Adding custom fields to packets in ndnSIM 2.3 without forking the entire repository.

Posted on March 16, 2017

The recommended way to build something on top of ndnSIM is to fork its scenario template (https://github.com/cawka/ndnSIM-scenario-template) repository and work inside there. You still need to download and compile (http://ndnsim.net/2.3/getting-started.html) the actual framework, however you will simply install it into `/usr/local` and link to it instead of actually working inside the main repository.

It turns out that this workflow actually makes certain tasks a lot more difficult. You might think a network simulator would make it easy to add new header fields to packets. Well, think again.

First Steps

What do we want to do? Our goal is to just add one field to the Interest packet header. The ndn::Interest (http://ndnsim.net/2.3/doxygen/classndn_1_1Interest.html) class inherits an interface called ndn::TagHost (http://ndnsim.net/2.3/doxygen/classndn_1_1TagHost.html), which allows you to attach arbitrary tags to it. Defining your own tag can be as simple as a single typedef, if you only need to contain a single value in that tag:

```
typedef ndn::SimpleTag<uint64_t, 0x60000001> MyCustomTag;
```

You simply specify the type of the tag and make up an ID for it. However, you must pick an unused tag from the valid range (https://redmine.named-data.net/projects/ndn-cxx/wiki/PacketTagTypes) given in the ndn-cxx wiki. My `0x60000001` is the first value in this range.

To attach a tag to an Interest, you simply call the `setTag` method:

```
interest.setTag<MyCustomTag>(std::make_shared<MyCustomTag>(54321));
```

To read a tag from an Interest, there is a corresponding `getTag` method:

```
std::shared_ptr<MyCustomTag> tag = interest.getTag<MyCustomTag>();
```

This gives you a pointer to the tag object, and you can get the value out of it quite easily... But first, check if it is null.

```
if (tag == nullptr) {
    // no tag
}
else {
    uint64_t tagValue = tag->get();
}
```

However, now is where we encounter our problem. Our tag *will not actually be encoded and sent over the network*. That's right – we can attach a tag to the Interest, but when it arrives at the next hop *it will be gone*.

How can we fix this?

Investigation

Vanilla ndnSIM uses these sorts of tags itself in a few places. One obvious one is the HopCountTag, which you can use to figure out how far a packet has gone in the network. A grep through the ndnSIM source brings us to a class called GenericLinkService (http://ndnsim.net/2.3/doxygen/classnfd_1_1face_1_1GenericLinkService.html). This class is responsible for actually encoding packets and sending them out on the wire. In particular, we can find the bit responsible for encoding the HopCountTag in a method called `encodeLpFields`:

```
shared_ptr<lp::HopCountTag> hopCountTag = netPkt.getTag<lp::HopCount
Tag>();
if (hopCountTag != nullptr) {
    lpPacket.add<lp::HopCountTagField>(*hopCountTag);
}
else {
    lpPacket.add<lp::HopCountTagField>(0);
}
```

Clearly, we need to define a MyCustomTagField to be able to encode our new tag.

Declaring a Tag

This is actually pretty easy, but first you need to know what kind of witchcraft is going on. Let's start with the actual code to define the field, then go on to analyze it:

```
enum {
 TlvMyCustomTag = 901
};

typedef ndn::lp::detail::FieldDecl<ndn::lp::field_location_tags::Hea
der, uint64_t, TlvMyCustomTag> MyCustomTagField;
```

First, we define a constant for the TLV type ID... There are actually a few hidden constraints to what we can pick. If we don't do this right, we get a packet parse error. Why?

Let's look at `ndn::lp::Packet`'s `wireDecode` method:

```
for (const Block& element : wire.elements()) {
    detail::FieldInfo info(element.type());
    if (!info.isRecognized && !info.canIgnore) {
        BOOST_THROW_EXCEPTION(Error("unrecognized field cannot be ig
nored"));
    }
    ...
}
```

Apparently, this FieldInfo (http://ndnsim.net/2.3/doxygen/classndn_1_1lp_1_1detail_1_1FieldInfo.html) class tells the decoder whether the field is recognized, and whether it can ignore it if it isn't. Let's peek at the constructor:

```
FieldInfo::FieldInfo(uint64_t tlv)
: ...
{
   boost::mpl::for_each<FieldSet>(boost::bind(ExtractFieldInfo(), th
is, _1));
   if (!isRecognized) {
     canIgnore = tlv::HEADER3_MIN <= tlvType
                 && tlvType <= tlv::HEADER3_MAX
                 && (tlvType & 0x01) == 0x01;
  }
}
```

Now this is interesting... To figure out what a TLV tag is, it iterates over FieldSet (which only contains the built-in tags, and we can't override). However, if it doesn't find a match, it determines if it is ignorable based on the value of the TLV type ID. We can't make the field recognized without forking the actual ndnSIM core, but we can make it ignorable by choosing the right ID.

To save you from looking up `tlv::HEADER3_MIN` and `tlv::HEADER3_MAX`, they are 800 and 959, respectively. Also, don't forget that the low bit has to be set. And don't pick one of the types that is already used (http://ndnsim.net/2.3/doxygen/lp_2tlv_8hpp_source.html).

Moving on from the TLV ID nonsense, the rest of the FieldDecl is pretty straightforward. We pass a flag that says "this goes in the header," followed by the type of the value and the TLV ID we just made up.

Note that for some reason, the code won't compile if the type is specified as anything other than `uint64_t`. I didn't care enough to figure this out, but it seems to have something to do with the fact that the only integer EncodeHelper defined is for `uint64_t` (http://ndnsim.net/2.3/doxygen/field-decl_8hpp_source.html#l00085).

Encoding the Tag

So far, we have defined our tag twice: once for the high-level Interest object, and once for the low-level TLV encoding. Now, we need to write code to convert between these two representations.

To do this, we need to create a new LinkService. Sounds intimidating, but really all we need to do is make a copy of GenericLinkService and change a few things. Yes, literally copy `generic-link-service.hpp` and `generic-link-service.cpp` out of `ns3/ndnSIM/NFD/daemon/face/` and into your own project. Rename the file as you see fit, and carefully rename the class to something like CustomTagLinkService. You will want to be careful because we still need to implement the GenericLinkServiceCounters interface if we don't want to break anything. We can also avoid redefining the nested Options class by using a typedef to import it from GenericLinkService into the new CustomTagLinkService namespace.

Now that we have an identical clone of the GenericLinkService, let's fix it. To encode your new field, take a look at the `encodeLpFields` method. Follow the pattern used by the CongestionMarkTag field to implement your new custom one:

```
shared_ptr<MyCustomTag> myCustomTag = netPkt.getTag<MyCustomTag>();
if (myCustomTag != nullptr) {
    lpPacket.add<MyCustomTagField>(*myCustomTag);
}
```

Then, add the corresponding decoding logic to `decodeInterest`:

```
if (firstPkt.has<MyCustomTagField>()) {
    interest->setTag(make_shared<MyCustomTag>(firstPkt.get<MyCustomT
agField>()));
}
```

Add the same code to the decodeData and decodeNack methods if you need them.

Using the LinkService

Specifying a custom LinkService isn't going to do us any good if we don't tell ndnSIM to use it. We'll have to replace the callback that sets up a Face in order to do this. We're going to focus on Faces for PointToPointNetDevices, but the following can be generalized for other types of links.

The call from our scenario file will look something like this:

```
stackHelper.UpdateFaceCreateCallback(
    PointToPointNetDevice::GetTypeId(),
    MakeCallback(CustomTagNetDeviceCallback)
);
```

For context, this is a method of the StackHelper (http://ndnsim.net/2.3/doxygen/classns3_1_1ndn_1_1StackHelper.html) that you're probably already using to install the NDN stack on nodes. To write the callback, copy the logic from the PointToPointNetDeviceCallback (http://ndnsim.net/2.3/doxygen/ndn-stack-helper_8cpp_source.html#l00295) in that same class. All you have to change is the instantiation of the LinkService – replace the GenericLinkService with your own. You will also need to copy the constructFaceUri method (verbatim) because your callback will need to refer to it, but it is out of scope.

Other Caveats

By default, the scenario template wants to compile your code in C++11 mode. However, the LinkService uses some C++14 features, so you'll have to edit the flags in `.waf-tools/default-compiler-flags.py`. Note that you need to re-run `./waf`

`configure` if you edit these flags.

Conclusion

I think this is way too much effort just to add a field to a packet. We've duplicated a lot of logic in order to do something so small. I feel like the ndnSIM developers should have made it a bit easier to add fields to a packet... At worst, I might expect a call to the StackHelper to add new fields. It would likely be possible to write a generic enough LinkService which will encode any custom fields as long as mappings between the TLV classes and tag classes are provided. I look forward to this feature, because it would have made the middle part of my week go a lot more smoothly. Until then, I hope that this post can be useful to anyone else trying to do the same thing.

← **PREVIOUS POST (/POSTS/POSTGRESQL-FULLTEXT-SEARCH/)**

NEXT POST → (/POSTS/GITLAB-CALENDAR/)