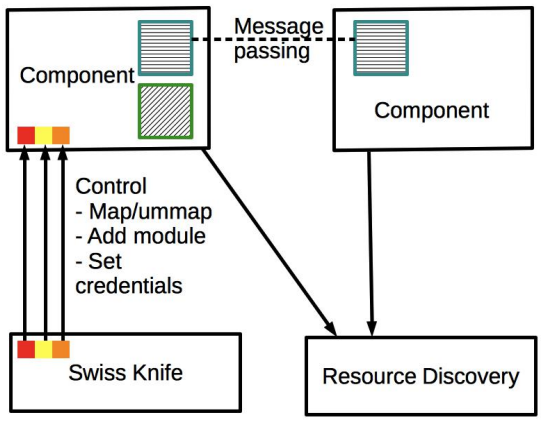
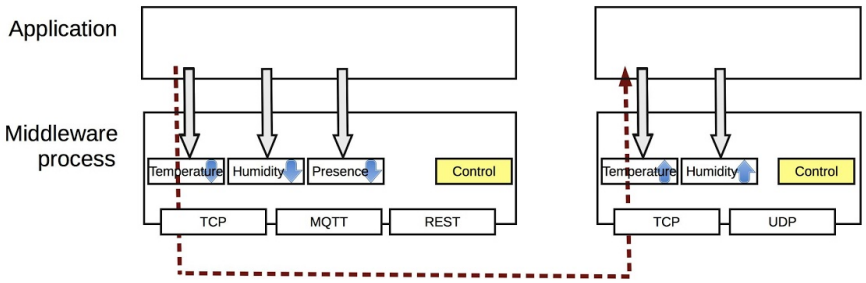
MW Tutorial

0. MW Basics

In this document we describe a middleware for managing interactions in dynamic environments. It is a framework for adaptive and dynamically reconfigurable components. It supports dynamic reconfiguration at runtime for management of interactions/information flows. That means not only can applications reconfigure themselves, but they can be controlled from a third party.



A *component* represents a process (e.g application, service) whose communication is managed by the middleware. Communication occurs through *endpoints,* which are essentially typed communication ports. A component may have a number of endpoints which can be *mapped* (connected) to any number of other endpoints to enable communication.



1. MW Introduction

This section describes how to get started with the MW. Sec 1.1 describes the installation steps and Sec 1.2 the basics of the API necessary to integrate it with your component .

1.1 Dependencies

The middleware requires json-c:

```
sudo apt-get install libjson0 libjson0-dev
```

Some com and access control modules require specific libraries. To install mosquitto:

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
sudo apt-get update
sudo apt-get install libmosquitto
```

To install openssl:

```
sudo apt-get install libssl-dev
```

1.2 Installation

Download the git repository:

```
git clone https://github.com/ComFlux/ComFlux-Middleware.git
```

The middleware uses links to uthash, krb5, and jsonschema-c submodules.

```
cd middleware
git submodule init
git submodule update --recursive
```

Install the middleware:

```
cmake .
make
sudo make install
```

Note that to generate SSL libraries and Kerberos access module the following command must replace the first one:

```
cmake -DSSLENABLED:BOOL=ON -DKRBENABLED:BOOL=ON .
```

To generate mosquitto com modules set the MQTTENABLED argument

```
cmake -DMQTTENABLED:BOOL=ON .
```

This will install in the build directory and will have the following structure:

1. `middleware/build/bin` contains binaries and configurations for the following
    a. `middleware/build/bin/core/` Contains the core binary and the configuration
    b. `middleware/build/bin/rdc/` Resource Discovery Component code and configurations
    c. `middleware/build/bin/swiss_knife/` Command line code and configurations
    d. `middleware/build/bin/examples/` Examples

2. `middleware/build/lib` contains the included communication and access control modules and their config files

Furthermore, system-wide the following paths are accessible:

1. `/usr/local/bin/core` is the path to the core binary
2. `/usr/local/lib` contains libmiddleware_api and libmiddleware_utils
3. `/usr/local/etc/middleware/` is the installation path for the middleware libraries and util components
    a. `/usr/local/etc/middleware/com_modules/` contains the dynamic libraries for the provided communication modules
    b. `/usr/local/etc/middleware/access_modules/` contains the dynamic libraries for the provided access control modules
    c. `/usr/local/etc/middleware/preloaded_schemata/` definitions for default endpoints
    d. `/usr/local/etc/middleware/rdc` Resource Discovery Component binary and config files
    e. `/usr/local/etc/middleware/swiss_knife` The command line tools

1.3 MW API

The installation provides an API to the developer. The following headers are available:

- `middleware.h`
    - This header contains communication and configuration functionality with the core.
- `endpoint.h`
    - This header provides endpoint definitions and endpoint-specific commands.
- `load_mw_config.h`
    - To facilitate middleware deployment configuration can be applied directly from json structures or fles.

Each component spawns the core of the middleware and connects to it. Using the middleware.h function

```
char* mw_init(const char* cpt_name, int log_lvl, bool use_socketpair);
```

The examples contain the following line

```
char* app_name = mw_init("sink_cpt", config_get_core_log_lvl(), true);
```

In case of success app_name should have the same value as the provided cpt_name, "sink_cpt" in this example. In case of error, such as core not started or communication not opened, the returned value is NULL.

These configurations can be summed up in a configuration file and applied at the beginning. This is an example of a config file that specifies the log level for for both, the app and the core of the mw.

```
{
    "app_config":{
        "log_level": 1,
        "log_file": "app.log",
        "manifest":{
```

```
            "name": "simple_sink"
        }
    },

    "core_config":{
        "log_level": 1,
        "log_file": "core.log",
        "com_libs": [
            {
                "lib_path": "../../lib/libcommoduletcp.so",
                "cfg_path": "../../lib/src_mw_cfg.json"
            }
        ],
        "access_libs": [
            { }
        ]
    }
}
```

The *manifest* is used to describe the current component to any entity that connects and can be any valid json. The configuration for the core specifies com and access control libs with their configs.

The app loads these configs with the following instruction before starting mw:

```
load_mw_config(mw_cfg_path);
```

And after starting the core, it loads the necessary libraries:

```
config_load_com_libs_array();
config_load_access_libs_array();
```

2. Endpoints and Components

The next step is to become familiar with the usage of endpoints and message passing by using practical examples.

Endpoint types correspond to communication patterns. In this section we explore source - sink

2.1 Typed messages

Run a simple source - sink examples by opening two terminals and typing

```
./simple_source src_tcp_cfg.json
./simple_sink 127.0.0.1:1503 snk_tcp_cfg.json
```

The file simple_source.c instantiates an endpoint

```
ENDPOINT *ep_src = endpoint_new_src_file(
                    "ep_source",
                    "example src endpoint",
```

```
                    "example_schemata/datetime_value.json");
```
Where the first argument is a generic name of the endpoint -- it is not required uniquely identify an object --, a user readable description, and json schema description of the message. Messages sent by ep_src are validated against this schema.

In simple_sink.c a sink endpoint
```
ENDPOINT *ep_snk = endpoint_new_snk_file(
                    "ep_sink", /* name */
                    "example snk endpoint", /* description */
                    "example_schemata/datetime_value.json", /* message schemata */
                    &print_callback); /* handler for incoming messages */
```

2.2 Queries and Filters

To pass messages from the source and the sink one of them needs to map to the other. A component may have multiple endpoints. Selecting between them can be done with queries on the *metadata* associated. Making sure that is the correct component is done by querying the component's *manifest*. The following code maps the sink endpoint to the source endpoint.
```
int map_result = endpoint_map_to(ep_snk, src_addr, ep_query_str, cpt_query_str);
```

In case of success the returned value is 0, otherwise an error code is provided.

Mapping involves
- the component's endpoint,
- the address of the target component,
- a query for the component at that address and
- an endpoint query that is meant to filter out through multiple endpoints of that component.

Queries are structured as a list of json strings, each individual query having the following form:
```
"attribute_name op value"
```
For examples, the simple_sink.c and creates an endpoint query for an endpoint called 'ep_source'.
```
Array *ep_query_array = array_new(ELEM_TYPE_STR);
array_add(ep_query_array, "ep_name = 'ep_source'");
JSON *ep_query_json = json_new(NULL);
json_set_array(ep_query_json, NULL, ep_query_array);
char* ep_query_str = json_to_str(ep_query_json);
```
Additionally, the middleware will automatically add queries about the endpoint.
For the component, the example creates an empty query that will match any component.
```
char* cpt_query_str = "";
```

Incoming and outgoing messages can be filtered using the same syntax of json query.
```
endpoint_add_filter(ep_snk, "value > 5");
```

This command will filter out json messages that don't conform and the core will pass to the component only those with 'value' attributes greater than 5.

2.3 Communication patterns

The first example opened a channel between a source and a sink, with a unidirectional message. Note that for a request-response communication there are two message types: a request and a response. A request endpoint is declared in simple_req.c

```
ENDPOINT* ep_req = endpoint_new_req_file(
                    "ep_req",
                    "example req endpoint",
                    "example_schemata/datetime_value.json", /* request schemata */
                    "example_schemata/datetime_value.json", /* response schemata */
                    &print_callback);
```

A response endpoint is declared in simple_resp.c

```
ep_resp = endpoint_new_resp_file("ep_response",
                    "example res endpoint",
                    "example_schemata/datetime_value.json", /* request schemata */
                    "example_schemata/datetime_value.json", /* response schemata */
                    &print_callback);
```

The mapping will validate both schemata.

2.4 Compile your new component

The distribution provides the core and the library of the middleware. To compile your component run:

```
 cc -o your_component your_component.c -lmiddleware_api -lmiddleware_utils
```

All functions presented so far are implemented in the `middleware_api`. The `middleware_utils` library contains data structures used in the mw and functions to manipulate them. For instance json, hashmap, array, message.

3. Modules

The power of the middleware rests in providing a mechanism to adapt to a dynamic environment in terms of underlying communication infrastructure and access control. This is achieved with a module-based design, where by the mechanisms to load modules at run time.

1. Communication modules
2. Access control modules

Furthermore, we provide the means to implement these modules.

3.1 Communication Modules

Com modules allow opening connections and sending data via different interfaces. They are loaded on the run as dynamic libraries with a set of predefined functions.

In a component you can individually load com modules:

```
int err = mw_load_com_module(
                    "../../lib/libcommodulecommrest.so",
                    "../../lib/rest_dst.cfg.json");
```

The first argument is the path of library and the second argument is the configuration file. In case of success the returned value is 0.

Com modules implement the interface in `middleware/modules/com_modules/com.h`. All functions are required.

There are two types of com modules. To differentiate between them make sure to set com_is_bridge accordingly.

- *Bridge modules*, expect a middleware core/component on the other side of the communication channel
- *Interfacing modules*, interface with non-middleware services. The distribution contains for communication with MQTT brokers and REST components (see commodulemqtt and commodulecommrest)

Notes on com modules:

- Mappings can address a specific module with the function call, for instance `endpoint_map_module(ep_src, "commqtt", mqtt_addr, ep_query_str, cpt_query_str);`
  Where the name of the module, 'commqtt' is specified in its configuration file.
- When passing a simple address to the core, without providing the module, such as endpoint_map_to or mw_add_rdc, the core will attempt all available modules.

Implemented com modules are

- commoduletcp - bridges mw via tcp sockets
- commodulesockpair - socketpair for communication between component and core
- commodulemqtt - interfaces with an mqtt broker
- commodulemqttbridge - bridges components using an mqtt broker and separated topics
- commodulecommrest - interfaces with a REST service
- commodulecommudp - bridges using udp datagrams
- commodulecommssl (requires SSLENABLED flag) - ssl encripted tcp communication

3.2 Access Control Modules

Access control modules allow to add and different types of authentice
Similarly, access control modules can be loaded with the following command.

```
int err = mw_load_access_module(
                    "../../lib/libmoduleaclplain.so",
                    "../../lib/aclplain.cfg.json");
```

Implemented access modules are

- moduleaclplain - useranme and password
- moduleaclssl (requires SSLENABLED) -
- moduleaclkrb (requires KRBENABLED) - kerberos

Access modules implement the interface in `middleware/modules/access_modules/auth.h`. All functions are required.

4. Resource Discovery: Registration and Lookup

The Resource Discovery Component (RDC) is a catalog of component data that can be queried and updated. When connecting
Run the RDC with the command `rdc`. The default configuration file (`/usr/local/etc/middleware/rdc/config/rdc.cfg.json`) loads the TCP com module and listens on port 1508. To change this run `rdc your_rdc_config.json` with your configuration file.

4.1 Using the RDC

Examples for discovering new resources and mapping to them using an RDC are lookup_source and lookup_sink.The difference from the simple source-sink scenario is that they do not share information about each other's existence, and the sink will query a RDC to find endpoints that match the mapping query.

Add the one or multiple RDCs to your component by giving its address
`mw_add_rdc("127.0.0.1:1508");`
And then register your component metadata to all known RDCs.
`mw_register_rdcs();`

The lookup_sink maps to a maximum of 10 matching endpoints provided by the RDC.
`endpoint_map_lookup(ep_snk, ep_query_str, cpt_query_str, 10);`

5. Swiss Knife

Swiss_knife allows interacting with the middleware via command line tools. Commands allow the user the application of a range of middleware instructions (as available through the API) from a third-party, at runtime.

```
Usage:
 > swiss_knife
       -a {msg | req | map | map_lookup | terminate | add_rdc | rdc_list}
       -T {target_addr | rdc_addr}      -t <target ep query>
       -R <remote_addr>           -r <remote ep query>
       -m <request schema filename>     -n <response schema filename>
       -M <json message>
       -f <filename>
Examples:
  > swiss_knife -a msg -T "127.0.0.1:1502" -m msg_term.json -t "[\"ep_name = 'TERMINATE'\"]" -M {}
 > swiss_knife -a terminate -T "127.0.0.1:1502"
 > swiss_knife -a add_rdc -T "127.0.0.1:1502" -R "127.0.0.1:1508"
```

Example of usage:

- `swiss_knife -a rdc_list -T "127.0.0.1:1508"`
  List everything from the RDC

- `swiss_knife -a msg -m /Users/admin/dev/middleware/build/bin/examples/example_schemata/msg_example.json -T "127.0.0.1:1508" -t "ep_name = 'source'" -M "{ \"the_number\":2}"`
  Map to an endpoint and send a message. The path to the json schema must be absolute.

- `swiss_knife -a terminate -T "127.0.0.1:1503"`
  Terminate a core instance and the app associated with it

- `swiss_knife -a add_rdc -T "127.0.0.1:1503" -R "127.0.0.1:1508"`
  Add the address of an RDC to a component

These commands use on default endpoints declared in the core, by default. Messages sent to these endpoints will pass unnoticed to the application.