ILNP DNS processing at the IETF 105 hackathon

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

First publication of this article on 23 July 2019

https://www.bortzmeyer.org/hackathon-ietf-105.html

The weekend of 20-21 July 2019, in Montréal, was the hackathon preceding the IETF 105 meeting <https://www.ietf.org/how/meetings/105/>. During this hackathon, I helped to improve the DNS support for the ILNP protocol, in the Knot DNS resolver <https://www.knot-resolver.cz/>.

ILNP is an **identifier/locator separation** network protocol. Each machine has at least one **identifier** such as `0:0:c:1`, which is stable (never changes), and at least one **locator** such as `2001:8b0:d3:cc22`, which may vary often, for instance when the machine moves from one network to another, or when the network is renumbered. The idea is to give stable identities to the machine, while keeping locators for efficient routing. If the locator looks like an IPv6 address, it is not a coincidence, it is because you can use ILNP locators everywhere you would use an IP address (more on that later). Each identifier/locator separation system requires a **mapping** service to get the locator from the identifier. With ILNP, this mapping service is the old and proven DNS. As the saying goes, "you can solve every problem in computer science just by adding one level of indirection".

ILNP is described in RFC 6740[1] and the DNS resource records on RFC 6742. These resource records are `NID` (find the identifier from the domain name) and `L64` (find the locator from the domain name). I simplify, there are other records but here are the two I worked with. You can see them online yourself :

```
% dig NID ilnp-aa-test-c.bhatti.me.uk
...
;; ANSWER SECTION:
ilnp-aa-test-c.bhatti.me.uk. 10 IN NID 10 0:0:c:1
ilnp-aa-test-c.bhatti.me.uk. 10 IN NID 20 0:0:c:2
ilnp-aa-test-c.bhatti.me.uk. 10 IN NID 30 0:0:c:3
...
```

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6740.txt

```
% dig L64 ilnp-aa-test-c.bhatti.me.uk
...
;; ANSWER SECTION:
ilnp-aa-test-c.bhatti.me.uk. 10 IN L64 10 2001:8b0:d3:cc11
ilnp-aa-test-c.bhatti.me.uk. 10 IN L64 20 2001:8b0:d3:cc22
...
```

ILNP could work just like that. RFC 6742, section 3.2, just recommends that the name servers do some additional processing, and return other possibly useful ILNP records when queried. For instance, if `NID` records are asked for, there is a good chance `L64` records will be needed soon, so the server could as well find them and return them in the additional section of the DNS response (in the two examples above, there is only the answer section).

Current name servers don't do this additional processing. Anyway, this weekend, we went a bit further, implementing an option which is not in the RFC. Following an idea by Saleem Bhatti, the goal was to have the DNS resolver return ILNP records when queried for an `AAAA` record (IPv6 address). This would allow unsuspecting applications to use ILNP without being modified. The application would ask the IP address of another machine, and a modified name resolution library would get the ILNP records and, if they exist, return to the application the locator instead of the "normal" IP address (remember they have the same syntax).

Now, let's see how to do that in the Knot resolver `<https://www.knot-resolver.cz/>`. Knot has a system of modules that allow to define specific processing for some requests, without modifying the main code. (I already used that at the previous hackathon `<https://www.bortzmeyer.org/hackathon-ietf-104.html>`.) Modules can be written in Lua or C. I choosed C. We will therefore create a `ilnp` module in `modules/ilnp`. A module can be invoked at several "layers" during the processing of a DNS request. We will use the `consume` layer, where the request has been received but not all the answers are known yet :

```
static int ilnp_consume(kr_layer_t *ctx, knot_pkt_t *pkt) {
   ...
}

KR_EXPORT
int ilnp_init(struct kr_module *module) {
static kr_layer_api_t layer = {
.consume = &ilnp_consume,
};
```

The entire module is 83 lines, including test of return values, logging, comments and empty lines. Now, if the module is loaded, Knot will invoke `consume()` for every DNS request. We need to act only for `AAAA` requests :

```
struct kr_request *req = ctx->req;
if (req->current_query->stype == KNOT_RRTYPE_AAAA) {
...
}
/* Otherwise, do nothing special, let Knot continue. */
```

And what do we do when we see the AAAA requests? We ask Knot to add a sub-request to the current requests (two, actually, for `NID` and `L64`). And we need also to check if the request is a sub-request or not. So we add some flags in `lib/rplan.h` :

```
struct kr_qflags {
...
bool ILNP_NID_SUB : 1;        /** NID sub-requests for ILNP (ilnp module) */
bool ILNP_L64_SUB : 1;        /** L64 sub-requests for ILNP (ilnp module) */
```

And we modify the code to add the sub-requests (`kr_rplan_push()`, here I show only the `NID` one) and to test them (`req->options.ILNP_..._SUB`) :

```
if (req->current_query->stype == KNOT_RRTYPE_AAAA && !req->options.ILNP_NID_SUB && !req->options.ILNP_L64_SUB)
 {
next_nid = kr_rplan_push(&req->rplan, req->current_query,
                     req->current_query->sname,  req->current_query->sclass,
                     KNOT_RRTYPE_NID);
next_nid->flags.ILNP_NID_SUB = true;
next_nid->flags.DNSSEC_WANT = true;
                next_nid->flags.AWAIT_CUT = true;
 }
```

And, when the sub-request returns, we add its answers to the additional section (`array_push()`, again, I show only for `NID`) :

```
 else if (req->current_query->stype == KNOT_RRTYPE_NID && req->current_query->flags.ILNP_NID_SUB) {

for (i=0; i<req->answ_selected.len; i++) {
if (req->answ_selected.at[i]->rr->type == KNOT_RRTYPE_NID) {
array_push(req->additional, req->answ_selected.at[i]->rr);
}
}
}
```

To test that, we have to load the module (people not interested in ILNP will not run this code, which is good for them, because the extra queries take time) :

```
modules = {'hints', 'view', 'nsid', 'ilnp'}
```

And Knot now gives us the ILNP records in the additional section :

```
% dig @MY-RESOLVER AAAA ilnp-aa-test-c.bhatti.me.uk
...
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 53906
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 6
...
;; ANSWER SECTION:
ilnp-aa-test-c.bhatti.me.uk. 10 IN AAAA 2001:8b0:d3:1111::c

;; ADDITIONAL SECTION:
ilnp-aa-test-c.bhatti.me.uk. 10 IN L64 10 2001:8b0:d3:cc11
ilnp-aa-test-c.bhatti.me.uk. 10 IN L64 20 2001:8b0:d3:cc22
ilnp-aa-test-c.bhatti.me.uk. 10 IN NID 10 0:0:c:1
ilnp-aa-test-c.bhatti.me.uk. 10 IN NID 20 0:0:c:2
ilnp-aa-test-c.bhatti.me.uk. 10 IN NID 30 0:0:c:3

;; Query time: 180 msec
;; MSG SIZE  rcvd: 194
```

A possible option would have been to return the ILNP records only if they are already in the cache, improving performances. (Today, most machines have no ILNP records, so querying them takes time for nothing.) But, as far as I know, Knot does not provide an easy way to peek in the cache to see what's in it.

I also worked on the Unbound resolver. Unbound is not modular so modifications are more tricky, and I was unable to complete the work. Unbound, unlike Knot, allows you to see what is in the cache (added in `daemon/worker.c`) :

```
rrset_reply = rrset_cache_lookup(worker->env.rrset_cache, qinfo.qname,
   qinfo.qname_len,
   LDNS_RR_TYPE_NID, LDNS_RR_CLASS_IN, 0, *worker->env.now, 0);
if (rrset_reply != NULL) {
    /* We have a NID for sldns_wire2str_dname(qinfo.qname,
    256)) */
    lock_rw_unlock(&rrset_reply->entry.lock); /* Read the
    documentation in the source: it clearly says it is locked
    automatically and you have to unlock when done. *
}
```

But NID and other ILNP records are not put into the cache. This is because Unbound does not follow the caching model described in RFC 1034. For performance reasons, it has two caches, one for entire DNS messages and one for resource records. The first one is used for "normal" DNS queries (for instance a `TXT` record), the second for information that may be needed to answer queries, such as the IP addresses of name servers. If you query an Unbound resolver for a TXT record, the answer will be only in the message cache, not in the resource record cache. This allows Unbound to reply much faster : no need to construct an answer, just copy the one in the cache. But it makes retrieval of data more difficult, hence the resource record cache. A possible solution would be to put ILNP records in the resource record cache (in `iterator/iter_scrub.c`) :

```
#ifdef ILNP
if (rrset->type == LDNS_RR_TYPE_NID || rrset->type == LDNS_RR_TYPE_L64 ||
    rrset->type == LDNS_RR_TYPE_L32 || rrset->type == LDNS_RR_TYPE_LP) {
  store_rrset(pkt, msg, env, rrset);
}
#endif
```

But, as I said, I stopped working on Unbound, lack of time and lack of brain.

Many thanks to Ralph Dolmans and Petr [Caractère Unicode non montré [2]]pa[Caractère Unicode non montré]ek for their help while I struggled with Unbound and Knot Resolver.

2. Car trop difficile à faire afficher par LaTeX