CHANGES IN YANG LEAF-LIST HANDLING

tail-f

Table of Contents

Introduction

One As of ConfD 6.5, the internal representation of leaf-lists has changed and this impacts how a ConfD developer handles leaf-lists in their application code. In this application note, we will discuss the changes in how leaf-list nodes are processed and what operations are affected for both configuration and operational leaf-list elements.

Leaf-lists now behave similar to lists in terms of operations, i.e., individual elements can now be created and deleted instead of setting/deleting the entire leaf-list as a single entity as in ConfD 6.4 and prior. Attributes can now be set on individual leaf-list elements. This new way of handling leaf-list operations in the backend makes it more intuitive to work with this type of node.

Temporary backward compatibility flags have been introduced, for convenience, to maintain the old behavior of the MAAPI, CDB and DP APIs. These flags are planned to be removed in ConfD 6.8. If you use the backward compatibility flags, you should plan to eliminate use of them and re-write your code to use the new operations by the time you upgrade to ConfD 6.8.

The backward compatibility flags apply to MAAPI, CDB API and DP API when used as an operational data provider, external configuration data provider and transformation callpoint handler. The backward compatibility flags cannot be used in set or transaction hook code, or in validation code.

We will talk about the new backward compatibility flags in the next section, and whenever they are relevant, in each section.

The new way of handling leaf-lists affects all of the language APIs:

- C
- Java
- Python
- Erlang
- JSON-RPC

This paper is organized in the following fashion:

- Backward Compatibility Flags
- CDB Subscribers handling of leaf-lists
- Data Providers for operational leaf-lists
- Data Providers for Configuration leaf-lists (Configuration stored outside of CDB)
- Validation Callbacks
- Transformations
- Hooks

Example C code is provided for ConfD 6.4, to show how a leaf-list was handled prior to the internal implementation change, and for ConfD 6.5, to illustrate how you will need to change your code to handle the new leaf-list behavior, for comparison.

Backward Compatibility Flags

Backward compatibility flags have been introduced which can be used to maintain some of the old behavior. These flags were introduced in 6.5 and are planned to be removed in 6.8. The goal is to help developers maintain the old behavior until they can rewrite their code for the new way of handling leaf-lists.

The differences between ConfD 6.4 and ConfD 6.5 can be summarized as: Handling of leaf-list nodes in ConfD 6.4 is similar to handling leaf nodes. In ConfD 6.5, it is similar to handling lists.

1-**ITER_WANT_LEAF_LIST_AS_LEAF:** This flag can be used in any of the iteration functions like cdb_diff_iterate() and maapi_diff_iterate(). Using this flag tells ConfD to report SET operations on leaf-lists instead of CREATE/DELETE operations, which is now the default as of ConfD 6.5.

2- **CONFD_DAEMON_FLAG_LEAF_LIST_AS_LEAF:** This flag declares that data provider and transform callbacks will treat leaf-lists as leafs and not as lists, e.g. use get_elem()/set_elem() rather than get_next()/create() .

When and where these flags can be used and what their limitations are, is discussed in the following sections.

CDB Subscriber Handling of leaf-lists

As of ConfD 6.5, individual elements in a leaf-list can be created and deleted in addition to the setting and deletion of the entire leaf-list as a single entity. The ability to still be able to get and set entire leaf-list nodes is considered a convenience for backward compatibility. The handling of a set and get, for the entire leaf-list content, should be changed in the long run.

A backward compatibility flag was also added in 6.5, which allows applications to still handle leaf-lists as they used to in ConfD 6.5. The flag is ITER_WANT_LEAF_LIST_AS_LEAF. It can be used in any of the iteration functions like cdb_diff_iterate() and maapi_diff_iterate(). Using this flag allows ConfD to report SET operations on leaf-lists instead of CREATE/DELETE operations which is the default in ConfD 6.5. So, if you use this flag, the code relative to ConfD 6.4, presented below, will still work in ConfD 6.5. However, this flag will be removed in a future release of ConfD and should only be used on a temporary basis until you rewrite your application code to use the new method of processing.

The CDB API provides an API called cdb_diff_iterate() which allows a CDB subscriber to iterate over configuration changes in a transaction. Iterating over a change set, in ConfD 6.5, for leaf-lists has a different semantic. “diff_iterate” will report create/delete operations for leaf-list elements instead of set/delete for the whole leaf-list.

In this section and the following sections, we will be using the following YANG module as an example, unless stated otherwise:

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

  import tailf-common {
    prefix tailf;
  }

  container root {
    leaf-list config {
      tailf:validate valPoint {
        tailf:dependency .;
      }
      type uint8;
    }
    leaf-list oper {
        config false;
        tailf:callpoint operCP;
        type uint8;
    }
  }
}
```

In this example, we have a subscriber using cdb_diff_iterate() to iterate over the configuration changes.

```
cdb_diff_iterate(subsock, sub_points[0], iter, ITER_WANT_PREV, &sock);
```

We will be using the following skeleton of the iter() callback to illustrate the example code:

```
static enum cdb_iter_ret iter(confd_hkeypath_t *kp, enum cdb_iter_op op,
confd_value_t *oldv, confd_value_t *newv,void *state)
{
    char buf[BUFSIZ];
    confd_pp_kpath(buf, BUFSIZ, kp);
    switch (op) {
        case MOP_CREATED: {
                    /* code */
       }
       break;
       case MOP_DELETED: {
                    /* code */
       }
       break;
       case MOP_MODIFIED: {
                    /* code */
       }
       break;
       case MOP_VALUE_SET: {
                    /* code */
       }
       break;
         default:
             fprintf(stderr, "Unexpected op %d for %s\n", op, buf);
             break;
    }
    return ITER_RECURSE;
}
```

In the YANG module root.yang, config is a configuration leaf-list node of type uint8.

In our processing of the configuration change, we may be reading the value directly from CDB, or reading the value passed to the iter() callback, “newv”.

If we are reading from CDB using the CDB API, cdb_get_list(), the example code below works in both versions, 6.4 and 6.5:

```
static void read_db(int cdbsock)
{
    int ret, i, n;

    if ((ret = cdb_start_session(cdbsock, CDB_RUNNING)) != CONFD_OK)
        confd_fatal("Cannot start session\n");
    if ((ret = cdb_set_namespace(cdbsock, root__ns)) != CONFD_OK)
        confd_fatal("Cnnot set namespace\n");

    if (cdb_exists(cdbsock, "/root/config")) {
        confd_value_t *myList_v;
        cdb_get_list(cdbsock, &myList_v, &n, "/root/config");
        for (i = 0; i < n; i++) {
            fprintf(stderr, "config[%d: %d\n", i, CONFD_GET_
UINT8(&myList_v[i]));
            confd_free_value(&myList_v[i]);
        }
        free(myList_v);
    }

    cdb_end_session(cdbsock);

}
```

Note that cdb_get_list() will return the whole leaf-list value.

The table below describes the different behavior in the iter() callback op argument between ConfD 6.4 and ConfD 6.5.

Operation	ConfD-6.4 behavior	ConfD-6.5 behavior
Set	Op = MOP_VALUE_SET	Op = MOP_CREATED
Delete an instance from the leaf-list resulting in a non-empty leaf-list	Op = MOP_VALUE_SET	Op = MOP_DELETED at myList’s instance level
Delete the whole leaf-list	Op = MOP_DELETED (One call at myList’s level)	Op = MOP_DELETED (Several MOP_DELETED operations corresponding to the number of instances in config.

Example code for handling the different operations:

ConfD 6.4

```
static enum cdb_iter_ret iter(confd_hkeypath_t *kp, enum cdb_iter_op op,
                              confd_value_t *oldv,
                              confd_value_t *newv,
                              void *state)
{
    char buf[BUFSIZ];
    confd_pp_kpath(buf, BUFSIZ, kp);
    switch (op) {
        case MOP_CREATED:
            fprintf(stderr, "MOP_CREATED Operation: %s\n", buf);
        break;
        case MOP_DELETED:
            fprintf(stderr, "MOP_DELETED Operation: %s\n", buf);
            confd_value_t *dtag = NULL;
            dtag = &kp->v[0][0];
            switch (CONFD_GET_XMLTAG(dtag)) {
                case root_config:
                    /* myList was deleted */
                    fprintf(stderr, "\nProcessing of the iter MOP_DELETE for
%s\n", buf);
                break;
            }
            return ITER_CONTINUE;
        break;
        case MOP_MODIFIED:
            fprintf(stderr, "MOP_MODIFIED Operation: %s\n", buf);
        break;
        case MOP_VALUE_SET: {
            char nbuf[BUFSIZ];
            confd_pp_value(nbuf, BUFSIZ, newv);
            fprintf(stderr, "MOP_VALUE_SET Operation: %s --> (%s)\n",
                    buf, nbuf);
            confd_value_t *leaf = &kp->v[0][0];
            switch (CONFD_GET_XMLTAG(leaf)) {
                case root_config: {
                    /* keypath is /root/config  */
                    /*              1       0   */

                    /* copy in the new value */
                    confd_value_t *vp = CONFD_GET_LIST(newv);
                    int size = CONFD_GET_LISTSIZE(newv);

                    for (int i = 0; i < size; i++) {
                        fprintf(stderr, "myList %d: %d\n", i, CONFD_GET_
UINT8(&vp[i]));
                    }
                    return ITER_CONTINUE;
                }
                break;
                default:
                    fprintf(stderr, "Unexpected path: %s\n", buf);
                break;
            }
        }
        break;
        default:
            fprintf(stderr, "Unexpected op %d for %s\n", op, buf);
            break;
    }
    return ITER_RECURSE;
}
```

ConfD 6.5

```
static enum cdb_iter_ret iter(confd_hkeypath_t *kp,
                              enum cdb_iter_op op,
                              confd_value_t *oldv,
                              confd_value_t *newv,
                              void *state)
{
    char buf[BUFSIZ];
    confd_pp_kpath(buf, BUFSIZ, kp);
    switch (op) {
        case MOP_CREATED: {

            fprintf(stderr, "MOP_CREATED Operation: %s\n", buf);
            confd_value_t *ctag = &kp->v[1][0];

            return ITER_CONTINUE;
        }
        break;
        case MOP_DELETED: {
            /* Read the leaf-list value being deleted */
            confd_value_t *listkey = &kp->v[0][0];
            fprintf(stderr, "MOP_DELETED Operation: %s\n", buf);
            fprintf(stderr, "Deleting leaf-list %d\n", CONFD_GET_
UINT8(listkey));


            return ITER_CONTINUE;
        }
        break;
        case MOP_MODIFIED:
            /* No leaf-list handling here */

        break;
        case MOP_VALUE_SET: {
            /* No leaf-list handling here */

        }
        break;
        default:
            /* We should never get MOP_MOVED_AFTER or MOP_ATTR_SET */
            fprintf(stderr, "Unexpected op %d for %s\n", op, buf);
            break;
    }
    return ITER_RECURSE;
}
```

Data Providers for Operational leaf-lists

In ConfD 6.5, leaf-list operational nodes require a get_next() callback registration. In ConfD 6.4 leaf-list operational nodes are provided just like any other "config false" leaf element, using get_elem().

A backward compatibility flag for Data Providers was introduced in ConfD 6.5: CONFD_DAEMON_FLAG_LEAF_LIST_AS_LEAF. This flag allows a Data Provider running in ConfD 6.5, to register a get_elem() callback instead of get_next(), thus, maintaining the previous behavior in ConfD 6.4. For Configuration Data Providers, this flag allows a set_elem() callback instead of create(). However, this flag will be removed in a future release of ConfD and should only be used on a temporary basis until you rewrite your application code to use the new method of processing.

This flag is not supported for hooks, i.e. hook callbacks will always treat leaf-lists as lists -

if the flag is set for a hook, it will cause an error on callback invocation for a leaf-list.

Mandatory callbacks needed to register with ConfD, for a leaf-list, via the Data Provider API:

ConfD 6.4	ConfD 6.5
get_elem() or get_object()/get_next_object() for a parent list	get_next()

Example code:

ConfD 6.4

```
/*      Keypath example         */
/*      /root/oper              */
/*      1     0                 */

static int get_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)
{
    confd_value_t v;

    switch (CONFD_GET_XMLTAG(&(keypath->v[0][0]))) {
        case root_oper:{
            confd_value_t arr[5];
            int i;
            for (i=0; i<5; i++)
                CONFD_SET_UINT8(&arr[i], i);
            CONFD_SET_LIST(&v, &arr[0], 5);
        }
        break;
        default:
            return CONFD_ERR;
    }
    confd_data_reply_value(tctx, &v);
    return CONFD_OK;
}
```

ConfD 6.5

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

    if (next < 4) {
        CONFD_SET_UINT8(&v, ++next);
        confd_data_reply_next_key(tctx, &v, 1, next);
    } else {
        confd_data_reply_next_key(tctx, NULL, -1, -1);
    }

    return CONFD_OK;
}
```

If the backward compatibility flag is used, ConfD 6.5 will require a get_elem() callback registration. The code for ConfD 6.4 will work perfectly in this case.

To use the flag, one must call confd_set_daemon_flags() right after confd_init_daemon().

Example:

```
    if ((dctx = confd_init_daemon("LeafListSubscriber")) == NULL)
        confd_fatal("Failed to initialize confd\n");

    /* Set the flag */
    confd_set_daemon_flags(dctx, CONFD_DAEMON_FLAG_LEAF_LIST_AS_LEAF);
```

Validation Callbacks

In this example, we will show how to validate a leaf-list change. The validation point is set on the leaf-list itself. Refer to the YANG module "root.yang" above.

The backward compatibility flag mentioned in the previous section: CONFD_DAEMON_FLAG_LEAF_LIST_AS_LEAF, has no effect for validation callbacks.

However, just as for a list, a tailf:validate statement for a leaf-list may use a tailf:call-once substatement to request a single invocation for validation of the whole leaf-list, thus, maintaining the old behavior of getting called once to validate the whole leaf-list content, instead of getting a callback for each entry in the leaf-list. The code can then read the entire leaf-list using maapi_get_elem(), or check for the existence of specific entries using maapi_exists().

ConfD 6.4

```
static int validate(struct confd_trans_ctx *tctx,

                    confd_hkeypath_t *keypath,
                    confd_value_t *newval)
{
    char buf[BUFSIZ];
    confd_pp_kpath(buf, BUFSIZ, keypath);

    fprintf(stderr, "Validation Keypath: %s \n", buf);

    confd_value_t *vp = CONFD_GET_LIST(newval);
    int size = CONFD_GET_LISTSIZE(newval);

    for (int i = 0; i < size; i++) {
        fprintf(stderr, "Validating myList %d: %d\n", i, CONFD_GET_
UINT8(&vp[i]));
    }

    return CONFD_OK;
}
```

Here the keypath will always be: /root/config

The validation callback is called to validate the remaining configuration, not to validate delete operations.

If we delete an element within the leaf-list, the validation callback will only be called if there are remaining elements in the leaf-list. It will not be called if the resulting value of the leaf-list after the delete doesn't exist. This means that we validate configuration, not operations, as is normal for ConfD validation callbacks. If a use case requires validating deletion of a whole leaf-list, a validation point will be needed at a parent node level. In this case, when a deletion occurs, the validation callback will be called since the parent node configuration changed.

ConfD 6.5

In ConfD 6.5, the validation callback of a leaf-list will be called as many times as the number of elements in the leaf-list. For example: If config = [1 2 3], the validation callback will be called 3 times, to validate each element.

With the YANG module provided above, the validation point is at the leaf-list level. In this case, deleting an instance in the leaf-list will not trigger the validation callback. The validation callback only gets triggered for 'set' operations.

Again, if triggering the validation callback for deletions is a requirement, the validation point would need to be set at a parent level.

Validation code example for ConfD 6.5:

```
static int validate(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath,
                    confd_value_t *newval)
{
    char buf[BUFSIZ];
    confd_pp_kpath(buf, BUFSIZ, keypath);

    fprintf(stderr, "Validation Keypath: %s \n", buf);
    fprintf(stderr, "Validating myList instance =  %d\n", CONFD_GET_
UINT8(newval));

    /* Process the validation logic here based on the newval value */
    /* This validation is only triggered for a Set operation */

    return CONFD_OK;
}
```

In summary:

Note: Here, the Tail-f extension "tailf:call-once true" is not used. If we use it, validation will be triggered the same way for both ConfD 6.4 and 6.5, including a call when the whole list is deleted.

Operation	ConfD-6.4 leaf-list validation	ConfD-6.5 leaf-list validation
Set	Validation CB called with keypath: /root/config	Validation CB is called with keypath: /root/config{<value> for each value in the leaf-list.
Delete a leaf-list instance with size>2, example myList = [1 2]	Validation CB called with keypath: /root/config	Validation CB not called.
Delete a leaf-list instance with size =1, example myList = [1], or delete the whole leaf-list.	Validation CB not called.	Validation CB not called.

Data Providers for Configuration leaf-lists

When configuration is stored outside of CDB, a Data Provider is required to register with ConfD for handling that data. For leaf-lists, the table below illustrates the callbacks needed for each operation in ConfD 6.4 and ConfD 6.5.

The backward compatibility flag CONFD_DAEMON_FLAG_LEAF_LIST_AS_LEAF, can be used here to maintain the old behavior in ConfD 6.5 as well. However, this flag will be removed in a future release of ConfD and should only be used on a temporary basis until you rewrite your application code to use the new method of processing.

Operation	ConfD 6.4	ConfD 6.5
set	set_elem() (Called when a new instance is added and for the whole new leaf-list, called also when an instance is deleted)	create() (Called for each newly added leaf-list instance)
get	get_elem() (Called for the whole leaf-list to be returned)	get_next() (Called for each instance in the leaf-list)
delete	remove() (Only called when the whole leaf-list is deleted)	remove() (called for each deleted instance in the leaf-list)

Example YANG module:

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

  import tailf-common {
    prefix tailf;
  }

  container root {
    leaf-list config {
      tailf:callpoint extConfigCP;
      tailf:validate valPoint {
        tailf:dependency .;
      }
      type uint8;
    }
    leaf-list oper {
        config false;
        tailf:callpoint operCP;
        type uint8;
    }
  }
}
```

Note the "tailf:callpoint extConfigCP;" statement here. This indicates to ConfD that the config element will be stored outside of CDB.

Callbacks implementation in ConfD 6.4:

```
static int get_config_elem(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)
{
    confd_value_t v;
    confd_value_t arr[MAXCFGELEMS];
    int i=0;
    myConfig_t *tmp = myConfig;

    while(tmp!=NULL) {
        CONFD_SET_UINT8(&arr[i], tmp->val);
        tmp = tmp->next;
        i++;
    }
    CONFD_SET_LIST(&v, &arr[0], i);

    confd_data_reply_value(tctx, &v);
    return CONFD_OK;
}

static int set_config_elem(struct confd_trans_ctx *tctx,
                       confd_hkeypath_t *keypath,
                       confd_value_t *newval)

{
    char nbuf[BUFSIZ];
    confd_pp_value(nbuf, BUFSIZ, newval);

    fprintf(stderr, "Setting leaf-list config %s\n", nbuf);

    /* Update the leaf-list value in the backend */
    confd_value_t *vp = CONFD_GET_LIST(newval);
    int size = CONFD_GET_LISTSIZE(newval);
```

continued on next page...

```
    myConfig = NULL;
    for (int i = 0; i<size && i<MAXCFGELEMS; i++) {
        fprintf(stderr, "Setting list element %d: %d\n", i, CONFD_GET_
UINT8(&vp[i]));
        insert_val(CONFD_GET_UINT8(&vp[i]));
    }

    return CONFD_OK;
}

static int remove_config(struct confd_trans_ctx *tctx,
                    confd_hkeypath_t *keypath)
{
    char buf[BUFSIZ];
    confd_pp_kpath(buf, BUFSIZ, keypath);

    fprintf(stderr, "Remove the list from backend: %s\n", buf);

    free(myConfig);
    myConfig = NULL;

    return CONFD_OK;
}
```

Callbacks implementation in ConfD 6.5:

In ConfD 6.5, the following Data Provider callbacks are mandatory for handling configuration leaf-lists stored outside of CDB:

```
static int get_config_next(struct confd_trans_ctx *tctx,
                        confd_hkeypath_t *keypath,
                        long next)
{
    confd_value_t v;

    if(myConfig == NULL) {
        confd_data_reply_next_key(tctx, NULL, -1, -1);
        return CONFD_OK;
    }
    if (next == -1) {
        CONFD_SET_UINT8(&v, myConfig->val);
        confd_data_reply_next_key(tctx, &v, 1, myConfig->val);
    } else {
        uint8_t nextVal;
        if(find_next_val((uint8_t)next,&nextVal) == 0)
            confd_data_reply_next_key(tctx, NULL, -1, -1);
        else {
            CONFD_SET_UINT8(&v, nextVal);
            confd_data_reply_next_key(tctx, &v, 1,(long)nextVal);
        }
    }

    return CONFD_OK;
}

static int config_exists_optional (struct confd_trans_ctx *tctx,
                  confd_hkeypath_t *keypath)
{
    if(find_myConfig(CONFD_GET_UINT8(&keypath->v[0][0])))
        confd_data_reply_found(tctx);
    else
       confd_data_reply_not_found(tctx);
```

continued on next page...

```
    return CONFD_OK;
}

/*      Keypath example          */
/*      /root/config{val}               */
/*      2       1     0                   */

static int create_config(struct confd_trans_ctx *tctx,
                         confd_hkeypath_t *keypath)

{
    uint8_t val = CONFD_GET_UINT8(&keypath->v[0][0]);
    fprintf(stderr, "Creating leaf-list instance %d\n", val);

    insert_val(val);

    return CONFD_OK;
}

/*      Keypath example          */
/*      /root/config{val}               */
/*       2       1     0                  */

static int remove_config(struct confd_trans_ctx *tctx,
                     confd_hkeypath_t *keypath)
{
    uint8_t val = CONFD_GET_UINT8(&keypath->v[0][0]);

    fprintf(stderr, "Remove the list instance %d from backend\n", val);

    delete_val(val);

    return CONFD_OK;
}
```

Transformations

Transformations are used in ConfD when you want to present a different view of some configuration data that has already been modeled in YANG. For example, you may already support interface configuration in your YANG model, but you also want to give your customer the ability to use a standard YANG model, such as one from the IETF to configure interfaces. For this you would use a transformation. A transformation allows the customer to use either the original model used to represent the configuration or the new model that maps to the original model.

Transformation code is very similar to the Data Provider for Configuration example discussed above. The difference is that in your implementation of the data callbacks, you invoked MAAPI functions to handle the values in the original model.

Operation	ConfD 6.4	ConfD 6.5
set	set_elem() (Called when a new instance is added and for the whole new leaf-list, called also when an instance is deleted)	create() (Called for each newly added leaf-list instance)
get	get_elem() (Called for the whole leaf-list to be returned)	get_next() (Called for each instance in the leaf-list)
delete	remove() (Only called when the whole leaf-list is deleted)	remove() (called for each deleted instance in the leaf-list)

For transformations, we have the original model (or the “to-model”):

```
module to-model {
    namespace “http://tail-f.com/ns/example/to-model”;
    prefix to;

    import tailf-common {
        prefix tailf;
    }

    container foo {
      container bar {
        leaf-list baz {
            type string;
        }
      }
    }
}
```

The new model we want to transform (or the “from-model”):

```
module from-model {
    namespace “http://tail-f.com/ns/example/from-model”;
    prefix from;

    import tailf-common {
        prefix tailf;
    }

    container foo {
        tailf:callpoint trans_foo {
            tailf:transform true;
        }
        leaf-list bar {
            type string;
        }
    }
}
```

Callbacks implementation in ConfD 6.4

Prior to the leaf-list implementation change, i.e. ConfD 6.4 and earlier, leaf-lists were handled like any other leaf element in ConfD. You would implement the get_elem, set_elem, and remove_elem callback functions in the Data Provider API.

```
static int cb_get_elem(struct confd_trans_ctx *tctx, confd_hkeypath_t *kp)
{
    TRACE_ENTER("");
    print_path("GET_ELEM() request keypath", kp);

    confd_value_t v_result;
    CONFD_SET_NOEXISTS(&v_result);

    int ret = CONFD_OK;
    int maapi_ret = CONFD_OK;
    maapi_ret = maapi_get_elem(maapisock, tctx->thandle, &v_result, "/to:foo/
bar/baz", &v_result)
;
    INFO("maapi_get_elem ret is %d", ret);
    if (CONFD_OK == maapi_ret) {
        if (C_NOEXISTS != v_result.type) {
            confd_data_reply_value(tctx, &v_result);
        } else {
            confd_data_reply_not_found(tctx);
        }
    } else {
        confd_data_reply_not_found(tctx);
    }
    TRACE_EXIT("(%d)", ret);
    return ret;
}

static int cb_set_elem(struct confd_trans_ctx *tctx,
        confd_hkeypath_t *keypath, confd_value_t *newval)
{
    DEBUG_ENTER("");
    int ret = CONFD_OK;
    trace_confd_kp("keypath=", keypath);
    trace_confd_val("newval=", newval);
    INFO("cb_set_elem: Set elem called.");

    ret = maapi_set_elem(maapisock, tctx->thandle, newval, "/to:foo/bar/
baz");

    DEBUG_EXIT("ret=%i", ret);
    return ret;
}

static int cb_remove(struct confd_trans_ctx *tctx,
        confd_hkeypath_t *keypath)
{
    DEBUG_ENTER("");
    int ret = CONFD_OK;
    trace_confd_kp("keypath=", keypath);

    INFO("cb_remove called");

    ret = maapi_delete(maapisock, tctx->thandle, "/to:foo/bar/baz");

    DEBUG_EXIT("ret=%i", ret);
    return ret;
}
```

Callbacks implementation in ConfD 6.5

As of ConfD 6.5, the following Data Provider callbacks are mandatory for handling tranformations involving leaf-lists:

```
int cb_get_next(struct confd_trans_ctx *tctx, confd_hkeypath_t *kp, long
next)
{

    struct maapi_cursor *mc;
    if (next == -1) {
        if (tctx->t_opaque == NULL) {
            /* allocate the cursor */
            mc = (struct maapi_cursor *)malloc(sizeof(struct maapi_cursor));
            tctx->t_opaque = mc;
        } else {
            /* re-use previously allocated cursor */
            mc = (struct maapi_cursor *)tctx->t_opaque;
            maapi_destroy_cursor(mc);
        }
        maapi_init_cursor(maapisock, tctx->thandle, mc, "/to:foo/bar/baz");
    } else {
        mc = (struct maapi_cursor *)tctx->t_opaque;
    }
    maapi_get_next(mc);
    if (mc->n == 0) {
        confd_data_reply_next_key(tctx, NULL, -1, -1);
        return CONFD_OK;
    }
    confd_data_reply_next_key(tctx, &(mc->keys[0]), 1, 1);
    return CONFD_OK;
}

static int cb_create(struct confd_trans_ctx *tctx,
        confd_hkeypath_t *keypath)
{
    int rv = CONFD_OK;

    char *valstring = (char*)CONFD_GET_BUFPTR(&keypath->v[0][0]);

    rv = maapi_create(maapisock, tctx->thandle, "/to:foo/bar/baz{%s}", val-
string);
    return rv;
}

static int cb_remove(struct confd_trans_ctx *tctx,
        confd_hkeypath_t *keypath)
{
    int rv = CONFD_OK;

    char *valstring = (char*)CONFD_GET_BUFPTR(&keypath->v[0][0]);

    rv = maapi_delete(maapisock, tctx->thandle, "/to:foo/bar/baz{%s}", val-
string);

    return rv;
}

int cb_exists_optional(struct confd_trans_ctx *tctx, confd_hkeypath_t *kp)
{
    int ret = CONFD_OK;
    char *valstring = (char*)CONFD_GET_BUFPTR(&kp->v[0][0]);

    if (maapi_exists(maapisock, tctx->thandle, "/to:foo/bar/baz{%s}", val-
string))
```

continued on next page...

```
        confd_data_reply_found(tctx);
    else
        confd_data_reply_not_found(tctx);

    return ret;
}
```

Hooks

A hook is a function that is invoked within the transaction when an object is modified. The hook function has access to the transaction, so it can modify other objects in the transaction as necessary.

For more details on how hooks work, please refer to the ConfD User Guide, Chapter 10.

The backward compatibility flags don't work for Hook Callbacks. A runtime error: Application Communication Failure, will be generated if the CONFD_DAEMON_FLAG_LEAF_LIST_AS_LEAF is used.

Use of hooks and leaf-lists in your ConfD application will require updating your code.

In this section, we will modify our YANG module by adding a "hook" point:

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

  import tailf-common {
    prefix tailf;
  }

  container root {
    leaf-list config {
      tailf:callpoint configHookCP {
          tailf:transaction-hook subtree;
      }
      tailf:validate valPoint {
        tailf:dependency .;
      }
      type uint8;
    }
    leaf-list oper {
        config false;
        tailf:callpoint operCP;
        type uint8;
    }
  }
}
```

Callbacks used for a leaf-list hook point:

ConfD 6.4	ConfD 6.5
set_elem(): called when changing the leaf-list value by either a set or a delete.	
remove(): Called once, when the leaf-list is completely deleted.	create(): Called for each leaf-list instance.
remove(): called for each leaf-list instance being removed.	

Limitations

- There is no option to downgrade to a previous version of ConfD.
- CDB Subscribers on HA slave nodes cannot use the backward compatibility flags for iterating over the change set.
- Because a set/delete operation on a leaf-list is now composed of several create/delete operations, conflicting transactions might result in an unexpected value for the leaf-list.
- The backward compatibility flags mentioned above are not supported for hooks. Hook callbacks will always treat leaf-lists as lists. If the flag is set for a hook, it will cause an error on callback invocation for a leaf-list.
- The flags have no effect for validation callbacks. However just as for a list, a 'tailf:validate' statement for a leaf-list may use a 'tailf:call-once' substatement, to request a single invocation for the validation of the whole leaf-list.

Summary

In this application note, we have discussed the changes in YANG leaf-list handling in ConfD application code due to leaf-list changes introduced in ConfD 6.5. Examples of how leaf-lists were handled in ConfD 6.4 and earlier and the new handling for ConfD 6.5 and later have been presented. If you have additional questions related to these changes and have a support contract, please, contact Tail-f Support. Otherwise, you can ask questions on the ConfD User Community Forum.

For more information about ConfD:
www.tail-f.com

Tail-f Support:
https://support.tail-f.com/rt/

ConfD User Community Forum:
http://discuss.tail-f.com/

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