Contents

1 Address resolution protocol

The *address resolution protocol* or **ARP** is a standalone module which performs the mapping of a given layer-3 address $addr_{NL}$ to another address, the link-layer address, which we will call $addr_{LL}$.

1.1 Why do we need this?

The reason that we require this $addr_{LL}$ is because when we need to send data to a host we do so over a link which is indicated in the routing table for said packet.

However, links don't speak the same network-layer protocol as twine - they speak whatever protocol they implement - i.e. Ethernet via `LIInterface` or the in-memory `PipedLink`. Needless to say there is also always a requirement of such a mapping mechanism because several links may be backed by a different link-layer protocols in their `Link` implementation and therefore we cannot marry ourselves to only one link-layer protocol - *we need something dynamic.*

1.2 The mapping function

We now head over to the technical side of things. Before we jump directly into an analysis of the source code it is worth considering what this procedure means in a mathematical sense because at a high-level this is what the code is modeled on.

If we have a router r_1 which has a set of links $L = \{l_1, l_2\}$ and we wanted to send a packet to a host addresses h_1 and h_2 which are accessible over l_1 and l_2 respectively then the mapping function would appear as such:

$$(h_1, l_1) \rightarrow addr_{LL_1}$$
$$(h_2, l_2) \rightarrow addr_{LL_2}$$

On the right hand side the $addr_{LL_1}$ and $addr_{LL_2}$ are the resolved link-layer addresses.

$$(h_i, l_i) \rightarrow addr_{LL_i}$$

Therefore we discover that we have the above mapping function which requires the network-layer h_i address you wish to resolve and the link l_i over which the resolution must be done, this then mapping to a single scalar - the link-layer address, $addr_{LL_i}$.

1.3 Implementation

We will begin the examination of the code at the deepest level which models this mathematical function earlier, first, after which we will then consider the code which calls it and how that works.

1.3.1 The entry type

Firstly let us begin with the definition of the in-memory data type which holds the mapping details. this is known as the `ArpEntry` struct and it is shown in part below:

```
public struct ArpEntry
{
    private string l3Addr;
    private string l2Addr;

    ...

    public bool isEmpty()
    {
        return this.l3Addr == "" && this.l2Addr == "";
    }

    public static ArpEntry empty()
    {
        return ArpEntry("", "");
    }
```

Please note the methods `isEmpty()`. An entry is considered empty if both its network-layer and link-layer fields have an empty string in them, this is normally accomplished by calling the `empty()` static method in order to construct such an `ArpEntry`.

1.3.2 Making an ARP request

The code to make an ARP request is in the `regen(Target target)` method and we will now go through it line by line.

1.3.2.1 Setting up the request and link Firstly we are provided with a `Target`, this is encapsulates the network-layer address and the `Link` instance we want to request over. We now extract both of those items into their own variables:

```
// use this link
Link link = target.getLink();

// address we want to resolve
string addr = target.getAddr();
```

Before we make the request we will need a way to receive the response, therefore we attach ourselves, the `ArpManager`, as a `Receiver` to the link:

```
// attach as a receiver to this link
link.attachReceiver(this);

logger.dbg("attach done");
```

This provides us with a callback method which will be called by the `Link` whenever it receives *any* traffic. It is worth noting that such a method will not run on the thread concerning the code we are looking at now but rather on the thread of the `Link`'s - we will discuss later how will filter it and deliver the result to us, *but for now - back to the code.*

1.3.2.2 Encoding and sending the request Now that we know what we want to request and over which link we can go ahead and encode the ARP request message and broadcast it over the link:

```
// generate the message and send request
Arp arpReq = Arp.newRequest(addr);
Message msg;
if(toMessage(arpReq, msg))
{
    link.broadcast(msg.encode());
    logger.dbg("arp req sent");
}
else
{
    logger.error("Arp failed but oh boy, at the encoding level");
}
```

As you can see we make use of the `broadcast(byte[])` method, this is handled by the link's implementation according to its link-layer protocol.

1.3.2.3 Waiting for a response We now have to wait for a response and not just any response. It has to be an ARP reply for the particular network-layer address we requested earlier.

This is done with the following code:

```
// wait for reply
string llAddr;
bool status = waitForLLAddr(addr, llAddr);

...
```

As you can see we have this call to a method called `waitForLLAddr(addr, llAddr)`. This method will block for us and can wake up if it is signaled to by the callback method running on the `Link`'s thread (as mentioned previously).

```
StopWatch timer = StopWatch(AutoStart.yes);

// todo, make timeout-able (todo, make configurable)
while(timer.peek() < this.timeout)
{
    this.waitLock.lock();

    scope(exit)
    {
        this.waitLock.unlock();
    }

    this.waitSig.wait(dur!("msecs")(500)); // todo, duty cycle if missed notify but also hel

    // scan if we have it
    string* llAddr = l3Addr in this.addrIncome;
    if(llAddr !is null)
    {
        string llAddrRet = *llAddr;
        this.addrIncome.remove(l3Addr);
        llAddrOut = llAddrRet; // set result
        return true; // did not timeout
    }
}

return false; // timed out
```

Because it is implemented using a condition variable, it could potentially miss a signal from the calling `notify()` if we only call `wait()` on our thread *after* the link's thread has called `notify()`. Therefore, we make our `wait()` wake up every now and then by using a timed-wait, to check if the data has been filled in by the other thread.

Second of all, what we do after retrying from `wait(Duration)` is check if the *requested network-layer address* has been resolved or not - this is that filtering I was mentioning earlier. This is important as we don't want to wake up for *any* ARP response, but only the one which matches our `addr` requested.

Thirdly, this also gives us a chance to check the while-loop's condition so that we can see if we have timed out (waited long enough) for an ARP response.

After all is done, the resulting entry is placed in a globally accessible `string[string] addrIncome` which is protected by the `waitLock` for both threads contending it. We then continue:

```
...

// if timed out
if(!status)
{
    logger.warn("Arp failed for target: ", target);
    return ArpEntry.empty();
}
// on success
else
{
    ArpEntry arpEntry = ArpEntry(addr, llAddr);
    logger.info("Arp request completed: ", arpEntry);
    return arpEntry;
}
```

We now check, as I said, if the entry is valid or not. If we timed-out then we would have returned `false`. Now, as we shall see later, we will still have to return *some* `ArpEntry` because that is the signature of our method, `regen(Target target)`. Thus, if we failed t get an `ArpEntry` we then return one generated by `ArpEntry.empty()`, else we return the actual entry that we received.

1.3.2.4 Catching responses

I have mentioned that the thread which waits for a matching ARP response to come in (the one which calls the `wait(Duration)`) above. So then, the question is - which thread is the one calling `notify()` on the condition variable and under which scenarios?

Recall that we attached the `ArpManager` as a `Receiver` to the `Link` object which was passed into the `regen(Target)` method:

```
// use this link
Link link = target.getLink();

// address we want to resolve
string addr = target.getAddr();

// attach as a receiver to this link
link.attachReceiver(this);

logger.dbg("attach done");
```

Now the reason for this is that whenever traffic is received on a `Link` it will copy the `byte[]` containing the payload to each attached `Receiver`.

This means that the `ArpManager` will receive all packets from a given link, the question is - which ones to we react to? Well that's easy. Below I show you the `onReceive(Link src, byte[] data, string srcAddr)` method which the arp manager overrides. This is called every time a given link receives data:

```
/**
 * Called by the `Link` which received a packet which
 * may be of interest to us
 *
 * Params:
 *   src = the `Link` from where the packet came from
 *   data = the packet's data
 *   srcAddr = the link-layer source address
 */
public override void onReceive(Link src, byte[] data, string srcAddr)
{
    Message recvMesg;
    if(Message.decode(data, recvMesg))
    {
        // payload type
        if(recvMesg.getType() == MType.ARP)
        {
            Arp arpMesg;
            if(recvMesg.decodeAs(arpMesg))
            {
                logger.dbg("arpMesg, received: ", arpMesg, "from: ", srcAddr);
                ArpReply reply;
                if(arpMesg.getReply(reply))
                {
                    logger.info("ArpReply: ", reply);
```

```

                    // place and wakeup waiters
                    placeLLAddr(reply.networkAddr(), reply.llAddr());
                }

               ...
            ...
        ...
    ...
}
```

What we do here is we attempt to decode each incoming packet into our `Message` type, then further check if it is an ARP-typed message. If this is the case then we check if it is an ARP request (because as we have seen, ARP requests are **not** handled here).

```
/**
 * Called by the thread which has an ARP response
 * it would like to pass off to the thread waiting
 * on the condition variable
 *
 * Params:
 *   l3Addr = the network layer address
 *   llAddr = the link-layer address
 */
private void placeLLAddr(string l3Addr, string llAddr)
{
    this.waitLock.lock();

    scope(exit)
    {
        this.waitLock.unlock();
    }

    this.waitSig.notify(); // todo, more than one or never?

    this.addrIncome[l3Addr] = llAddr;
}
```

If this is the case then we will place the link-layer address into a key-value map where the key is the network-layer address and the value is the link-layer address. After this we wake up the sleeping thread by calling `notify()`.

1.3.3 Caching

I mentioned that there is caching involved. The involvement is that all `ArpEntry`'s are stored in a `CacheMap!(ArpEntry)` which means that they will exist in there for some period of time and then be evicted.

If an entry has not yet been cached-in then it is created on demand when you do `map.get(Target)`. Now remember the `regen(Target)` method? Well, thats the regeneration method that we supply this cache map upon instantiation - therefore it works as expected.

1.4 The API

We have now discussed the gritty internals which aid us in creating requests, awaiting replies and then returning the matched entry. We now must move over to the publicly facing API of the `ArpManager`. This really just contains a single method:

```
Optional!(ArpEntry) resolve(string networkAddr, Link onLink)
```

The way this method works is that it will return an `Optional!(ArpEntry)`, meaning that you can test to see if the arp resolution process succeeded or failed (i.e. timed-out for example) using code that looks akin to what shall follow.

I have prepared an example which can illustrate the usage of the `ArpManager`. In fact this example is part of a unittest which tests the various scenarios that can occur with the manager itself.

1.4.1 Mock links

Firstly we setup a pseudo-link. This is a sub-class of the `Link` class which is specifically configured to respond **only** to ARP requests and only to those which a mapping exists for.

In this example I configure two mappings of network-layer addresses to link-layer addresses:

$$(host_{A_{l3}}, dummyLink) \rightarrow host_{A_{l2}}$$
$$(host_{B_{l3}}, dummyLink) \rightarrow host_{B_{l2}}$$

The code to do this is as follows:

```
// Map some layer 3 -> layer 2 addresses
string[string] mappings;
mappings["hostA:l3"] = "hostA:l2";
mappings["hostB:l3"] = "hostB:l2";

// create a dummy link that responds with those mappings
ArpRespondingLink dummyLink = new ArpRespondingLink(mappings);
```

1.4.2 Resolution

We then must create an `ArpManager` we can use for the resolution process:

```
ArpManager man = new ArpManager();
```

Now we are ready to attempt resolution. I first try to resolve the link-layer address of the network-layer address `hostA:l3` by specifying it along with the mock link, `dummyLink`, which we created earlier:

```
// try resolve address `hostA:l3` over the `dummyLink` link (should PASS)
Optional!(ArpEntry) entry = man.resolve("hostA:l3", dummyLink);
assert(entry.isPresent());
assert(entry.get().llAddr() == mappings["hostA:l3"]);
```

In the above case the mapping succeeds and we get an `ArpEntry` returned from `entry.get()`, upon which I extract the link-layer address by calling `llAddr()` on it and comparing it to what I expected, `mappings["hostA:l3"]` - which maps to `hostA:l2`.

We do a similar example for the other host:

```
// try resolve address `hostB:l3` over the `dummyLink` link (should PASS)
entry = man.resolve("hostB:l3", dummyLink);
assert(entry.isPresent());
assert(entry.get().llAddr() == mappings["hostB:l3"]);
```

Lastly, I wanted to show what a failure would look like. With this we expect that `entry.isPresent()` would return `false` and therefore stop right there:

```
// try top resolve `hostC:l3` over the `dummyLink` link (should FAIL)
entry = man.resolve("hostC:l3", dummyLink);
assert(entry.isPresent() == false);
```

This resolution fails because our `ArpRespondingLink`, our *dummy link*, doesn't respond to mapping requests of the kind $(host_{B_{l3}}, dummyLink)$.

1.4.3 Shutting it down

We need to shut down the `ArpManager` when we shut down the whole system, this is then accomplished by running its destructor:

```
// shut down the arp manager
destroy(man);
```

2 Routing and forwarding

Routing is the process by which one announces their routes to others, whilst forwarding is the usage of those learnt routes in order to facilitate the transfer of packets from one endpoint to another through a network of inter-connected routers.

2.1 A route

Before we can get into the process of routing we must first have a conception of a *route* itself.

A route consists of the following items:

1. A *destination*
    - Describes to whom this route is for, i.e. a route to *who*
2. A *link*
    - The `Link` object over which we can reach this host
3. A *gateway*
    - This is who we would need to forward the packet *via* in order to get the packet either to the final destination (in such a case the $gateway = destination$) *or* the next-hop gateway that we must forward via ($gateway \neq destination$)
4. A *distance*
    - This is metric which doesn't affect *how* packets are forwarded but rather how routes that have the same matching *destination* are tie-broken.
    - Given routes $r = \{r_1, r_2\}$ and a function $d(r_i)$ which returns the distance we shall install the route r_i which has the lowest distance, hence $r_{installed} = r_i where d(r_i) = min(d(r))$ (TODO: fix this maths)
5. A *timer* and *lifetime*
    - We have *timer* which ticks upwards and a *lifetime* which allows us to check when the $timer > lifetime$ which signifies that the route has expired, indicating that we should remove it from the routing table.

And in code this can be found as the `Route` struct shown below:

```
/**
 * Represents a route
 */
public struct Route
{
    private string dstKey; // destination
    private Link ll; // link to use
    private string viaKey; // gateway (can be empty)
    private ubyte dst; // distance

    private StopWatch lifetime; // timer
```

```
    private Duration expirationTime; // maximum lifetime

    ...
}
```

2.1.1 Methods

Some important methods that we have:

Method	Description
`isDirect()`	Returns `true` when *gateway* = *destination*, otherwise `false`
`isSelfRoute()`	Returns `true` if the `Link` is `null`, otherwise `false`

2.1.2 Route equality

Lastly, route equality is something that is checked as part of the router's code, so we should probably show how we have overrode the `opEquals(Route)` method. This is the method that is called when two `Route` structs are compared for equality using the `==` operator.

Our implementation goes as follows:

```
public struct Route
{
    ...

    /**
     * Compares two routes with one
     * another
     *
     * Params:
     *   r1 = first route
     *   r2 = second route
     * Returns: `true` if the routes
     * match exactly, otherwise `false`
     */
    public static bool isSameRoute(Route r1, Route r2)
    {

        return r1.destination() == r2.destination() &&
               r1.gateway() == r2.gateway() &&
               r1.distance() == r2.distance() &&
               r1.link() == r2.link();
    }
```

```
    /**
     * Compares this `Route` with
     * another
     *
     * Params:
     *   rhs = the other route
     * Returns: `true` if the routes
     * are identical, `false` otherwise
     */
    public bool opEquals(Route rhs)
    {
        return isSameRoute(this, rhs);
    }
}
```