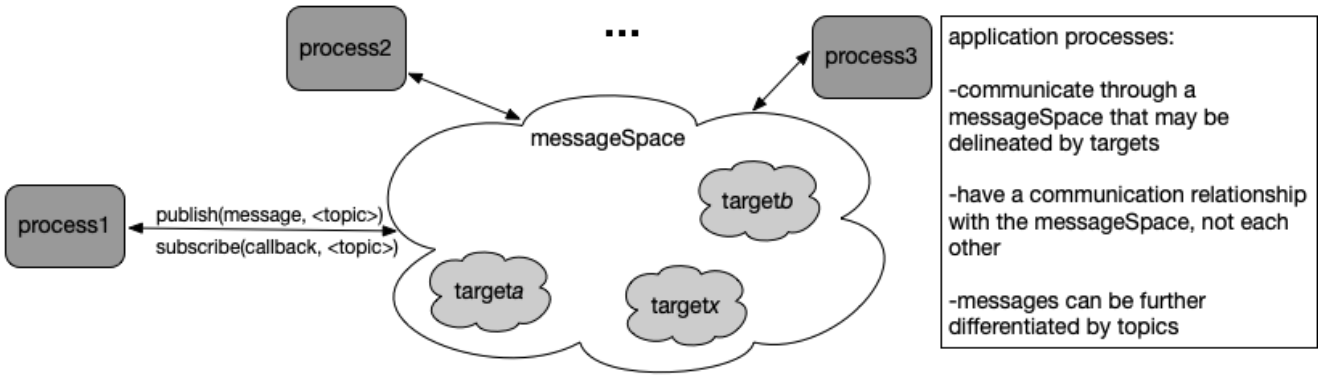
Message-Based Publish/Subscribe (MBPS)

MBPS is a data-centric bespoke transport that presents applications with an MQTT-inspired interface. This directory contains three example applications using a message-based publish/subscribe (MBPS) transport. Each example uses a customized "shim", **mbps.hpp,** that implements an MBPS client and presents an API based on *messages* as an application level unit of information. The API is inspired by MQTT client APIs, but MBPS is *brokerless*. Three (just mbps0.hpp at present) examples of MBPS use are supplied: the interface to application processes stays identical for all versions while each provides different levels of security. The mbpsv*n*.hpp shims have minor differences while the trust schemas used to compose and validate packets are quite different.

An MBPS client presents a messageSpace where processes can publish and subscribe to messages in a particular application domain which may be further limited to a specific target within that domain.


process2
...
process3
messageSpace
process1
publish(message, <topic>)
subscribe(callback, <topic>)
target*b*
target*a*
target*x*
application processes:
-communicate through a messageSpace that may be delineated by targets
-have a communication relationship with the messageSpace, not each other
-messages can be further differentiated by topics

Messages for a target may be further specified with topic information and subscriptions can be particularized by these topics or may just subscribe to all messages in the MBPS client's messageSpace or messageSpace target.

Build notes

MBPS works on Macs and Linux and uses IPv6 multicast. Note that this uses the ndn-ind library (https://github.com/operantnetworks) and, for broadcast performance, NFD version with the the *no-nacks-on-multicast-faces* and s*hip-pending-interests-on-register* NFD patches as well as ndn-cxx patch (for keys) available at: https://github.com/pollere/NDNpatches. [Also must fix the bug that causes rejection of registrations within the same millisecond before can use key distributors.] After cloning the DCT repo, the example programs can be used. Use the utilities in the tools directory to first compile the schema, for mbps0:

```
schemaCompile -o mbps0.scm mbps0.schema
```

and then install the trust schema in the PIB using:

```
schema_install mbps0.scm
```

An 'ndnsec list' should show the schema in the PIB. After this, it should be possible to do a 'make' in your examples/mbps directory.

Using the examples

The **mbps** directory contains three applications (app1.cpp, app2.cpp, and app3.cpp) and three versions of mbps{0,1,2}.hpp. [**NOTE**: the currently only 0 is released.] The application code is meant to provide simple examples that can be extended and customized for other purposes. Each version's mbps{0,1,2}.hpp shim, coupled with its trust schema (mbps0.schema), implements a different security model while the applications are identical. An MBPS client can be created with a set target value or, if not set, the client will handle all the messages in its domain, the approach in these examples. An MBPS client is a client of a *distributed application* not of a *server* (or broker), and its implementation reflects that. For example, in MQTT, a message of type CONNECT is the first packet sent from the client to the server. To generate this message, MQTT applications usually call a client *connect* method, passing in a callback function to handle a CONNACK message from the server. Since MBPS is serverless, its *connect* method is used to do all initialization required in order to communicate (acquire keys, etc) and a passed-in handler function should run once communications become possible. In the examples, that handler subscribes to to all the messages with a single callback function *msgRecv*, creates the first message body and calls *msgPubr* which uses the client's publish function and schedules another message to be sent. The MBPS client's *run*() is called after *connect*(). app2.cpp's *msgPubr* does not schedule another message, instead the *msgRecv* method schedules a new message to be published after a short delay (and only if there's not another message pending in order to avoid an explosion of messages). The files also have examples of using the message confirmation (roughly similar to MQTT QoS 1).

Running examples

The host must be running an NDN Forwarding Daemon (with Pollere's patches) and must have a localnet route to the IPv6 multicast Face. At least two application entities should be started or nothing will happen. (For now, do a "nfdc cs erase /localnet" before starting a new session.) app1 publishes the given number of messages, sending a short message each time. If the -p option is set, the application will persist waiting for more messages.

Example usage:

`nfdc cs erase /localnet`

`app1 -c 10` *or* `app1 -c 10 -p`

app2 publishes an initial message and waits to receive a message before scheduling another message publication at a short delay.

Example usage:

`app2`

app3 randomizes some publication components and removes the confirmation callbacks of app2.

Example usage:

app3

In addition to a message body, mbps.hpp's *msgArgs* struct can be used to exchange additional information about a message with the MBPS client that can be used by the MBPS transport.

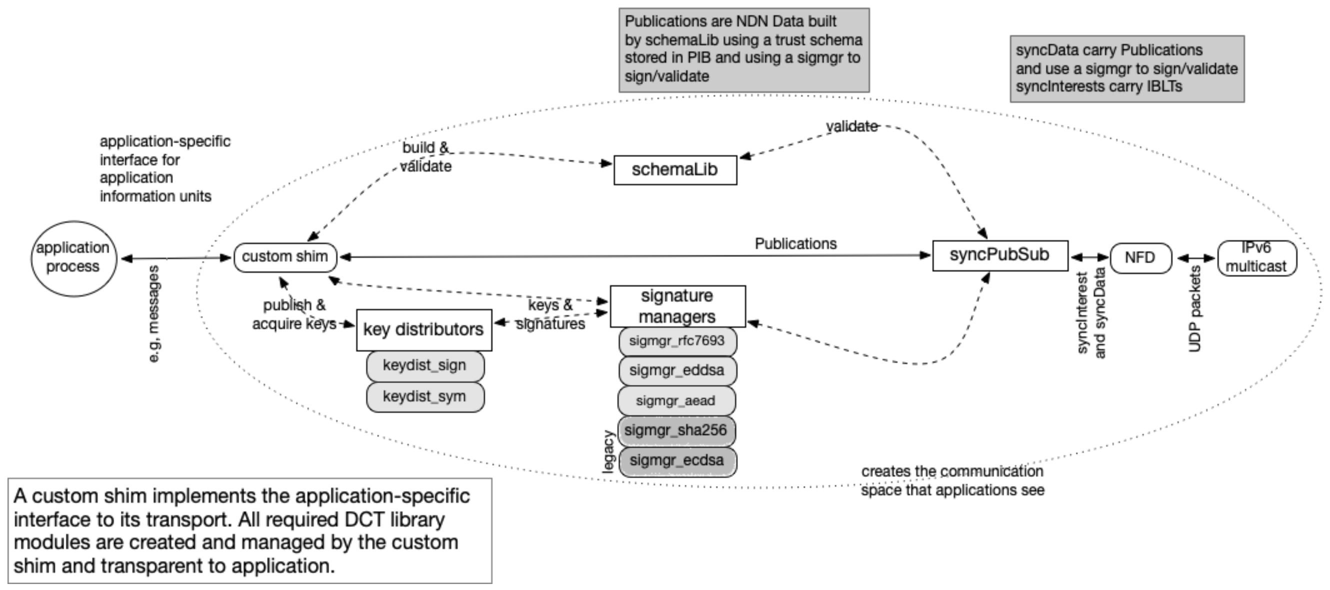
```
struct msgArgs {
    msgArgs() {}
    bool dup{0};            //may be used to indicate duplicate
    std::string cap{};      //capability
    std::string loc{};      //location
    std::string topic{};    //message type
    std::string args{};     //specifiers
    std::string ts{};       //message creation time
};
```

The MBPS transport is described in more detail below in the Elements of the MBPS Bespoke Transport section.

Elements of the MBPS Bespoke Transport

The MBPS transport implements its messageSpace using modules from Pollere's Data-Centric Toolkit which uses the NDN Forwarding Daemon (NFD) to provide network layer primitives over an IPv6 multicast network. This section provides an overview of the module interaction. This is an area of active work, that is, we are still refining how the "pieces" go together; this is the current snapshot. The modules that uniquely define a particular transport are the shim (here mbps0.hpp) and the trust schema (mpbs0.schema) and the rest are DCT library modules. MBPS's shim translates application messages into secured Publications (and Publications to messages) that are passed to and from a *SyncPubSub* object (in syncps.hpp) that manages the set synchronization of a particular Collection of Publications using NDN primitives (syncInterests and syncData). Publications are NDN Data which have distinct name components and a data portion that holds the message or a segment of a large message. (Publications are encapsulated in NDN Sync Data, which are exchanged with NDN forwarders, by SyncPubSub . Since the focus is on broadcast communications, those go to the **localnet** Face.)

The figure shows how the various modules of DCT are used by a custom shim (like mbps.hpp) to make a bespoke transport that presents an application-specific interface and uses NFD and its IPv6 multicast interface to send and receive packets on a local network.


Publications are NDN Data built by schemaLib using a trust schema stored in PIB and using a sigmgr to sign/validate
syncData carry Publications and use a sigmgr to sign/validate syncInterests carry IBLTs
validate
build & validate
schemaLib
application-specific interface for application information units
application process
custom shim
Publications
syncPubSub
NFD
IPv6 multicast
e.g. messages
publish & acquire keys
key distributors
keys & signatures
signature managers
keydist_sign
keydist_sym
sigmgr_rfc7693
sigmgr_eddsa
sigmgr_aead
sigmgr_sha256
sigmgr_ecdsa
legacy
syncInterest and syncData
UDP packets
creates the communication space that applications see

A custom shim implements the application-specific interface to its transport. All required DCT library modules are created and managed by the custom shim and transparent to application.

MBPS Publication Names

Each syncPubSub handles the synchronization of Publications within its Collection, which may encompass an entire application domain. Alternatively, a Collection may be further specified by a target. Publication names are constructed according to the following table, where the component structure can be used for hierarchical topic and subtopic handling and the individual components are specified in, and can be secured through, the domain's trust schema.

component	description
networkID	e.g., campus/main_building (multiple components for some deployments)
domain	the top-level name for application communications, here **mbps**
target	a capability, attribute, property
location	target location, e.g. device(s), room, local, component (smartthings)
topic	gives the type of message
arguments	more specific information for this topic
origin	identifies the publisher: role-specific ID and deviceID
mesgID	unsigned 32 bit number to uniquely identify each message
sCnt	unsigned 16 bit integer; indicates if single publication message or multiples (see below)
mtimestamp	time mbps client receives message from application process in UTC microsecs

A target is the "audience" for the Publication. This can be used in many ways, but one (IoT-centric) way to think of it is as a *capability* e.g., lock, alarm, colorControl, motionSensor. The target location can be a particular device, class of device, component of a device, a room, a vicinity. Similarly, topic can be used in many ways, for example to indicate the type of message carried, e.g. an event, a command, or a status. The arguments can supply any additional information associated with that topic. The origin component can be used to identify the publisher, mesgID is set to a hash in mbps.hpp. sCnt is packaged in an unsigned 16 bit integer which, if not 0 to indicate a single publication message, uses the upper 8 bits for the sequence of this piece within the total message and the lower 8 bits for the total number of pieces in the message. This means that any message must fit into 255 publication segments (so byte-limited to maximum content size times 255) but the current shims limit the number of segments to 64 for an efficient implementation and for the iblt implementation, the limit should be kept under 80.

Confirmation of Publications ("QoS")

MBPS provides mechanisms that enable applications to set a callback (if desired) indicating whether its message made it into the messageSpace. This relies on mechanisms in syncPubSub that can check that its Publication was seen on a Sync Interest issued by another process and methods in the shim that ensure all the Publications of any particular message have been confirmed. This gives an indication of message publication "success" roughly analogous to MQTT's QoS 1. It is invoked at the application by passing a callback to the mbps.hpp publish() method as the third argument. If the callback argument is not used, the option is not employed by the client. (For some purposes, a particular shim will employ the mechanism, for example, for key distribution in the connect method.)

mbps0.hpp

An entity has to have the correct identity and the trust schema in order to build and validate Publications. Publications and syncData are signed for integrity checking using a SHA256 hash. The trust schema in mbps0.schema allows for two roles, device and operator. The applications will have a device role by default, but the optional argument -r *role* can be used to designate a role of "operator". In this simple schema, an operator role is required to publish **command** messages and a "device" role is required to publish **status** and **event** messages.

mbps1.hpp

During connect(), each MBPS client publishes its signing cert chain and subscribes to that of others in the domain. Publications are integrity signed as in 0 but the syncData are signed with the secret key of their published signing certificate.

mbps2.hpp

During connect(), the MBPS client publishes its signing cert chain and subscribes to that of others in the domain. Any client that has the capability (role) to create and sign the symmetric group keys attempts to become the symmetric key creator for the domain. (The simple self-selection algorithm used here could certainly be improved upon but is provided as an illustration.) Once a valid symmetric key is obtained, messages can be encrypted/decrypted so the connect callback is invoked.
