Transponder Abstraction Interface

NTT Electronics America

Wataru Ishida

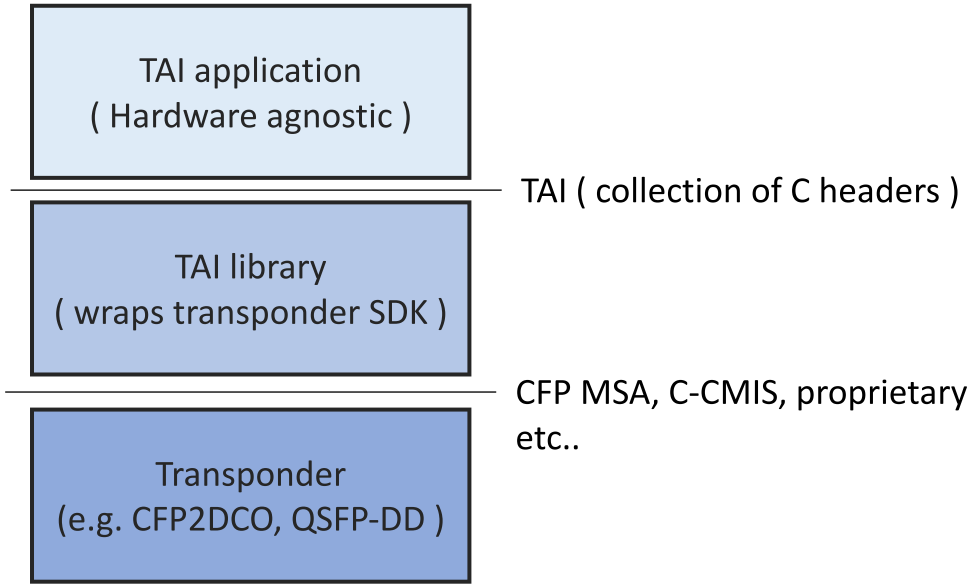
Topics

1. TAI basics from software perspective
2. TAI meta library
3. TAI shell
4. TAI multiplexer
5. TAI library framework

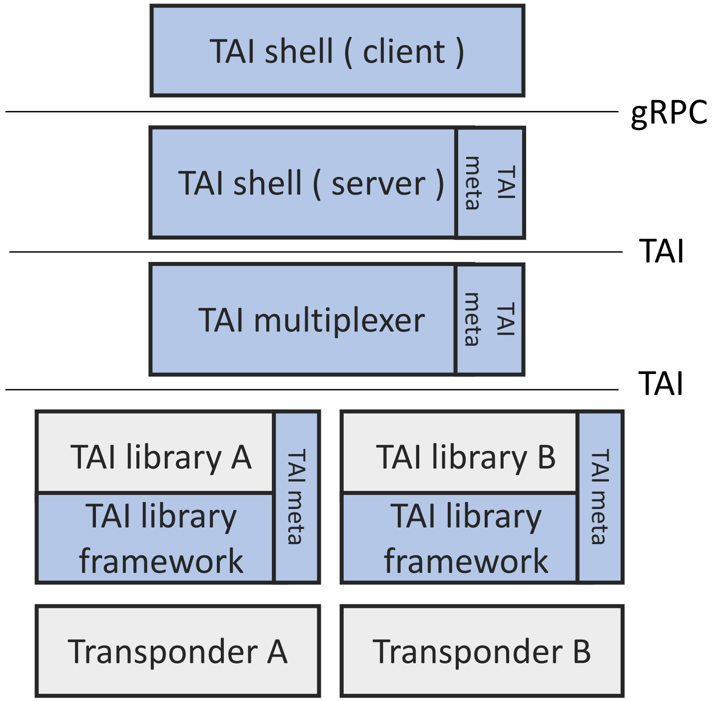
1. TAI basics from software perspective

Transponder Abstraction Interface - TAI

- Open software interface between transponder and NOS application
- + various utility tools to help developing transponder NOS



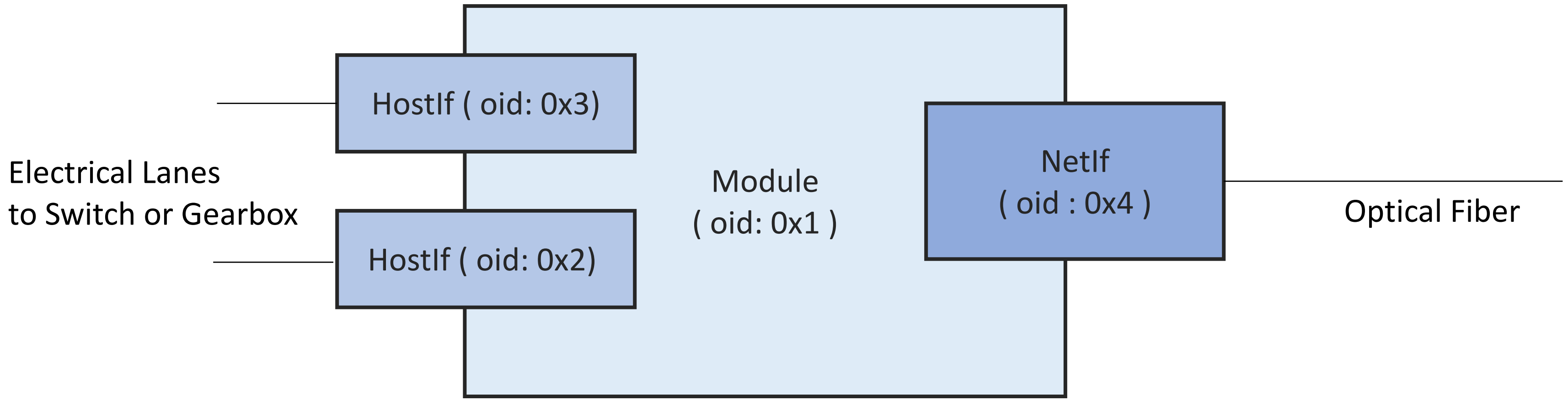
TAI – open software interface



Various utility tools (blue boxes) in TAI

TAI object

- 3 objects : module, network interface, and host interface
- Each object gets a unique object ID (oid) from TAI library when created
- Network interface and Host interface belong to one module



Example TAI object composition : 200G CFP2DCO (1 lambda, 2x100GbE)

TAI attribute

- Each object type has set of pre-defined attributes
    - All attribute uses *tai_attribute_t*
- TAI application controls hardware by setting and getting these attributes

```
tai_attribute_t attr;
attr.id = TAI_NETWORK_INTERFACE_ATTR_TX_LASER_FREQ;
attr.value.u64 = 193500000000000; // 193.5Thz
netif_api->set_network_interface_attribute(oid, &attr);
```

Example of setting TAI attribute

```
/**
 * @brief Network interface attribute IDs
 */
typedef enum _tai_network_interface_attr_t
{
    /**
     * @brief Start of attributes
     */
    TAI_NETWORK_INTERFACE_ATTR_START,

    ...
    ...

    /**
     * @brief The TX laser frequency in Hz
     *
     * @type #tai_uint64_t
     * @flags CREATE_AND_SET
     * @default vendor-specific
     */
    TAI_NETWORK_INTERFACE_ATTR_TX_LASER_FREQ,

    ...
    ...
} tai_network_interface_attr_t;
```

Example of pre-defined attributes

```
/**
 * @brief Data Type
 */
typedef union _tai_attribute_value_t
{
    bool booldata;
    char chardata[32];
    tai_uint8_t u8;
    tai_int8_t s8;
    tai_uint16_t u16;
    tai_int16_t s16;
    tai_uint32_t u32;
    tai_int32_t s32;
    tai_uint64_t u64;
    tai_int64_t s64;
    tai_float_t flt;
    tai_pointer_t ptr;
    tai_object_id_t oid;
    tai_object_list_t objlist;
    tai_char_list_t charlist;
    tai_u8_list_t u8list;
    tai_s8_list_t s8list;
    tai_u16_list_t u16list;
    tai_s16_list_t s16list;
    tai_u32_list_t u32list;
    tai_s32_list_t s32list;
    tai_float_list_t floatlist;
    tai_u32_range_t u32range;
    tai_s32_range_t s32range;
    tai_object_map_list_t objmaplist;
    tai_attr_value_list_t attrlist;
    tai_notification_handler_t notification;
} tai_attribute_value_t;

typedef struct _tai_attribute_t
{
    tai_attr_id_t id;
    tai_attribute_value_t value;
} tai_attribute_t;
```

tai_attribute_t : common struct for TAI attribute

TAI API

- # of *directly* exposed functions are only 7. All defined in tai.h
    1. tai_api_initialize
    2. tai_api_query
    3. tai_api_uninitialized
    4. tai_log_set
    5. tai_object_type_query
    6. tai_module_id_query
    7. tai_dbg_generate_dump

- Core functionality (create TAI objects, set attribute etc..) is categorized by *tai_api_t* and retrieved as function pointers via *tai_api_query*

tai_api_initialize

- The application can pass *module_presence* callback to *tai_api_initialize*
- TAI library calls *module_presence* callback to show module presence
- *char* location* is used to identify a module since an object ID is not assigned yet (TAI module is not created)

```cpp
int event_fd;
std::queue<std::pair<bool, std::string>> queue;

void module_presence(bool present, char* location) {
    uint64_t v = 1;
    queue.push(std::pair<bool, std::string>(present, std::string(location)));
    write(event_fd, &v, sizeof(uint64_t));
}

int main() {
    tai_service_method_table_t services = {0};

    services.module_presence = module_presence;
    event_fd = eventfd(0, 0);

    tai_api_initialize(0, &services);

    while (true) {
        uint64_t v;
        read(event_fd, &v, sizeof(uint64_t));
        while ( !queue.empty() ) {
            auto p = queue.front();
            auto present = p.first;
            auto location = p.second;

            if (present) {
                create_module(location)
            }

            queue.pop();
        }
    }
}
```

Example of TAI application initialization (C++)

tai_api_query

- TAI's core functionality needs to be retrieved via *tai_api_query*
- 4 API types (*tai_api_t*) are defined
  - TAI_API_MODULE
  - TAI_API_NETWORKIF
  - TAI_API_HOSTIF
  - TAI_API_META
- Function pointers are returned from TAI library if it is supported

```c
/**
 * @brief Module method table retrieved with tai_api_query()
 */
typedef struct _tai_module_api_t
{
    tai_create_module_fn            create_module;
    tai_remove_module_fn            remove_module;
    tai_set_module_attribute_fn     set_module_attribute;
    tai_set_module_attributes_fn    set_module_attributes;
    tai_get_module_attribute_fn     get_module_attribute;
    tai_get_module_attributes_fn    get_module_attributes;

} tai_module_api_t;
```

tai_module_api_t definition from taimodule.h

```c
struct tai_api_method_table_t {
    tai_module_api_t* module_api;
    tai_host_interface_api_t* hostif_api;
    tai_network_interface_api_t* netif_api;
    tai_meta_api_t* meta_api;
};

tai_api_method_table_t g_api;

int main() {
    // right after calling tai_api_initialize()
    tai_api_query(TAI_API_MODULE, (void **)(&g_api.module_api));
    tai_api_query(TAI_API_NETWORKIF, (void **)(&g_api.netif_api));
    tai_api_query(TAI_API_HOSTIF, (void **)(&g_api.hostif_api));
    tai_api_query(TAI_API_META, (void **)(&g_api.meta_api));
}
```

Example of TAI API query

TAI module creation

- We got module location via *module_presence* callback
- We got module API function pointers via *tai_api_query*
- Call *create_module* with a list of attributes including the location to create a TAI module object
  - TAI library creates the object and assign object ID

```
void create_module(char* location) {
  tai_object_id_t oid;
  tai_attribute_t attrs[2];

  attrs[0].id = TAI_MODULE_ATTR_LOCATION;
  attrs[0].value.charlist.count = strlen(location);
  attrs[0].value.charlist.list = location;

  attrs[1].id = TAI_MODULE_ATTR_ADMIN_STATUS;
  attrs[1].value.u32 = TAI_MODULE_ADMIN_STATUS_UP;

  auto status = module_api->create_module(&oid, 2, attrs);

  if ( status != TAI_STATUS_SUCCESS ) {
    error("failed to create module whose location is %s", location);
  }

  info("module created. oid: 0x%x", oid);
}
```

Example of TAI module creation

TAI hostif/netif creation

- We got module oid
- We got hostif and netif API function pointers via *tai_api_query*
- Call *get_module_attributes* to get # of hostif and netif the module has
- Call *create_host_interface* and *create_network_interface* with the module oid and a list of attributes <u>including the index</u>

```
void create_netif_and_hostif(tai_object_id_t module_oid) {
  tai_attribute_t attrs[2];

  attrs[0].id = TAI_MODULE_ATTR_NUM_HOST_INTERFACES;
  attrs[1].id = TAI_MODULE_ATTR_NUM_NETWORK_INTERFACES;

  auto status = module_api->get_module_attributes(module_oid, 2, attrs);
  if ( status != TAI_STATUS_SUCCESS ) {
    error("failed to get module attributes");
  }

  auto num_hostif = attrs[0].value.u32;
  create_hostif(module_oid, num_hostif);

  auto num_netif = attrs[1].value.u32;
  create_netif(module_oid, num_netif);
}

void create_hostif(tai_object_id_t module_oid, uint32_t num) {
  for ( uint32_t i = 0; i < num; i++ ) {
    tai_object_id_t oid;
    tai_attribute_t attrs[1];
    attr.id = TAI_HOST_INTERFACE_ATTR_INDEX;
    attr.value.u32 = i;

    hostif_api->create_host_interface(&oid, 1, attrs);
  }
}
```

Example of TAI hostif and netif creation

TAI basics - summary

- TAI is an open software interface between transponder and NOS application
  - It also includes various utility tools

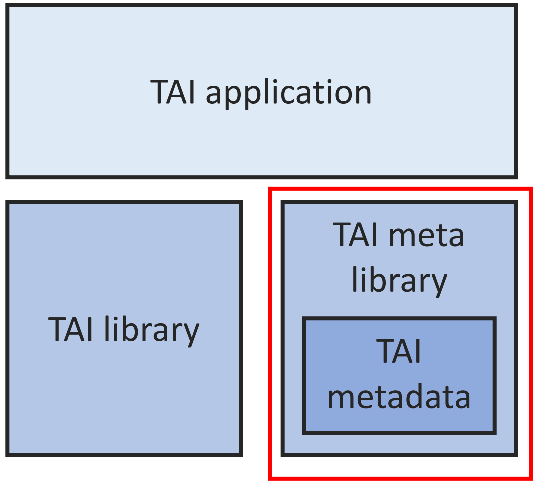
- TAI API is object oriented and declarative
  - Easy to use for NOS application
  - Difficult to implement TAI library
    - TAI library framework provides basic framework to develop TAI library

2. TAI meta library and TAI metadata

TAI meta library and TAI metadata

- TAI meta library : A utility library (libmeta-tai.so) to help TAI software development (e.g. stringify, memory allocation)

- TAI metadata : meta information about TAI objects and TAI attributes
  - This information is embedded inside TAI meta library



TAI metadata

- TAI metadata structs hold meta information like
  - human-readable attribute name
  - value type (uint32, bool etc..)
  - flags (read-only, create-only)

- TAI metadata structs are passed to various APIs in the TAI meta library

```
extern bool tai_metadata_is_allowed_enum_value(
        _In_ const tai_attr_metadata_t *metadata,
        _In_ int value);
```

```
/**
 * @brief Defines attribute metadata.
 */
typedef struct _tai_attr_metadata_t
{
    /**
     * @brief Specifies valid TAI object type.
     */
    tai_object_type_t                           objecttype;

    /**
     * @brief Specifies valid attribute id for this object type.
     */
    tai_attr_id_t                               attrid;

    /**
     * @brief Specifies valid attribute id name for this object type.
     */
    const char* const                           attridname;

    /**
     * @brief Specifies valid short attribute id name for this object type.
     */
    const char* const                           attridshortname;
```

taimetadatatypes.sh

How TAI metadata is generated

taimodule.h

```
typedef enum _tai_module_attr_t
{
    /**
     * @brief Start of attributes
     */
    TAI_MODULE_ATTR_START,

    /**
     * @brief The location of the module
     *
     * Used (and required) in the tai_create_module
     * adapter to uniquely identify the module. T
     * slot identifier, or other value that all      e adapter to determine
     * which optical module is being initial
     *
     * @type #tai_char_list_t
     * @flags MANDATORY_ON_CREATE | CREATE_ONLY
     */
    TAI_MODULE_ATTR_LOCATION  = TAI_MODULE_ATTR_START,

    ...
}
```

taimetadata.c (auto-generated)

```
const tai_attr_metadata_t tai_metadata_attr_TAI_MODULE_ATTR_LOCATION = {
    .objecttype           = TAI_OBJECT_TYPE_MODULE,
    .attrid               = TAI_MODULE_ATTR_LOCATION,
    .attridname           = "TAI_MODULE_ATTR_LOCATION",
    .attridshortname      = "location",
    .attrvaluetype        = TAI_ATTR_VALUE_TYPE_CHARLIST,
    .flags                = TAI_ATTR_FLAGS_MANDATORY_ON_CREATE|TAI_ATTR_FLAGS_CREATE_ONLY,
    .isenum               = false,
    .enummetadata         = NULL,
    .isoidattribute       = false,
    .ismandatoryoncreate  = true,
    .iscreateonly         = true,
    .iscreateandset       = false,
    .isreadonly           = false,
    .isclearable          = false,
    .iskey                = false,
    .defaultvaluetype     = TAI_DEFAULT_VALUE_TYPE_NONE,
};
```

- TAI metadata is generated automatically by parsing the TAI header files when building the meta library
- libclang is used internally to do the header parsing

Example usage of TAI meta library - serialize

```
int serialize_module_oper_status() {
    int ret;
    char buf[128] = {0};
    tai_attribute_t attr = {0};
    tai_serialize_option_t option = {
        .valueonly = true,
        .json = false,
    };
    auto meta = tai_metadata_get_attr_metadata(TAI_OBJECT_TYPE_MODULE, TAI_MODULE_ATTR_OPER_STATUS);

    attr.id = TAI_MODULE_ATTR_OPER_STATUS;
    attr.value.s32 = TAI_MODULE_OPER_STATUS_READY;   Enum value (integer)

    ret = tai_serialize_attribute(buf, 128, meta, &attr, &option);
    if ( ret <  0 ) {
        return -1;
    }
    ret = strcmp(buf, "TAI_MODULE_OPER_STATUS_READY"); Human readable string
    if ( ret != 0 ) {
        return -1;
    }
    return 0;
}
```

Example usage of TAI meta library - deserialize

```
int deserialize_network_interface_attr() {
    int32_t value;
    int ret;
    tai_serialize_option_t option = {
        .human = true,
    };
    ret = tai_deserialize_network_interface_attr("tx-laser-freq", &value, &option);
    if ( ret < 0 ) {
        return -1;
    }
    if ( value != TAI_NETWORK_INTERFACE_ATTR_TX_LASER_FREQ ) {
        return -1;
    }
    return 0;
}
```

Human readable string

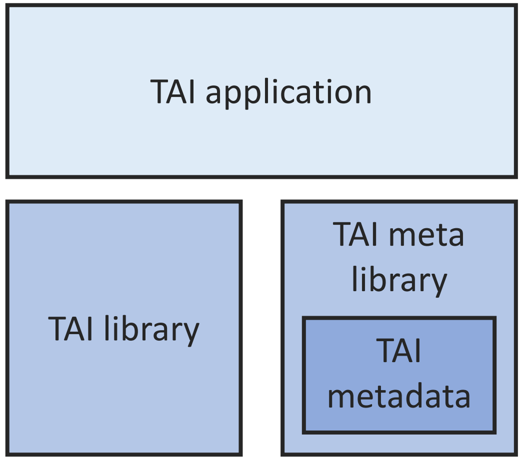
Enum value (integer)

Example usage of TAI meta library - memory

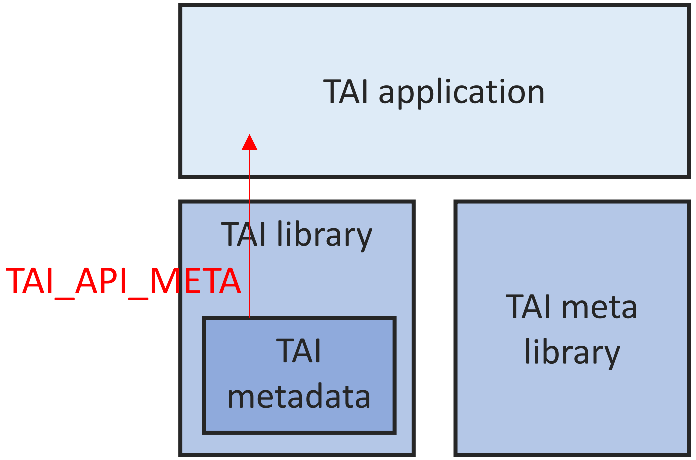
```
int deepcopy_attr_value() {
    const tai_attr_metadata_t* meta = tai_metadata_get_attr_metadata(TAI_OBJECT_TYPE_NETWORKIF,
TAI_NETWORK_INTERFACE_ATTR_TX_ALIGN_STATUS);
    tai_attribute_t src, dst = {0};
    tai_status_t status;
    status = tai_metadata_alloc_attr_value(meta, &src, NULL);  Memory allocation
    if ( status != TAI_STATUS_SUCCESS ) {
        printf("failed to alloc attr value: %d\n", status);
        return -1;
    }
    status = tai_metadata_alloc_attr_value(meta, &dst, NULL);
    if ( status != TAI_STATUS_SUCCESS ) {
        printf("failed to alloc attr value: %d\n", status);
        return -1;
    }
    src.value.s32list.count = 2;
    src.value.s32list.list[0] = TAI_NETWORK_INTERFACE_TX_ALIGN_STATUS_TIMING;
    src.value.s32list.list[1] = TAI_NETWORK_INTERFACE_TX_ALIGN_STATUS_OUT;

    status = tai_metadata_deepcopy_attr_value(meta, &src, &dst);  Deep copy
    if ( status != TAI_STATUS_SUCCESS ) {
        return -1;
    }
    if ( dst.value.s32list.count != 2 ) {
        return -1;
    }
    if ( dst.value.s32list.list[0] != TAI_NETWORK_INTERFACE_TX_ALIGN_STATUS_TIMING ) {
        return -1;
    }
    if ( dst.value.s32list.list[1] != TAI_NETWORK_INTERFACE_TX_ALIGN_STATUS_OUT ) {
        return -1;
    }
}
```

TAI_API_META : new TAI API type

- TAI_API_META is recently added to provide the metadata information directly from TAI library
- This is needed to support multiple TAI libraries with different metadata in one system
  - e.g.) EdgeCore Cassini, Wistron Galileo – mix of ACO and DCO PIU in one system



TAI library without TAI_API_META support



TAI library with TAI_API_META support

```
/**
 * @brief Meta methods table retrieved with tai_api_query()
 */
typedef struct _tai_meta_api_t
{
    tai_meta_list_metadata_fn              list_metadata;
    tai_meta_get_attr_metadata_fn          get_attr_metadata;
    tai_meta_get_object_info_fn            get_object_info;
} tai_meta_api_t;
```

tai_meta_api_t definition

TAI meta library and TAI metadata - summary

- TAI meta library is a utility library that helps TAI software development
- TAI metadata is an auto-generated structs that hold meta information about TAI objects and TAI attributes
- TAI_META_API is recently added to support embedding TAI metadata inside a TAI library
  - This enables supporting multiple TAI libraries with different capability under TAI multiplexer environment (explained later)

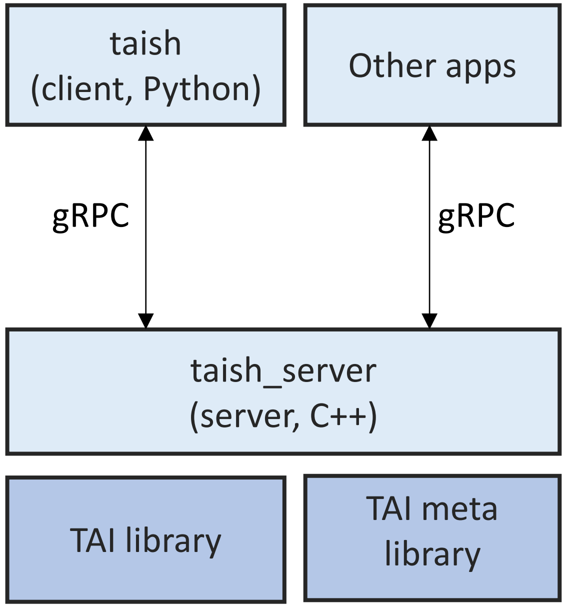
3. TAI shell

TAI shell

- A TAI application to quickly test TAI library interactively
- Server-client architecture
- gRPC is used for the communication between the server and the client

```
$ taish
> module 0
module(0)> netif 0
module(0)/netif(0)> set modulation-format dp-qpsk
module(0)/netif(0)> get modulation-format
dp-qpsk
module(0)/netif(0)> q
module(0)> get vendor-name
BASIC
module(0)>
```

taish : look and feel



TAI shell architecture

taish gRPC API

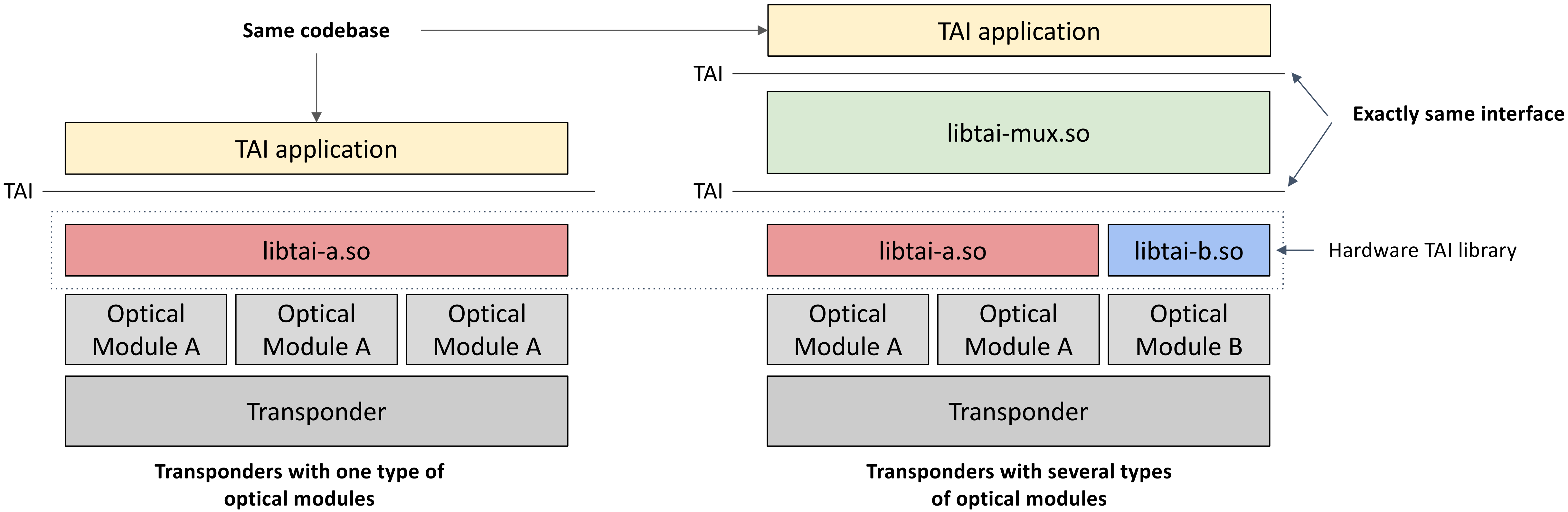
- gRPC : https://grpc.io/
- all TAI APIs can be executed over network (even monitoring)

```
syntax = "proto3";

package taish;

service TAI {
    rpc ListModule(ListModuleRequest) returns (stream ListModuleResponse);
    rpc ListAttributeMetadata(ListAttributeMetadataRequest) returns (stream ListAttributeMetadataResponse);
    rpc GetAttributeMetadata(GetAttributeMetadataRequest) returns (GetAttributeMetadataResponse);
    rpc GetAttribute(GetAttributeRequest) returns (GetAttributeResponse);
    rpc SetAttribute(SetAttributeRequest) returns (SetAttributeResponse);
    rpc ClearAttribute(ClearAttributeRequest) returns (ClearAttributeResponse);
    rpc Monitor(MonitorRequest) returns (stream MonitorResponse);
    rpc SetLogLevel(SetLogLevelRequest) returns (SetLogLevelResponse);
    rpc Create(CreateRequest) returns (CreateResponse);
    rpc Remove(RemoveRequest) returns (RemoveResponse);
}
```

4. TAI multiplexer

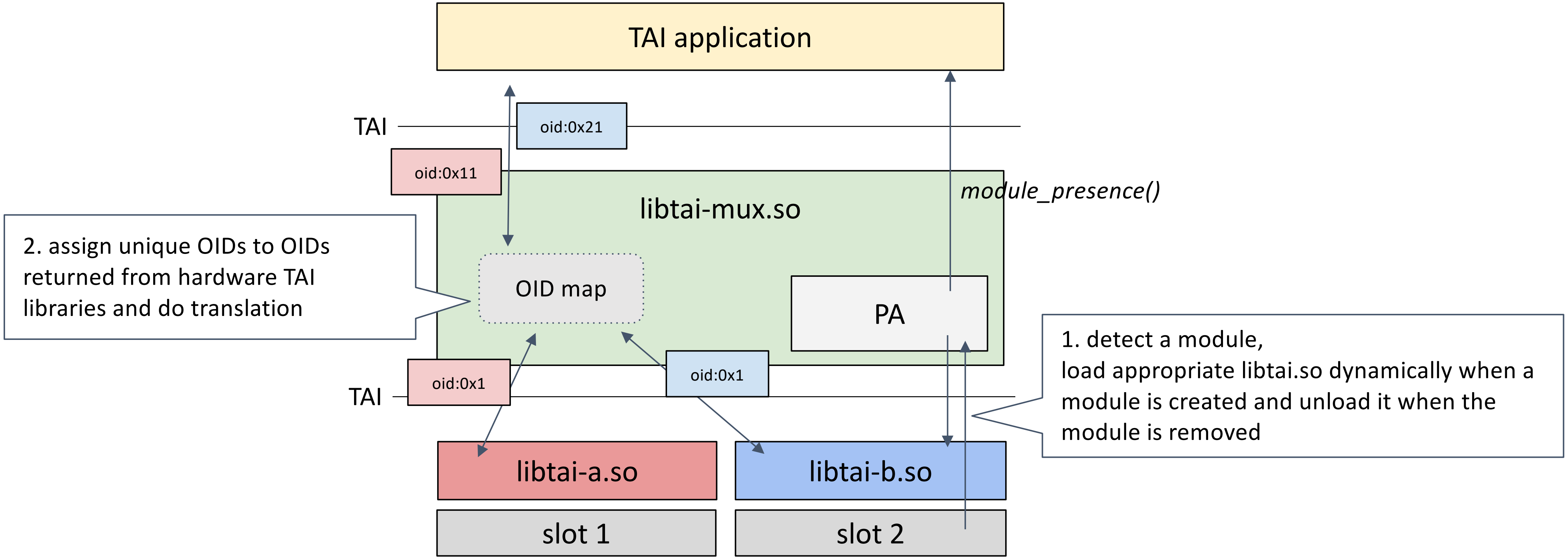
TAI multiplexer - libtai-mux.so

- TAI library to multiplex multiple TAI libraries
- Supports hardware which can have multiple types of optical module (e.g. Edgecore Cassini, Wistron Galileo)
- Available here:
  - https://github.com/Telecominfraproject/oopt-tai-implementations/tree/master/tai_mux



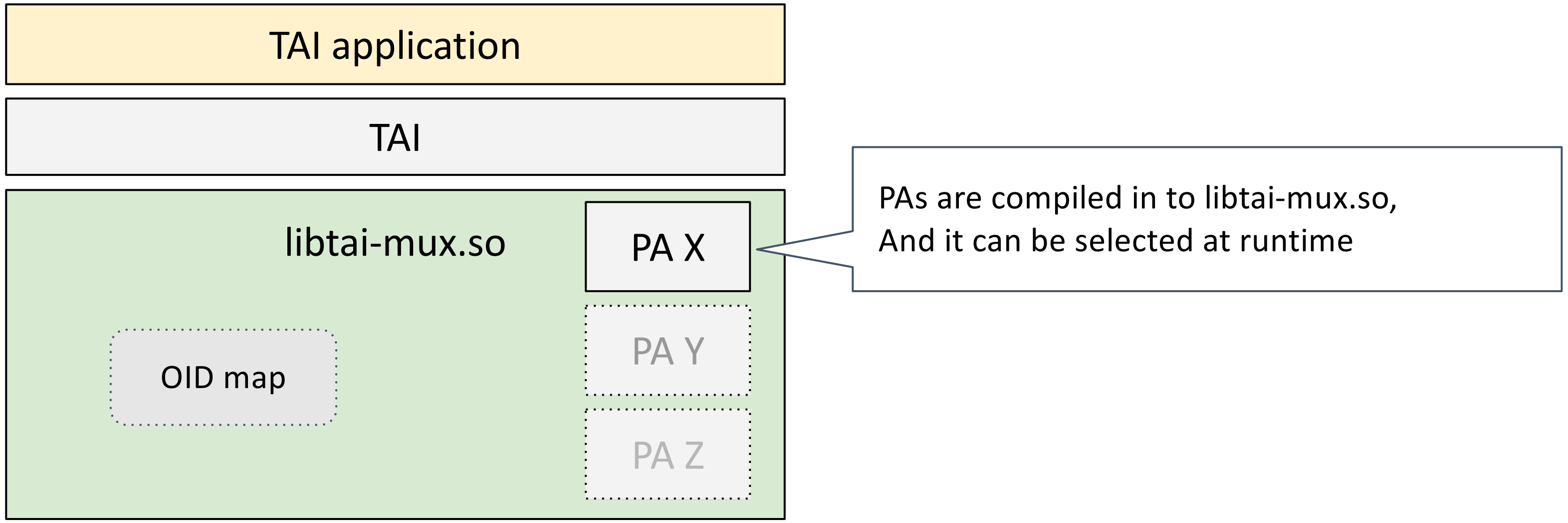
What libtai-mux.so does

1. Dynamic hardware TAI library loading/unloading
   - Platform Adapter (PA) detects modules and decides which hardware TAI library to use
2. Object ID (OID) mapping
   - Hardware TAI libraries could use the same object ID for different objects
   - libtai-mux.so manages object ID map and ensures unique IDs are returned to TAI application



Platform Adapter (PA)

- PA detects module insertion/removal, decides which hardware TAI library to load/unload based on their policy/configuration
- libtai-mux.so has modular design to support various types of PA
  - The methods to detect modules varies between OS and hardware
  - software/system vendors may develop their own PA
  - Users can select a PA at runtime by passing an env variable `TAI_MUX_PLATFORM_ADAPTER`
- Currently 2 platform adapters are open sourced : static PA and exec PA



static PA

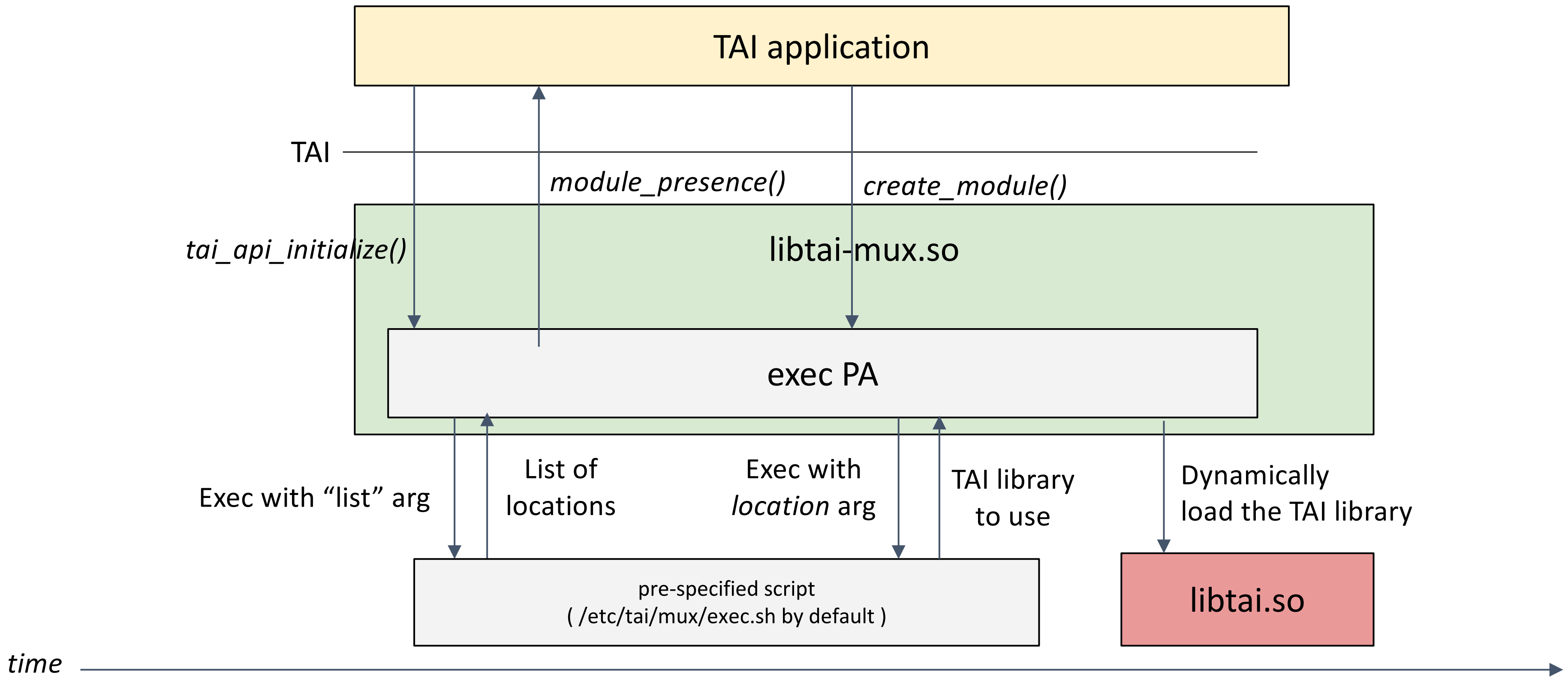
- static PA is a PA which uses static configuration for TAI library selection
- It doesn't do module detection and blindly call *module_presence()* callback based on the static configuration
- Configuration format is json. The key is location of the module and the value is the TAI library to use
- By using the configuration below, libtai-a.so is used for modules whose location is 1,2,3,4, and libtai-b.so is used for modules whose location is 5,6,7,8

```
{
  “1”: “libtai-a.so”,
  “2”: “libtai-a.so”,
  “3”: “libtai-a.so”,
  “4”: “libtai-a.so”,
  “5”: “libtai-b.so”,
  “6”: “libtai-b.so”,
  “7”: “libtai-b.so”,
  “8”: “libtai-b.so”
}
```

/etc/tai/mux/static.json

exec PA

- exec PA is a PA which executes a pre-specified script for TAI library selection
  - Platform Adapter (PA) detects modules and decides which hardware TAI library to use
- The script is executed with ‘list’ argument during the module detection
  - The script needs to return list of modules that is on the system
- The script is executed with a location string when *create_module()* is called by TAI application
  - The script needs to return the name of the TAI library to use for the location



TAI multiplexer - summary

- TAI multiplexer is a TAI library that multiplexes multiple TAI libraries
- Platform Adapter is the core function block of TAI multiplexer
  - Platform Adapter handles the module detection and TAI library selection
  - Modular design to support various types of platform adapter

5. TAI library framework

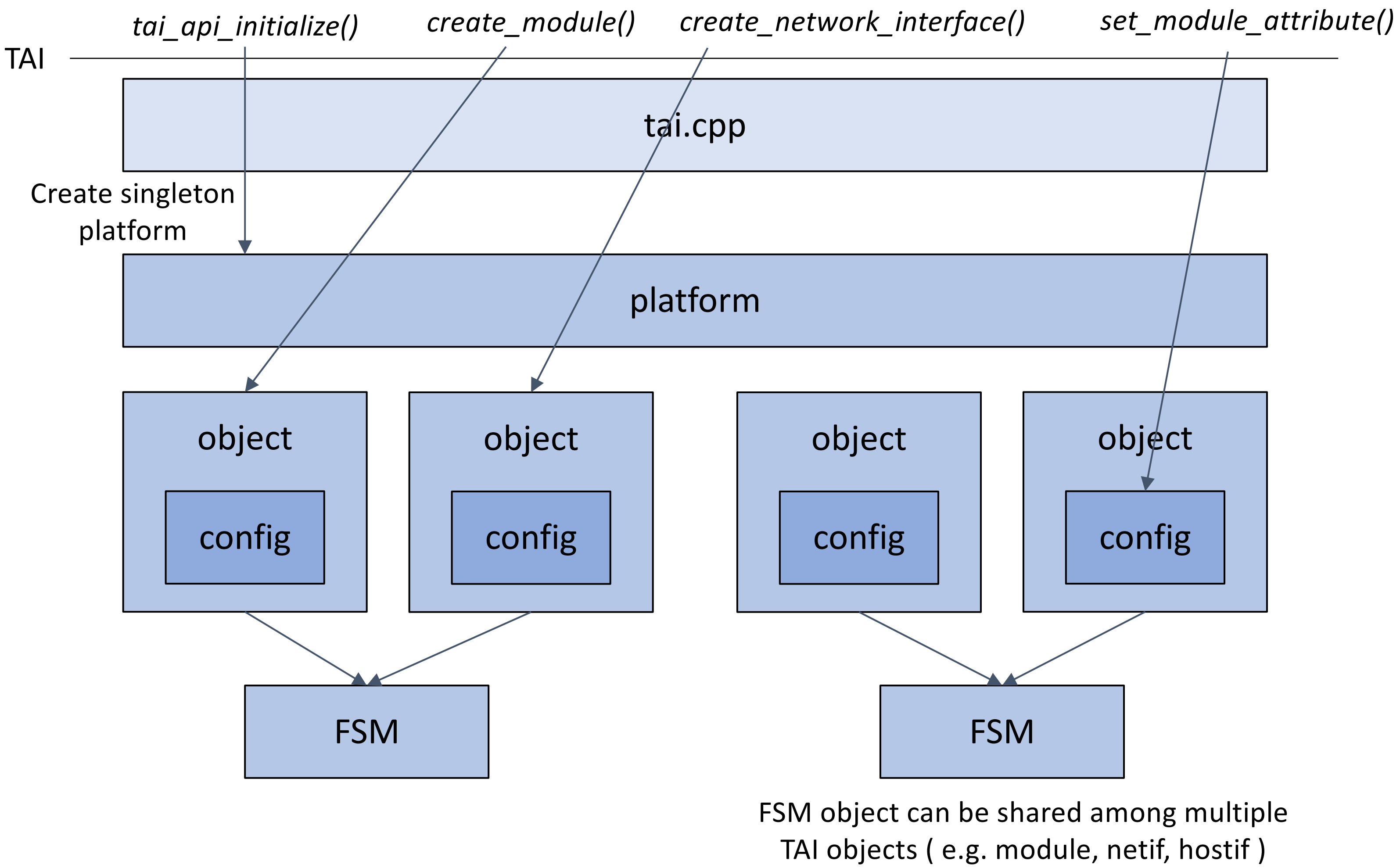
TAI library framework

- https://github.com/Telecominfraproject/oopt-tai/tree/master/tools/framework
- A framework to build a TAI library (C++17)
- Consists of 5 files
  - tai.cpp - TAI API boilerplate : delegates the actual handling to a platform object
  - platform.hpp - a singleton class that handles TAI object management (create, remove)
  - object.hpp - a base class of TAI object. Handles set and get of TAI attribute
  - config.hpp - a class that holds TAI attribute
  - fsm.hpp – optional FSM class. Provides state handling in TAI library

- Two examples under /examples directory
  - Stub – bare minimum example (~300 lines including many comments)
  - Basic – basic example with FSM (~800 lines)

- TAI multiplexer also uses this framework

TAI library framework - architecture



Supported attribute enumeration

Module attribute

Network Interface attribute

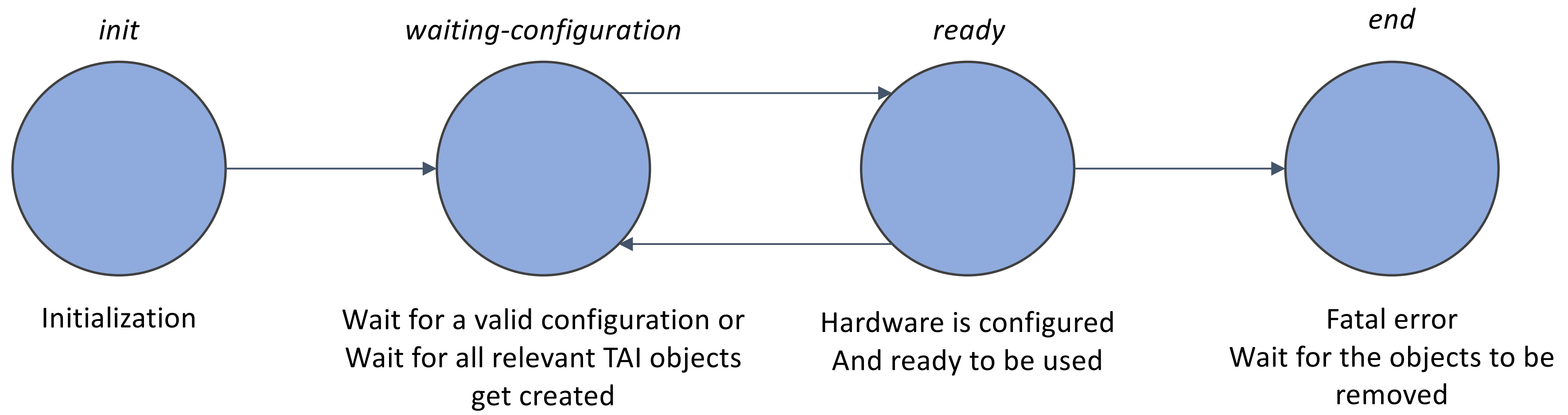
Host Interface attribute

```
using M = AttributeInfo<TAI_OBJECT_TYPE_MODULE>;
using N = AttributeInfo<TAI_OBJECT_TYPE_NETWORKIF>;
using H = AttributeInfo<TAI_OBJECT_TYPE_HOSTIF>;

template <> const AttributeInfoMap<TAI_OBJECT_TYPE_MODULE> Config<TAI_OBJECT_TYPE_MODULE>::m_info {
    stub::M(TAI_MODULE_ATTR_LOCATION),
    stub::M(TAI_MODULE_ATTR_VENDOR_NAME)
        .set_default(&tai::stub::default_tai_module_vendor_name_value),   Default value setting
    stub::M(TAI_MODULE_ATTR_NUM_NETWORK_INTERFACES)
        .set_default(&tai::stub::default_tai_module_num_network_interfaces),
    stub::M(TAI_MODULE_ATTR_NUM_HOST_INTERFACES)
        .set_default(&tai::stub::default_tai_module_num_host_interfaces),
    stub::M(TAI_MODULE_ATTR_ADMIN_STATUS)
        .set_validator(EnumValidator({TAI_MODULE_ADMIN_STATUS_DOWN, TAI_MODULE_ADMIN_STATUS_UP})),
    stub::M(TAI_MODULE_ATTR_MODULE_SHUTDOWN_REQUEST_NOTIFY),
    stub::M(TAI_MODULE_ATTR_MODULE_STATE_CHANGE_NOTIFY),                  Attach validator
};

template <> const AttributeInfoMap<TAI_OBJECT_TYPE_NETWORKIF> Config<TAI_OBJECT_TYPE_NETWORKIF>::m_info {
    stub::N(TAI_NETWORK_INTERFACE_ATTR_INDEX),
    stub::N(TAI_NETWORK_INTERFACE_ATTR_TX_DIS),
    stub::N(TAI_NETWORK_INTERFACE_ATTR_TX_LASER_FREQ),
    stub::N(TAI_NETWORK_INTERFACE_ATTR_OUTPUT_POWER),
};

template <> const AttributeInfoMap<TAI_OBJECT_TYPE_HOSTIF> Config<TAI_OBJECT_TYPE_HOSTIF>::m_info {
    stub::H(TAI_HOST_INTERFACE_ATTR_INDEX),
};
```

code from stub library example

TAI library framework - FSM

- TAI API is declarative (create, remove, set and get)
  - User of the TAI library doesn't need to care about the hardware state
- Hardware control involves state handling in most case
  - e.g.) set to low-power mode, change configuration, set to high-power mode
- Built-in FSM bridges these two worlds
  - User of the framework can say "go to this *software* state when this attribute is configured"
- TAI library framework has 4 pre-defined states
  - Supports adding custom states as needed



set_fsm_state()

```
tai_status_t module_tributary_mapping_getter(tai_attribute_t* const attribute, void* user) {
    auto fsm = reinterpret_cast<FSM*>(user);
    return fsm->get_tributary_mapping(attribute);
}

template <> const AttributeInfoMap<TAI_OBJECT_TYPE_MODULE> Config<TAI_OBJECT_TYPE_MODULE>::m_info {
    basic::M(TAI_MODULE_ATTR_LOCATION),
    basic::M(TAI_MODULE_ATTR_VENDOR_NAME)
        .set_default(&tai::basic::default_tai_module_vendor_name_value),
    basic::M(TAI_MODULE_ATTR_OPER_STATUS),
    basic::M(TAI_MODULE_ATTR_NUM_NETWORK_INTERFACES)
        .set_default(&tai::basic::default_tai_module_num_network_interfaces),
    basic::M(TAI_MODULE_ATTR_NUM_HOST_INTERFACES)
        .set_default(&tai::basic::default_tai_module_num_host_interfaces),
    basic::M(TAI_MODULE_ATTR_ADMIN_STATUS)
        .set_validator(EnumValidator({TAI_MODULE_ADMIN_STATUS_DOWN, TAI_MODULE_ADMIN_STATUS_UP}))
        .set_fsm_state(FSM_STATE_WAITING_CONFIGURATION),
    basic::M(TAI_MODULE_ATTR_TRIBUTARY_MAPPING)
        .set_getter(tai::basic::module_tributary_mapping_getter),
    basic::M(TAI_MODULE_ATTR_MODULE_SHUTDOWN_REQUEST_NOTIFY),
    basic::M(TAI_MODULE_ATTR_MODULE_STATE_CHANGE_NOTIFY),
    basic::M(TAI_MODULE_ATTR_NOTIFY),
};
```

Specify what state to move when this attribute is changed

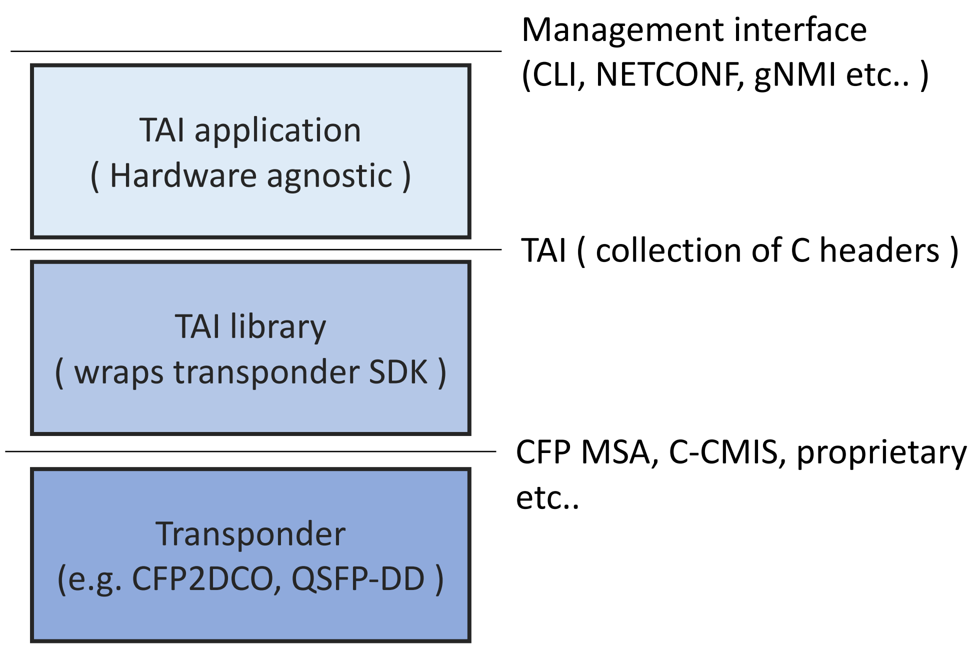
code from basic library example

TAI library framework - summary

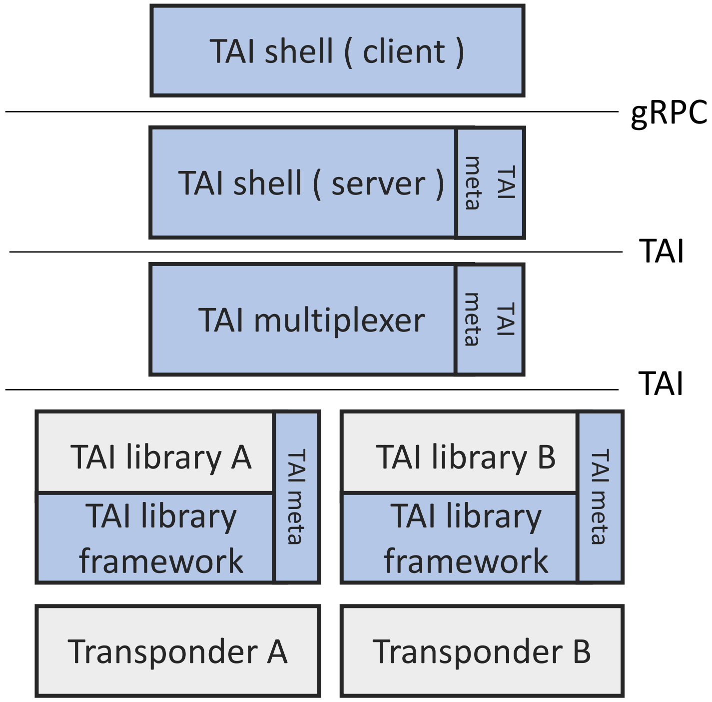
- TAI library development is tedious and sometimes difficult
  - Boilerplate code to expose the TAI API
  - The TAI API is declarative, but hardware control always requires state control
- TAI library framework reduces required effort to develop TAI library
  - Bare minimum TAI library in just 300 lines.
- Catching up with the new TAI functionality is easy
  - The new TAI_API_META can be automatically supported by just recompiling

TAI – future work

- Software infrastructure is ready for extension!
- More attributes to expose the hardware details (waiting for contribution from OOPT group)
- Supporting different kinds of hardware (e.g. client transceivers, peripheral hardware?)



TAI – open software interface



Various utility tools (blue boxes) in TAI