        Object obj = (Object) row;
        entries.add(obj);
    }
    return entries.iterator();
}

@DataCallback(callPoint = arpe.callpoint_arpe,
            callType = {DataCBType.GET_NEXT})
public ConfKey getKey(DpTrans trans, ConfObject[] kp, Object obj)
    throws DpCallbackException {
    log.trace("==> getKey");
    // convert iterator object (value) to a Row
    Row row = (Row) obj;
    ConfKey key = null;
    try {
        key = new ConfKey(new ConfObject[]{new ConfIPv4(row.getInet("ip").getHostAddress()),
                                           new ConfBuf(row.getString("ifname"))});
    } catch(ConfException e) {}  // catch exception
    log.trace("<== getKey");
    return key;
}

@DataCallback(callPoint = arpe.callpoint_arpe, callType = DataCBType.GET_NEXT_OBJECT)
public ConfValue[] getIteratorObject(DpTrans trans, ConfObject[] kp, Object obj) throws
    DpCallbackException {
    log.debug("==> getIteratorObject");
    Row row = (Row) obj;
    ConfValue[] retVal = new ConfValue[]{
        new ConfIPv4(row.getInet("ip")),
        new ConfBuf(row.getString("ifname")),
        new ConfBuf(row.getString("hwaddr")),
        new ConfBool(row.getBool("permanent")),
        new ConfBool(row.getBool("published"))
    };
    log.debug("<== getIteratorObject");
    return retVal;
}
@DataCallback(callPoint = arpe.callpoint_arpe, callType = DataCBType.GET_NEXT_OBJECT_LIST)
public List<ConfValue[]> getIteratorObjectList(DpTrans trans,
    ConfObject[] kp, 
    Object obj,
    Iterator<? extends Object> iterator)
    throws DpCallbackException {
    log.debug("==> getIteratorObject");

    // return MAX_ENTRIES, rest will be handled by next call
    final int MAX_ENTRIES = 100;
    Row row = (Row) obj;
    NextObjectArrayList retVal = new NextObjectArrayList<ConfValue[]>();
    ConfValue[] value = new ConfValue[]{
        new ConfIPv4(row.getInet("ip")), 
        new ConfBuf(row.getString("ifname")), 
        new ConfBuf(row.getString("hwaddr")), 
        new ConfBool(row.getBool("permanent")), 
        new ConfBool(row.getBool("published"))
    };
    retVal.add(value);
    int i = 0;
    while (iterator.hasNext() && i < MAX_ENTRIES - 1) {
        row = (Row) iterator.next();
        value = new ConfValue[]{
            new ConfIPv4(row.getInet("ip")), 
            new ConfBuf(row.getString("ifname")), 
            new ConfBuf(row.getString("hwaddr")), 
            new ConfBool(row.getBool("permanent")), 
            new ConfBool(row.getBool("published"))
        };
        retVal.add(value);
        i++;
    }
    log.debug("<== getIteratorObject  retVal.size()=" + retVal.size());
    return retVal;
}

@DataCallback(callPoint = arpe.callpoint_arpe, callType = DataCBType.GET_OBJECT)
public ConfObject[] getObject(DpTrans trans, ConfObject[] kp)
    throws DpCallbackException {
    log.trace("<== getObject");
String ip = ((ConfKey) kp[0]).elementAt(0).toString();
String ifname = ((ConfKey) kp[0]).elementAt(1).toString();
ResultSet rs = session.execute("SELECT * FROM arp.arpstats
WHERE ip = "+ip+" AND ifname = "+ifname+";");
Row row = rs.one();
ConfObject[] retVal = null;
try {
    retVal = new ConfValue[]{
        new ConfIPv4(ip),
        new ConfBuf(ifname),
        new ConfBuf(row.getString("hwaddr")),
        new ConfBool(row.getBool("permanent")),
        new ConfBool(row.getBool("published"))
    };
} catch(ConfException e){}
log.trace("==> getObject");
return retVal;

@DataCallback(callPoint = arpe.callpoint _ arpe,
callType = {DataCBType.GET _ELEM})
public ConfValue getElem(DpTrans trans, ConfObject[] kp) {
    log.trace("==> getElem");
    String ip = ((ConfKey) kp[1]).elementAt(0).toString();
    String ifname = ((ConfKey) kp[1]).elementAt(1).toString();
    ResultSet rs = session.execute("SELECT * FROM arp.arpstats
WHERE ip = "+ip+" AND ifname = "+ifname+";");
    Row row = rs.one();
    ConfValue retVal = null;
    ConfTag leaf = (ConfTag) kp[0];
    switch (leaf.getTagHash()) {
        case arpe.arpe _ ip:
            try {
                retVal = new ConfIPv4(ip);
            } catch(ConfException e){}
            break;
        case arpe.arpe _ ifname:
            retVal = new ConfBuf(ifname);
            break;
        case arpe.arpe _ hwaddr:
            retVal = new ConfBuf(row.getString("hwaddr"));
            break;
        case arpe.arpe _ permanent:
            try {
                retVal = new ConfBool(row.getBool("permanent"));
            } catch(ConfException e){}
            break;
        case arpe.arpe _ published:
            try {
                retVal = new ConfBool(row.getBool("published"));
            } catch(ConfException e){}
            break;
    }
```java
retVal = new ConfBool(row.getString("published"));
} catch(ConfException e){
    break;
    default:
    break;
}
log.trace("== getElem");
return retVal;
}

/**
 * Initiates a connection to the cluster
 * specified by the given contact point.
 * @param contactPoint the contact point to use.
 * @param port          the port to use.
 */
public void connect(String contactPoint, int port) {
    cluster = Cluster.builder().addContactPoint(contactPoint).
    withPort(port).build();
    System.out.println("Connected to cluster: "+cluster.
    getMetadata().getClusterName());
    session = cluster.connect();
}

/**
 * Closes the session and the cluster.
 */
public void close() {
    session.close();
    cluster.close();
}

static public void main(String args[]) throws Exception {
    log.trace("==> main");
    // create new control socket
    Socket ctrlSocket = new Socket(HOST, Conf.PORT);
    // init and connect control socket
    Dp dp = new Dp("arpstats", ctrlSocket);
    // register the stats callbacks
    dp.registerAnnotatedCallbacks(new StatsCb());
    /* register the validation callbacks and valpoints */
    dp.registerDone();

    ArpStat client = new ArpStat();
    try {
        client.connect(CONTACT_POINT, PORT);
```
Socket s = new Socket("localhost", Conf.PORT);
Maapi maapi = new Maapi(s);
maapi.startPhase(2);
s.close();

log.trace("ArpStat started");
// read input from the control socket
try {
    while (true) {
        dp.read();
    }
} catch (Exception e) {
    log.error(e);
    log.trace("<== main");
    client.close();
}

The Cassandra integration is quite similar to the one that we did for the Postgres example above except for using the Java API instead of a C API. One reason for similarity is that the Cassandra Query Language (CQL) is purposefully similar to Structured Query Language (SQL) used in relational databases like Postgres. Many queries are very similar between the two. In fact, a lot of basic things are exactly the same, as you probably already noticed when comparing the two external operational state DB integration examples.

Not shown in the code above is how we create the database schema and populate the database similar to how the original ConfD 5-c_stats example reads the data from the ARP table to populate a linked list holding it. A quick summary below is for the purpose of being able to follow the above example. The example below does not read the ARP cache in the system, it just populates Cassandra from a file for test purposes.

```java
public class ArpDbInit {
    private static Logger log = Logger.getLogger(ArpDbInit.class);
    static String[] CONTACT_POINTS = {"127.0.0.1"};
    static int PORT = 9042;

    public static void main(String[] args) {

        ArpDbInit client = new ArpDbInit();

        try {
            client.connect(CONTACT_POINTS, PORT);
            client.createSchema();
        }
    }
}
```
client.populateEntries();
} catch (Exception e) {
    System.out.println("Exception "+e);
} finally {
    //System.out.println("close");
    client.close();
}

private Cluster cluster;
private Session session;

/**
 * Initiates a connection to the cluster
 * specified by the given contact point.
 * @param contactPoints the contact points to use.
 * @param port the port to use.
 */
public void connect(String[] contactPoints, int port) {
    cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
    System.out.println("Connected to cluster: "+cluster.
        getMetadata().getClusterName());
    session = cluster.connect();
}

/**
 * Creates the schema (keyspace) and tables
 * for this example.
 */
public void createSchema() {
    session.execute("CREATE KEYSPACE IF NOT EXISTS arp WITH
        replication " +
        "= {'class':'SimpleStrategy', 'replication_factor':1};");
    session.execute("DROP TABLE IF EXISTS arp.arpstats");
    session.execute("CREATE TABLE IF NOT EXISTS arp.arpstats (" +
        "ip inet," +
        "ifname text," +
        "hwaddr text," +
        "permanent boolean," +
        "published boolean," +
        "PRIMARY KEY (ip, ifname)" +
        ");");
}

/**
 * Inserts data into the tables.

public void loadData() {
    session.execute(
        "INSERT INTO arp.arpstats (ip, ifname, hwaddr, permanent, published) " +
        "VALUES (" +
        "'10.0.0.1'," +
        "'en1111111'," +
        "'00:00:00:00:00:00'," +
        "false," +
        "false)" +
        ",");
}

/**
* Queries and displays data.
*/
public void querySchema() {

    ResultSet results = session.execute("SELECT * FROM arp.arpstats;");

    System.out.printf("%-21s%-21s%-21s%-21s%-21s%n", "ip", "ifname", "hwaddr", "permanent", "published");
    System.out.println("-------------------+-------------------+-------------------+-------------------+-------------------+");
    for (Row row : results) {
        System.out.printf("%-21s%-21s%-21s%-21s%-21s%n", row.getInet("ip").getHostAddress(), row.getString("ifname"), row.getString("hwaddr"), row.getBool("permanent"), row.getBool("published"));
    }
}

/**
* Closes the session and the cluster.
*/
public void close() {
    session.close();
    cluster.close();
}

private void populateEntries() {
    log.trace("==> populateEntries");
    try {
        
    } catch (Exception e) {
        // Handle exceptions
    }
}
String s;
FileInputStream fstream;
fstream = new FileInputStream("/tmp/test");
BufferedReader br = new BufferedReader(new InputStreamReader(fstream));
while ((s = br.readLine()) != null) {
    processArpLine(s);
}
br.close();
} catch (IOException e) {
    log.error(e);
}
log.trace("<== populateEntries");
}

private void processArpLine(String s) {
    String[] toks = s.split(",");
    if (toks.length == 5) {
        session.execute("INSERT INTO arp.arpstats (ip, ifname, hwaddr, permanent, published) VALUES (" +
                        "+" +
                        "+" +
                        "+" +
                        "+" +
                        ");
    }
}

for (Row row : results) {
    System.out.printf("%-21s%-21s%-21s%-21s%-21s\n", row.getInet("ip").getHostAddress(), row.getString("ifname"), row.getString("hwaddr"), row.getBool("permanent"), row.getBool("published"));
}

/**

ln("--------------------+--------------------+--------------------+-----------------------------+---------------------+
---------+--------------------+
")

ln("-----------------------------+-----------------------------+-----------------------------+-----------------------------+
-----------------------------+-----------------------------+
")

for (Row row : results) {
    System.out.printf("%-21s%-21s%-21s%-21s%-21s\n", row.getInet("ip").getHostAddress(), row.getString("ifname"), row.getString("hwaddr"), row.getBool("permanent"), row.getBool("published"));
}

*/
* Closes the session and the cluster. */
public void close() {
    session.close();
    cluster.close();
}

private void populateEntries() {
    log.trace("==> populateEntries");
    try {
        String s;
        FileInputStream fstream;
        fstream = new FileInputStream("/tmp/test");
        BufferedReader br = new BufferedReader(new InputStreamReader(fstream));
        while ((s = br.readLine()) != null) {
            processArpLine(s);
        }
        br.close();
    } catch (IOException e) {
        log.error(e);
    }
    log.trace("<== populateEntries");
}

private void processArpLine(String s) {
    String[] toks = s.split",";
    if (toks.length == 5) {
    }
}
10. Summary

There can be great benefits for applications to integrate a database externally to ConfD for the purpose of storing operational state data. Example scenarios include storing large amounts of state data, replicating this data in best effort becoming eventually consistent, and doing joins and indexing of large amounts of data that is displayed in real time where you have some state data that you also want to display over, for example, NETCONF or RESTCONF without having to replicate it into ConfD’s CDB operational datastore. We began by showing how the YANG model becomes the ConfD CDB database schema and then moved on to how you can do different types of filter queries to get both state and configuration data that you need using NETCONF as an example. We demonstrated how a data provider application can, with the addition of a couple of powerful callbacks, retrieve large tables (YANG lists) from a simple cache or file or from well-established databases such as PostgreSQL and Apache Cassandra. We also demonstrated a second type of data provider application that feeds the accumulated data into the ConfD CDB operational datastore using equally powerful set operations where the table/list is pushed into CDB in large but configurable chunks. The goal has been to provide guidance into how to utilize the many features which ConfD provides for various different use cases where operational state data is provided over northbound management interfaces such as NETCONF, RESTCONF, etc.

For More Information

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

Application note: “Inside the ConfD Operational Data Provider API”