

NDN-DTN integration

Document title: Documentation on the code

Author: Dimitris Vardalis

NDN consists of:

1. *ndn-cxx* library, which provides infrastructure facilities for the system,
2. *ndn*, which includes the actual forwarding *daemon* (NFD) and several related tools

The following contributions have been made to realise the NDN-DTN integration:

- Library *ndn-cxx_umobile* contributions:
 1. Extended the `FaceURI` class so that it can handle DTN faces.
 - a. Added a "dtn" scheme that takes as the `endpointPrefix` as the host (e.g. `dtn-node1`) and the `endpointAffix` as the path (e.g. `/nfd`)
 - b. Added a `DTNCanonizeProvider` that checks if a certain dtn *URI* is canonical. Currently all *URIs* are considered canonical.
- *Daemon ndn-dtn* contributions:
 1. Added section in the *FaceManager* that initializes the DTN Face from the *nfd.conf* configuration file. The configuration allows for setting the host and port for the *ibrdsn* daemon and the endpoint prefix and affix for *ibrdsn*.
 - a. Normally the *ibrdsn daemon* host is the localhost, but a remote host could also be used as a gateway
 - b. The *endpoint id* of the local dtn host is in the form of `dtn://dtn-node1/nfd`, where "dtn" is the scheme, "dtn-node1" is the `endpointPrefix` and `nfd` is the `endpointAffix`. `nfd` is the dtn application endpoint that the *nfd daemon* will subscribe to in order to receive ndn-related bundles.

2. Created `DtnFactory` class, responsible for creating and maintaining the `DtnChannel`. When the *daemon* initializes, a `DtnChannel` is created. Subsequent invocations of the `createChannel` function of the `DtnChannel` return the already created channel.
3. Created an `AsyncIbrDtnClient` for asynchronously connecting to the *ibrdsn daemon*. The client starts a new thread that runs in the background and is always connected to the *dsn daemon*. When a new bundle arrives, the client notifies the associated `DtnChannel`.
4. Created `DtnChannel` that receives and sends data. `DtnChannel` listens for incoming data and forwards them to the *daemon* through the relevant face and also sends outgoing data, again through the relevant face.
 - a. Each `DtnChannel` creates an `AsyncIbrDtnClient` in the "listen" function and passes itself as a constructor argument. When a new bundle arrives the client calls the relevant callback at the `DtnChannel`. The client thread remains on for the entire lifetime of the channel and gets deleted in the channel destructor.
 - b. Handling of incoming bundles is queued in a global *daemon* event queue. Instead of the `AsyncIbrDtnClient` calling directly the `DtnChannel` callback it posts the task of running the callback with the appropriate arguments to the global I/O queue of the *daemon*. This way thread synchronization issues between the bundle receiving thread and the core *nfd* thread are avoided.
 - c. `DtnChannel` differs from the rest of the *ndn* channels in that it receives all bundles for all faces, even if bundles for the same destination have appeared earlier. Other *ndn* channels pass connected sockets to the relevant face, which then directly receives any other bundles that may arrive at this socket, bypassing the `DtnChannel`.
 - d. `DtnChannel` creates faces similarly to the way datagram channels work. The link service used is the `GenericLinkService` and the transport is

a special `DtnTransport` described below. When the `processBundle` function is invoked by the scheduler, a face is created (or just retrieved if it already exists) and the bundle is passed on to the transport of the face.

5. Created `DtnTransport` that inherits the `Transport` base class. The option to use the datagram transport template class was ruled out because it is too closely tied to actual IP sockets, whereas the `DtnTransport` sends and receives bundles via the `ibrdtn` client.
 - a. Created a `receiveBundle` function that is invoked by the `DtnChannel` and accepts an incoming bundle. The function reads the payload from the incoming bundle, copies the DTN payload to an NDN block element and inserts the block element into a new `NDN Transport::Packet`. Finally, the `Transport::receive` function is called with the newly created transport packet, which propagates the packet into the NDN core for processing.
 - b. Implemented the virtual `doSend` function, which receives a `Transport::Packet`, creates a bundle, and copies the contents of the packet into the bundle. The bundle destination address is set to the `remoteUri` of the face (a face is created for each remote peer). Finally, the `ibrdtn` bundle is sent through an `ibrdtn Client`. The client is created, connected to the `ibrdtn` daemon, fed the bundle and closed before the function exits. Contrary to the `AsyncIbrDtnClient`, which is constantly running in the background, the Client created for sending the bundle has only local scope.