NIIP lab 5 : opportunistic and mesh networking

In many cases, there is no way to know the network topology on which an application will be deployed. Sometimes, we can't even be sure that there will be a consistent connection or infrastructure available all the time. This is where opportunistic and mesh networking protocols will be useful.

In this lab, we will simulate an emergency condition where the main telecommunications infrastructure has been damaged. There is no mobile access to the WAN (3G/4G) and you will solely have to rely on inter-device communication to pass messages.

You may assume the following:

- The messages you are trying to pass are simple text based entries. We will assume an SMS or Twitter like character limit. The user can just enter this string to be passed when prompted or you can pass it through a command line parameter.
- Each application instance has a unique ID; we will assume these are telephone numbers to make it a little easier and to ensure they are unique. That way, we can also assume that users (in the emergency situation) will have the contact information of the people they are trying to reach already in their phone.
- The application may be developed on a plain x86 Linux system. There is no (initial) need to take into account specifics of mobile platforms or specialized hardware. You need not take into account details on (ad-hoc) wireless networking for the purpose of this lab. You can thus use your laptops to run all of the software.
- The opportunistic or mesh network is an overlay network, meaning that it runs over an already established physical network. Just like we discussed with P2P networks in Network Design and Applications. We will therefore use an application-layer implementation in Python 3 and simulate network specifics on the application layer.

The challenge in designing/implementing these protocols is to validate that they actually work. To do that, you will build your own simulation environment using Containers.

- We use Docker on Linux as a container system. You can manage these using command line parameters or (apparently) also using some Python IDEs.
- Each container runs a single instance of your application. It thus simulates the device of a single user. As long as the container is running, it will be part of the network and able to send and receive messages (on container level). Communication is done through the container's network adapter; you could use e.g. UDP to encapsulate your application layer overlay network messages.
- All containers share a single IP space to facilitate actual L3 routing on the network (routing specifics are normally not exposed to the application using the overlay network). You may decide for yourself how you configure this; it could be done either through IPv4 or v6. The addressing scheme in your overlay network should be based on the telephone number system discussed before.
- Containers should be started/stopped using scripting; you can use e.g. random timers to determine if/when they will be active. There is no state within the container or persistent storage; assume that

they only handle a single task (running your application). This simulates users turning on/off their phone at any time to conserve battery life.
- On the application level, we will simulate that not all clients are able to contact each other at all times. This simulates clients coming in or going out of range of each other, depending on their location (which is variable). To do this, you have to be able to filter the incoming messages (on the application layer) from other containers. For example, you could enable a filter where a system only accepts messages from others with an 'odd' or 'even' telephone number. You can make this behavior change over time to simulate mobility of clients.
- The system needs to be able to handle individual clients leaving/joining at any time. We require a 'proof of delivery' of the text messages. If they have not arrived within a specific time, retransmissions are required.
- You need to simulate large numbers of clients (at least tens of instances); if you keep your application lean and mean, this should not be an issue.

For an intro on opportunistic networking, we recommend checking the following paper : https://www.brambonne.com/docs/barzan13opportunisticrouting.pdf . It also discusses why opportunistic networking makes sense in these conditions. Of course, there is no need to implement all opportunistic protocols discussed in the paper, limit yourself to a simple case (e.g. a flooding-based scheme).

When extending to the case of mesh networking, you have to consider issues like routing for messages to follow the more optimal path(s) between end-nodes. There is a huge amount of existing work on routing algorithms for these networks, however it is not likely that this can be implemented in the given timespan. We therefore recommend you to think about a basic scheme yourself and to validate it in the simulation environment. Given the limited amount of time, do not go overboard in designing a very complex scheme, keep it simple (at first). To give you some pointers if you need ideas – again, you should NOT implement these protocols as they are, this is way too complex – you can have a look at

- AODV (Ad hoc On-Demand Distance Vector Routing)
- ABR (Associativity-based Routing)
- OLSR (Optimized Link-state Routing Protocol)

You have also used a similar algorithm before, but not in the context of mesh networking. OSPF (see CN/NLTT course) also uses some of these principles.

Make sure you add useful debug information to your software to enable us to see what is going on. It will also help you in finding out what bugs there are in the implementation.

We expect a brief report on your findings (google doc), a presentation and a demonstration. You have 2 weeks for this assignment.

General approach

Deel 1: Direct delivery

- Broadcast flood (filter op ontvangen replies voor simulatie)
- Ontvang adressen van available clients
- Unicast message naar client
- Acknowledge unicasted terug
- Indien geen ACK ontvangen binnen een timeout (vb 10 seconden), retransmit message.

Deel 2: Mesh met relay

- Broadcast flood (filter op ontvangen replies voor simulatie)

Bij verzenden van message:

1. Address (nummer) "gekend", maar locatie niet, probeer toch te zenden: Epidemic
    - Zend naar iedereen (buiten source) tot destination reached en ACK geflood wordt
2. Address (nummer) "gekend", maar locatie niet, vraag locatie aan buren: ~First Contact
    - Vraag aan bekende clients welke andere via hun available zijn (recursive)
    - Bij zenden message, maak relay packet dat via een andere client gaat om destination te bereiken.
    - Bij nieuwe client, uitpakken en opnieuw capsuleren om te relayen.
    - Acknowledge op message ook relayed naar oorspronkelijke zender.

Dit waren mogelijke strategieën waar we aan dachten, voor iets te implementeren. Hieronder een kort overzicht van wat we uit de paper van Bram Bonné.

Opportunistic Routing protocols (from paper)

- Direct delivery
  Only directly deliver to destination (no relay)
- First Contact
  Only one copy of a message exists at a certain point in time (message is relayed to the first node unless node already got the message at some point)
- Epidemic
  Replicate messages using a flooding approach (all messages are constantly being sent to all nodes in range)
- Spray and Wait
  A maximum of n copies of a message can exist in a network (copies are sprayed to encountered nodes, until only one copy remains at the sender)
- PRoPHET
  Try to estimate the chance a node can deliver the message to another node, based on historical information and spread messages based on this chance

Implementatie

Basis

Creatie van client

Elke client wordt gemaakt met behulp van een gsm nummer (32 bit integer). Dit nummer kan meegegeven worden met het docker command en er wordt verwacht dat dit een numerieke waarde is (dus geen letters en speciale tekens). Tevens kan er een initiële message worden meegegeven in de vorm: `<address>:<msg>`. Deze message zal vervolgens gestuurd worden naar het meegegeven adres (gsm-nummer, dus niet het IP-adres van de client). Het docker commando om te starten, ziet er als volgt uit:

```
docker run -e MSG="123:Hello World!" -e ADDR="0499123456" -it lab5
```

Met hierbij het adres van de nieuwe container 499123456 en een initiële boodschap "Hello World!" die naar de client met nummer 123 verzonden moet worden. De "-it" flag, zorgt dat de container in een interactieve modus met terminal gestart wordt, zodat we ook nieuwe messages kunnen verzenden.

Later voegden we ook een extra optie toe aan het adres om naar toe te zenden. In het formaat `<address>@<strategy>:<msg>` kunnen we na een @-symbool de zend strategie meegeven, zijnde Direct Delivery (1), Opportunistic Routing (2; zie verder voor details) en Mesh Routing (3; zie verder voor details).

Aanmelden in netwerk

Nadat een client gemaakt is, zal deze zich aanmelden in het netwerk met behulp van een broadcast. De client zal een UDP-pakket sturen naar het broadcast adres van het netwerk (255.255.255.255) zodat elke client die reeds in het netwerk zit dit pakket zal ontvangen. In dit UDP-pakket zit een `DISCOVER`-pakket geëncapsuleerd. Dit pakket bevat een packet-id, een gsm-nummer (source address) en een IP-adres waarop het nummer te bereiken is, in de payload.

Nadat er een discover pakket verstuurd is, zal er een nieuwe thread gecreëerd worden die gaat luisteren naar nieuwe broadcasts pakketten. Binnenkomende broadcast pakketten kunnen van het type `DISCOVER` of `DISCOVER_ACK` zijn.

Wanneer er een `DISCOVER` ontvangen wordt, zal de client een `DISCOVER_ACK` maken. Hierin zit het adres (nummer) van de client met als destination het nummer van degene die het stuurde. In de payload geven we het eigen IP-adres mee, zodat de ontvanger het adres kan toevoegen in zijn adresboek en associëren met het nummer.

Bij het ontvangen van een `DISCOVER_ACK` pakket wordt het adres van de afzender toegevoegd aan het adresboek van de ontvanger. Verder zijn hier geen extra stappen nodig om de handshake tot stand te brengen. Bij ontvangen van eender welk `DISCOVER` pakket, zal het (netwerk) adresboek ook geüpdatet worden en geprint. Enkel naar deze adressen kan iets verzonden worden.

Deze stap wordt altijd uitgevoerd om het netwerk te *discoveren*, ongeacht welke methode van zenden of eigenlijke netwerktopologie er verwacht wordt. MEt extra command line parameters (ook aan de docker run) kan een whitelist worden meegegeven. Enkel inkomende pakketten uit die lijst zullen worden behandeld (zie verder), elke andere gedropt. Er wordt een boodschap in de console geprint als dit gebeurt.

Opzetten van message server en retransmit handler

Nu de client zich aangemeld heeft in het netwerk en de overige clients het adres hebben geleerd, gaat de client een message server en retransmit handler initiëren, beide in een nieuwe thread.

Bij het aanmaken van de message server wordt er een nieuwe UDP server socket aangemaakt die vervolgens gebruikt wordt in de server thread. Deze server thread handelt de binnenkomende pakketten af (zie verder).

De retransmit handler thread zal constant door een lijst lopen waarin de pakketten vermeld staan die nog niet acknowledged zijn. Wanneer er te lang (`RETRANSMISSION_TIMEOUT`) geen acknowledgement ontvangen wordt, zal de client het pakket opnieuw verzenden. Voor het Opportunistic routing (zie verder) zal de geschiedenis van geziene pakketten na een Time To Live ook verwijderd worden. In het geval van Mesh Routing zal het hele algoritme van RouteRequest tevens herhaald worden (om zo te vermeiden dat onbruikbare routes hergebruikt zullen worden).

Server thread

Bij het ontvangen van een `MESSAGE` pakket (Direct Delivery), indien het destination address het onze is, printen we de payload (de boodschap zelf). Daarna stellen we een `MESSAGE_ACK` pakket samen met hetzelfde ID en zenden we deze terug. Indien het niet voor deze client bestemd was, wordt het pakket genegeerd.

Bij ontvangen van een `MESSAGE_ACK` (met de huidige client als bestemming), kunnen we het ID terug vrijgeven en moeten we de message die dit pakket ACKte niet meer opnieuw verzenden.

Verzenden van messages (Direct Delivery)

Bovenstaande threads behandelen inkomende broadcasts en messages (en hun ACKs). Wanneer er bij initiatie een message werd meegegeven wordt deze eerst verzonden naar het desbetreffende gsm-nummer (in `<address>:<msg>` formaat). Vervolgens zal er via de terminal een mogelijkheid zijn om bijkomende messages te verzenden. De client zal een nummer vragen als bestemming en vervolgens de message zelf. Indien het nummer gekend is, kan de message verzonden worden. Er is een character limiet ingesteld van 255 voor elke message. Bij langere, krijg je een invalid message boodschap.

Messages worden geëncapsuleerd in een `MESSAGE` pakket, dit pakket bevat een ID, gsm-nummer van de afzender (client zelf), gsm-nummer van de ontvanger en de desbetreffende data. Nadat we het pakket verzenden via UDP, slaan we het pakket op in de lijst met verwachte acknowledgements en zetten de tijd van verzending. Op deze manier kan de client achteraf nakijken of er op tijd een acknowledgement ontvangen werd voor het verzonden bericht en eventueel retransmitten.

Een retransmit kan bijvoorbeeld nodig zijn als een andere client op het netwerk kwam, zijn adres broadcastte, maar dan terug weg ging. De huidige client kent nog steeds het nummer, maar het IP is nu niet meer bereikbaar. Als er later een client met hetzelfde nummer joint, wordt door diens broadcast het IP geüpdatet en zal de eerstvolgende retransmit lukken.

Voorbeeld output

```
PS C:\Users\William\OneDrive\UH\Master\NIIP\Labo's\Lab5> docker run -e ADDR="0499123456" -it lab5
Waiting for keypress... (set-up tcp dump now)

[Client@499123456] Setting up client at 172.17.0.2...
[Client@499123456] Joining network by broadcasting address info...
[Client@499123456] Broadcasting <DISCOVER id=1, len=4, src=499123456, dst=0, data=b'\xac\x11\x00\x02'>
[Client@499123456] Incoming broadcast from 172.17.0.3:10100...
[Client@499123456] Received: <DISCOVER_ACK id=2, len=4, src=487654123, dst=499123456, data=b'\xac\x11\x00\x03'>
[Client@499123456] Address book: 499123456, 487654123
Enter an address     > 487654123
Enter a new message > Hello

[Client@499123456] Sending to 487654123: b'Hello'
[Client@499123456] Sending: <MESSAGE id=2, len=5, src=499123456, dst=487654123, data=b'Hello'>
Enter an address     >
[Client@499123456] Incoming connection with 172.17.0.3:5000
[Client@499123456] Received: <MESSAGE_ACK id=2, src=487654123, dst=499123456>
[Client@499123456] Message was acknowledged!
```

(Omdat andere threads kunnen printen terwijl user input wordt gevraagd, staat de prompt mogelijk nog op een vorige lijn dan verwacht, zoals hierboven de 2de "Enter an address". Men kan gewoon typen en ENTER drukken, de andere print statements hebben geen invloed op de interactie.)

Docker container

Onze docker container file wordt gemaakt met volgende instellingen:

```
#publicly available docker image "python" on docker hub will be pulled
FROM python:3

# Set label
LABEL build_version="NIIP_LAB5_1.0"
LABEL maintainer="Bram_William"

# Expose port
EXPOSE 5000

ENV ADDR=0 \
    MSG="None" \
    WHITELIST="None" \
    FILTER=1

COPY /Client/ /source

CMD python3 -u /source/client.py -a ${ADDR} -m "${MSG}" -w ${WHITELIST} -f ${FILTER}
```

We voorzien ook een tweede container met tcpdump om traffic te capturen:

```
FROM ubuntu
RUN apt-get update && apt-get install -y tcpdump
CMD tcpdump udp -vv -i eth0
```

De workflow is dan als volgt:

1. Eerst moet het script in de root folder uitgevoerd worden: `.\BuildDocker.ps1`
    a. Wij gebruiken momenteel Docker op Windows via Powershell, maar de commando's kunnen natuurlijk ook gewoon zo gebruikt worden op een unix systeem.
    b. Dit script stopt eerst alle lopende (docker) processen, pruned daarna de bestaande images en build de bovenstaande basis DockerFile.
2. Optioneel kan daarna het script in Dumper uitgevoerd worden om de tcpdump container te maken:
    a. `cd Dumper && build.bat` of apart:
    ```
    cd Dumper
    docker build -t tcpdump .
    docker network create demo-net
    ```
    b. Nu kan elke container het demo-net gebruiken.
3. Run een aantal clients:
    a. `docker run -e ADDR="0499123456" --network demo-net --name client1 -it lab5`
    b. `docker run -e ADDR="123" --network demo-net -it lab5`
4. Elke client zal eerst wachten op een ENTER voordat hij start. Hierdoor kan eerst de tcpdump container gestart worden:
    a. `docker run -it --net=container:client1 tcpdump`
    b. Deze hecht zich nu aan de interface van client1 binnen demo-net
5. Nu kunnen we de clients verder zetten door op ENTER te drukken.
    a. Messages kunnen verzonden worden zoals eerder beschreven.
6. In de tcpdump terminal kunnen we het netwerkverkeer vervolgens analyseren.

Voorbeeld output

```
PS Lab5\Dumper> docker run -it --net=container:client1 tcpdump tcpdump udp -vv
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
15:39:19.187644 IP (tos 0x0, ttl 64, id 32650, offset 0, flags [DF], proto UDP (17), length 43)
    817ed025d77a.10100 > 255.255.255.255.10100: [bad udp cksum 0xac3f -> 0x2aee!] UDP, length 15
15:39:19.188407 IP (tos 0x0, ttl 64, id 30358, offset 0, flags [DF], proto UDP (17), length 74)
    817ed025d77a.35627 > 192.168.65.1.53: [bad udp cksum 0xae08 -> 0x8981!] 18165+ PTR?
255.255.255.255.in-addr.arpa. (46)
15:39:20.037065 IP (tos 0x0, ttl 37, id 4388, offset 0, flags [none], proto UDP (17), length 74)
    192.168.65.1.53 > 817ed025d77a.35627: [udp sum ok] 18165 NXDomain q: PTR?
255.255.255.255.in-addr.arpa. 0/0/0 (46)
15:39:20.038102 IP (tos 0x0, ttl 64, id 31160, offset 0, flags [DF], proto UDP (17), length 71)
    817ed025d77a.40725 > 192.168.65.1.53: [bad udp cksum 0xae05 -> 0x26a8!] 14819+ PTR?
1.65.168.192.in-addr.arpa. (43)
15:39:20.067176 IP (tos 0x0, ttl 37, id 24941, offset 0, flags [none], proto UDP (17), length
71)
    192.168.65.1.53 > 817ed025d77a.40725: [udp sum ok] 14819 NXDomain q: PTR?
1.65.168.192.in-addr.arpa. 0/0/0 (43)
15:40:19.735031 IP (tos 0x0, ttl 64, id 21198, offset 0, flags [DF], proto UDP (17), length 43)
    6cf7c03e9bbc.demo-net.10100 > 255.255.255.255.10100: [bad udp cksum 0xac40 -> 0x6f0f!] UDP,
length 15
15:40:19.735272 IP (tos 0x0, ttl 64, id 19297, offset 0, flags [DF], proto UDP (17), length 43)
    817ed025d77a.10100 > 6cf7c03e9bbc.demo-net.10100: [bad udp cksum 0x5858 -> 0x01d5!] UDP,
length 15
15:40:37.625329 IP (tos 0x0, ttl 64, id 33874, offset 0, flags [DF], proto UDP (17), length 44)
    817ed025d77a.48477 > 6cf7c03e9bbc.demo-net.5000: [bad udp cksum 0x5859 -> 0xc45c!] UDP,
length 16
15:40:37.625548 IP (tos 0x0, ttl 64, id 55892, offset 0, flags [DF], proto UDP (17), length 39)
    6cf7c03e9bbc.demo-net.5000 > 817ed025d77a.5000: [bad udp cksum 0x5854 -> 0x3f65!] UDP,
length 11
```

Legende:

Traffic entry start, Extra, irrelevante info van IP-pakket

Client 1 id, Client 2 id, UDP DNS service van docker?

Analyse:

1. 817ed025d77a.10100 > 255.255.255.255.10100 (length 15)
    a. Dit is de initiële broadcast (DISCOVER) van client 1, vertrekkende van zijn de standaard broadcast poort 10100 (in onze implementatie), naar het broadcast IP van het netwerk.
    b. Hier is er geen antwoord, omdat client 1 de enige client is op het netwerk.
    c. Merk op dat de lengte 15 bytes is: 1 byte voor type + 1 voor lengte + 1 voor id + 4 voor bron adres (nummer 499123456) + 4 voor destination (nummer 0) + 4 voor het IP van client 1 in de payload == 15
2. Exchange met 192.168.65.1
    a. Dit is vermoedelijk de DNS service binnen het virtuele docker netwerk. Het IP adres van client 1 is (volgens socket.gethostname()) namelijk 172.21.0.2, dus buiten dit 192.168.x.x netwerk.
3. 6cf7c03e9bbc.demo-net.10100 > 255.255.255.255.10100
    a. Client 2 komt online in demo-net en stuurt een broadcast.
    b. Opnieuw zien we de correcte lengte van 15 bytes.
4. 817ed025d77a.10100 > 6cf7c03e9bbc.demo-net.10100
    a. Client 1 kreeg de broadcast binnen en stuurt een DISCOVER_ACK terug (naar broadcast poort).
    b. De lengte is hier opnieuw 15 bytes, omdat de payload nu het IP van client 1 bevat.

5. `817ed025d77a.48477 > 6cf7c03e9bbc.demo-net.5000 (length 16)`
    a. Client 1 stuurt de boodschap "Hello" naar client 2
    b. Aangezien de payload nu uit 5 bytes bestaat, zien we dat de lengte inderdaad 1 byte meer is dan bij DISCOVER related pakketten.
6. `6cf7c03e9bbc.demo-net.5000 > 817ed025d77a.5000 (length 11)`
    a. Client 2 antwoord met een `MESSAGE_ACK`.
    b. Dit pakket bestaat uit type, lengte, id, source en destination address, zonder payload, en dus 11 bytes.

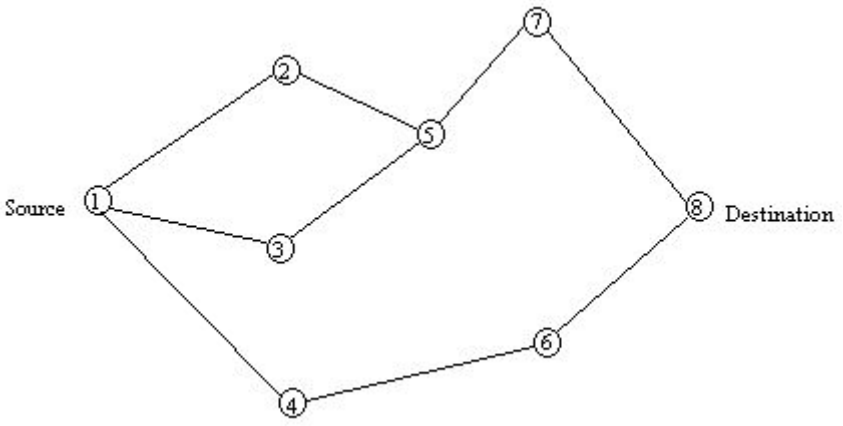
Netwerk simulatie

1. Netwerk condities

- Filteren van verbindingen, vb even nummers kunnen enkel met oneven communiceren (inkomende pakketten van zelfde soort worden gedropt, e.g. broadcast)
  - ➢ Zodat minstens 1 hop nodig om zelfde soort nummers te bereiken
  - ➢ Met `FILTER` argument aan docker run (filter=1: allow all, filter=2: only opposite evenness)
  - ➢ Vb `docker run … -e FILTER=1 ...`
- Whitelisten van adressen
  - ➢ Met WHITELIST argument aan docker run
  - ➢ Bevat lijst zodat elke client inkomende `DISCOVER`s en `DISCOVER_ACK`s kan filteren en enkel de adressen in de lijst overhouden. Dit kan gebruikt worden om enkel bepaalde clients met enkele andere te verbinden en zo een bepaalde topologie op te stellen.
  - ➢ Vb `docker run … -e WHITELIST="[1,5]" ...`
- Na random tijd een client disconnecten en mogelijk reconnecten (met hopelijk ander IP)
  - ➢ Niet geïmplementeerd, maar te simuleren door client zelf te stoppen en terug te starten.
- Na ontvangen van een pakket, een random tijd wachten voordat we het behandelen of opnieuw relayen.
  - ➢ Niet geïmplementeerd.

2. Dynamisch aanmaken van clients

- Python script dat clients spawnt, een message verzend en daarna eventueel terug disconnect.
  - ➢ Hierbij worden dan ook het `WHITELIST` argument gezet, zodat de netwerktopologie in acht wordt genomen.
  - ➢ We voorzien het script `.\Docker\start_network.bat` dat 8 clients aanmaakt volgens de topologie hieronder. Elke nieuwe terminal is genoemd naar de client die het voorstelt. En na creatie, is er een 4 seconden delay voorzien, zodat men de vensters kan verslepen naar de locaties zoals op onderstaande afbeelding. Zo is het makkelijker om te visualiseren welke client met welke verbonden is tijdens uitvoeren. (dit script maakt enkel het netwerk, maar zendt geen initiële boodschap)



Opportunistic routing

Merk op dat bij het activeren van een nieuwe client altijd een `DISCOVER` *pakket gebroadcast zal worden om alle clients in het netwerk te zoeken. Dit is een essentiële voorbereidingsstap om de onderliggende IP-adressen te kunnen vinden van bereikbare clients. Door gebruik te maken van de* `WHITELIST` *lijst of het filter argument, kan men clients enkel met bepaalde andere laten verbinden om een topologie te simuleren.*

1. First contact

Werking

- Zend 1 pakket naar het eerste gekende adres of rechtstreeks naar destination als dat een gekend contact was.
- Elke client zendt het pakket gewoon naar zijn eerstvolgende contact, maar houdt wel bij naar welke contacten hij het bericht met (source en id) al verzonden heeft.
- Als een client een pakket kreeg dat hij al eens gezien heeft, zendt hij het terug van waar het kwam, indien hij het nog niet naar die client verzond. Als dit wel zo is, moet hij het sturen naar een volgende contact.
- Dit blijft gebeuren tot een client geen contacten heeft om het pakket naar toe te sturen, en de destination dus niet bereikbaar bleek. Het pakket wordt dan gedropt en na de ACK timeout op de source, zal het opnieuw verzonden worden. Hierbij wordt het op de source dan naar het volgende contact verzonden. Indien alle contacten reeds gekozen werden, wordt de history voor dat pakket gereset en proberen we het opnieuw vanaf het eerste contact.
- Indien het pakket de destination bereikt, zal de ACK met dezelfde techniek terug worden gestuurd.

Dit principe hebben we geïmplementeerd als modus 2 bij het zenden van een pakket.

Mesh routing

Merk op dat bij het activeren van een nieuwe client altijd een `DISCOVER` *pakket gebroadcast zal worden om alle clients in het netwerk te zoeken. Dit is een essentiële voorbereidingsstap om de onderliggende IP-adressen te kunnen vinden van bereikbare clients. Door gebruik te maken van de* `WHITELIST` *lijst of het filter argument, kan men clients enkel met bepaalde andere laten verbinden om een topologie te simuleren.*

1. Relay routing

Werking

- Zend een Relay pakket waar de previous en next hop (samen met source en destination address) in staan aangegeven, naar elke gekende client.
- Een client relayed dit pakket dan naar een volgende gekende client, en natuurlijk niet naar de vorige. Indien de source het pakket opnieuw ontvangt, wordt het gedropt.
- Elke client stelt een eigen lijst op van source-destination hops vanuit inkomende Relay pakketten.
  Vb (zie netwerktopologie afbeelding) client 5 krijgt een Relay binnen van 1 (source) die van 3 (previous hop) komt en naar 8 (destination) moet gaan. Hij kan dus onthouden dat client 1 bereikbaar is via client 3 (met cost van hop count). Als hij later een pakket van client 7 krijgt met als destination 1, kan hij er van uitgaan dat die te bereiken is via 3.
- Indien een route gekend is naar de destination, kan de volgende relay naar daar verzonden worden, indien niet, wordt het gerelayed naar de gekende clients zoals voorheen, tot het aankomt op de destination.

- De destination kan dan hetzelfde doen, maar aangezien elke client nu een next hop kent, kan er een meer directe route genomen worden.
- Dit is ongeveer hetzelfde principe als First Contact hierboven, maar nu houdt elke client effectief rekening met de routes en hops in zijn geschiedenis van geziene pakketten. Waar een client bij FirstContact enkel per pakket (id, src, dest) kijkt, maken we nu een globaler overzicht van mogelijke routes op elke client.

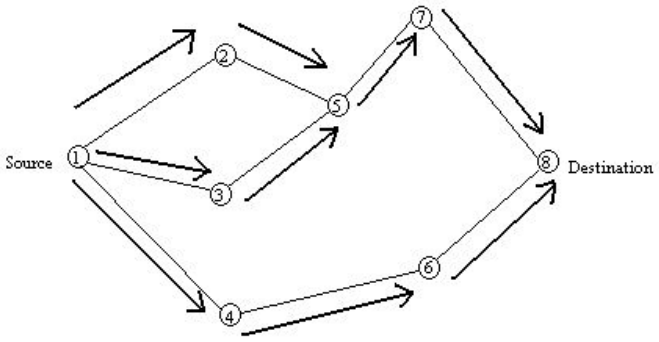
2. Address requests (gelijkaardig aan de traceroute utility)

Werking

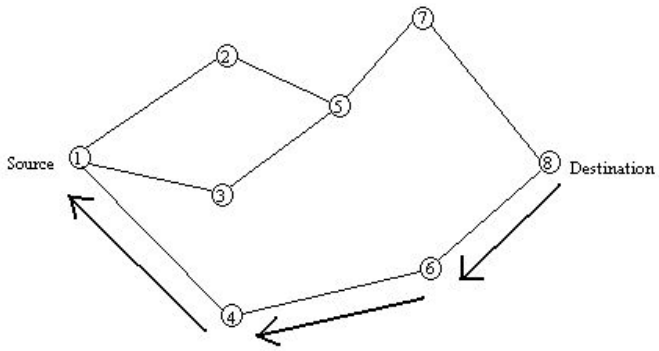
- Dit werkt ongeveer hetzelfde als het Relay routing, maar nu niet uit het standpunt van aparte clients, maar uit die van de source.
- De source zendt een `AddressRequest` naar zijn gekende adressen en update telkens een graaf met de adressen die zijn buren kennen. Daarna stuurt die een `AddressRequest` naar elke nieuwe clients die een andere client kende, tot hij een client tegenkomt die het adres van de gevraagde destination kent. Vanaf dan kan een Relay pakket gestuurd worden die de gevonden route volgt naar de destination.
  Vb (zie netwerktopologie afbeelding), client 1 zendt een `AddressRequest` naar (2, 3, 4) en krijgt terug: (2 -> 5, 3 -> 5, 4 -> 6). Daarna zendt hij een request naar (5, 6), relayed via hun respectievelijke hops. Als resultaat krijgen we dan (5 -> 7, 6-> 8). Aangezien we naar 8 willen zenden, kunnen we nu dus een Relay route opstellen met een serie hops: (4, 6, 8) om de destination te bereiken.

3. Route requests

Werking



(a) Propogation of Route Request (RREQ) Packet



(b) Path taken by the Route Reply (RREP) Packet

- Zend een `RouteRequest` pakket: dit wordt verstuurt naar elke verbonden client met het doel een route op te stellen naar de destination. Hiervoor voegt een client zichzelf toe als next hop in het pakket en forward het nieuwe pakket naar zijn buren.

- Bij ontvangen van een `RouteRequest` dat verzonden werd door de client zelf, wordt deze gedropt.
- Als de destination client één of meer `RouteRequests` binnen krijgt, worden deze allemaal geACKt. Uiteindelijk besluit de oorspronkelijke client welke van de verkregen routes (uit RouteRequest ACK) dat hij gebruikt.
    - ➢ Het `RouteRequestAck` pakket bevat nu de omgekeerde route, en elke client die deze moet relayen, zal in die route de next hop zoeken om het pakket te forwarden.
    - ➢ De source moet wachten tot er minstens 1 ACK binnenkomt, indien dit niet gebeurt binnen een bepaalde tijd, zal het `RouteRequest` pakket opnieuw verstuurt moeten worden.
    - ➢ Bij ontvangen van 1 ACK, zou hij nog even kunnen wachten op een volgende, maar hij kan die al gebruiken, aangezien het waarschijnlijk de snelste route zal zijn.
- Nu een route gekend is, kan de eigenlijke message (`RouteRelay`) verzonden worden in een Relay pakket (met de omgekeerde route van het ACK pakket), waar telkens de next hop in gebruikt wordt om de route te volgen.
- Bij ontvangen van de message, zal de destination ofwel de route van het pakket kunnen gebruiken in reverse, of zal die een nieuw `RouteRequest` moeten sturen.

Opmerkingen

- Indien het netwerk en de verbonden clients niet erg stabiel zijn, is het mogelijk dat er veel verkeer gegenereerd wordt, wat niet altijd wenselijk is.
- Pas als de `RouteRelayAck` ontvangen wordt, kunnen we de message beschouwen als verzonden. Indien dit niet gebeurt, zal er als retransmit een nieuwe `RouteRequest` moeten verzonden worden om de route te herontdekken.

Opportunistic implementatie: First Contact (Contact Relay)

Voor het opportunistic routing namen we dus het First Contact principe. Bruikbaar met strategie 2 als argument: `<address>@2`.

ContactRelay

Omdat we geen idee hebben van hoe het netwerk er precies uitziet, enkel van de buren die we kennen, sturen we ons pakket met message gewoon door naar het eerste contact dat we kennen. We houden ook bij dat we een message doorsturen met id, source en destination via dat contact. Als één van onze contacten de beoogde destination is, kunnen we het pakket gewoon rechtstreeks naar die verzenden.

Bij het ontvangen van een `ContactRelay` pakket dat niet voor ons bestemd is, zenden we het pakket ook naar het eerstvolgende contact, en houden we dit ook bij in een geschiedenis. Indien een client hetzelfde pakket nog eens krijgt, vraagt hij volgens zijn geschiedenis het volgende contact op waar het pakket naar verzonden kan worden. Als alle contacten gebruikt werden, zenden we het pakket terug naar de originele client die het ons stuurde, zo kunnen we het netwerk depth-first trachten af te gaan. Als het pakket terug binnenkomt op de source, maar we verzonden het al naar alle buren, resetten we de geschiedenis en proberen het opnieuw.

Om te voorkomen dat we oneindig proberen naar, misschien al weggevallen buren, te verzenden, zal de retransmit handler thread ook om de 60 seconden nieuwe `Discover` pakketten genereren om naar elke contact te verzenden. Zo kunnen we onze adreslijst up to date houden. Inkomende broadcasts worden nog steeds toegevoegd aan de lijst.

ContactRelayAck

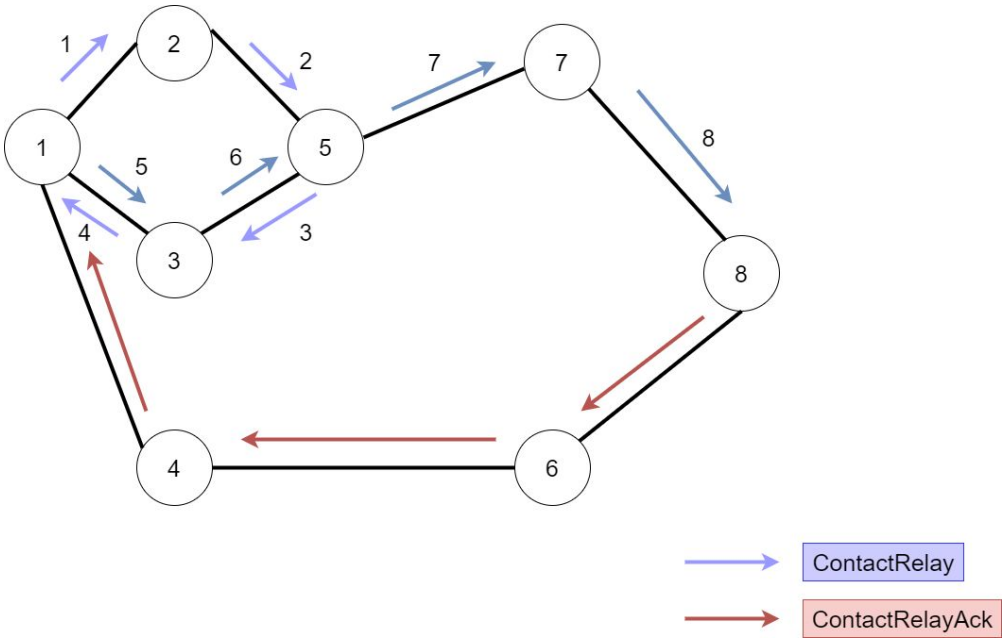
Als een client de destination is van een inkomend `ContactRelay` pakket, heeft hij dus de message ontvangen. Een `ContactRelayAck` pakket wordt nu opgesteld en op dezelfde manier doorgestuurd als het `ContactRelay` pakket. Indien een client dit pakket ziet, zal hij het ook toevoegen aan zijn geschiedenis en daarna forwarden. Als de geschiedenis een entry bevat van het oorspronkelijke `ContactRelay` pakket, mag dit nu ook verwijderd worden.

Elke entry in deze geschiedenis heeft een Time to Live (TTL) waarna de entry verwijderd zal worden. Daarbovenop tellen we de hop count binnen het pakket ook op bij elke forward. Als deze groter wordt dan 255, wordt het pakket gedropt.

Opportunisctic flow test

Op onderstaande afbeelding zenden we een pakket van client 1 naar client 8. Er bestaat telkens slechts 1 instantie van het pakket in het netwerk dat wordt doorgegeven naar een volgende client. Client 1 stuurt het pakket naar zijn eerste contact, namelijk client 2. We zien een loop ontstaan met volgende hops: `1->2->5->3->1`. Elke client zendt namelijk naar zijn eerste contact, zoals gezegd. Als het pakket terug op client 1 aankomt, checkt hij zijn pakketgeschiedenis: hij verzond het al naar 2, maar nog niet naar 3 en 4. Aangezien hij het pakket enkel naar een nieuw contact zal zenden als hij het terug krijgt van zijn eerste verzending (van client 2 dus), zendt hij het terug via client 3. Deze heeft het nog niet naar 5 verzonden, dus forward het naar daar. Client 5 kreeg het van 2 en verzond het al naar 3, dus blijft nu enkel 7 nog over. Uiteindelijk wordt client 8, de destination, nu bereikt.

Die stelt dan een `ContactRelayAck` op en zendt het pakket op zijn beurt naar het eerste contact, client 6. Hier kan client 1 wel bereikt worden met een minimaal aantal hops via `8->6->4->1`.



Mesh implementatie: Route requests

Omtrent het mesh netwerk, hebben wij ervoor gekozen om het principe van `RouteRequests` in te bouwen. Om dit te realiseren hebben we 4 nieuwe pakketten geïntroduceerd die het mogelijk maken een route te bepalen en te gebruiken om een message mee te versturen. Om deze strategie te gebruiken, kan je het volgende formaat gebruiken bij het invoeren van een destination adres: `<address>@3`.

RouteRequest

Wanneer we een pakket trachten te versturen op ons mesh netwerk, zal het algoritme beginnen met het opstellen van een `RouteRequest` pakket. Zoals in onze andere pakketten wordt hier een source en destination address aan toegekend, met bijgevoegd een hop-list. Initieel bevat deze hop-list enkel het adres van de zendende partij. De zender stuurt dit pakket naar alle gekende adressen in zijn adresboek.

Bij het ontvangen van zo een pakket, zal en client eerst een check uitvoeren of hij de gewenste destination is. Mits dit niet het geval is, zal deze zichzelf toevoegen aan de hop-list en vervolgens het pakket relayen naar de nodes in zijn adresboek. Alvorens het pakket te sturen, controleren we eerst of het adres niet reeds voorkomt in de hop-list. Wanneer een gekend adres reeds voorkomt in deze lijst, zal het pakket niet opnieuw verstuurd worden naar die partij.

Wanneer de ontvanger de gewenste destination is, zal deze een `RouteRequestAck` vormen (zie verder).

RouteRequestAck

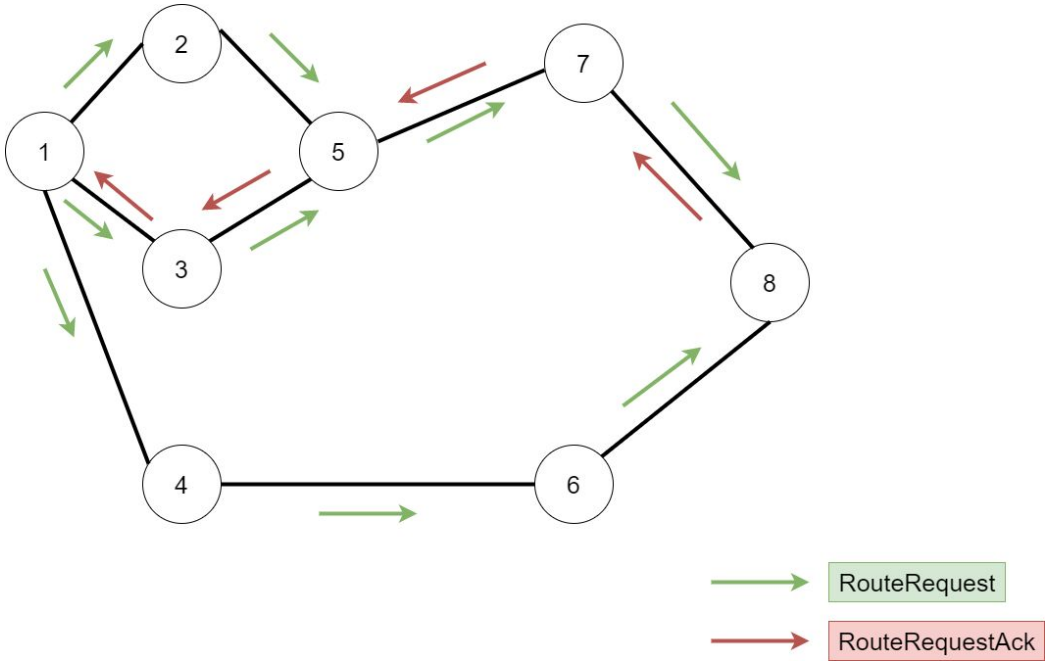
Zoals hierboven beschreven werd, zal de destination een acknowledgement maken wanneer deze een `RouteRequest` binnenkrijgt die voor hem bestemd was. Dit pakket bevat het ID van het ontvangen pakket, waarbij het source adres ingevuld wordt met de destination van het `RouteRequest` pakket en vice versa. Zoals bij het `RouteRequest` pakket, verwacht dit pakket tevens een hoplist. Deze bestaat uit de omgekeerde lijst die uit het `RouteRequest` pakket ontstaan is, met als eerste hop de destination van het `RouteRequest` pakket. Vervolgens zal dit pakket over diezelfde route teruggestuurd worden naar de originele source.

Wanneer een node een `RouteRequestAck` pakket binnen krijgt die niet voor hem bestemd is (dus dat de node in kwestie een relay-node is), zal deze het eerstvolgende adres in de hoplist opvragen. Vervolgens zal dit pakket gerelayed worden naar deze node.

Wanneer tenslotte de `RouteRequestAck` aankomt bij de node die initieel de `RouteRequest` heeft aangevraagd, zal deze de eigenlijke message versturen over deze route met behulp van een `RouteRelay` pakket.

RouteRequest: flow

In onderstaande afbeelding is een schematische weergave voorzien van het werkingsprincipe van een RouteRequest:



Vanuit client 1, wordt eerst het RouteRequest pakket verstuurd naar alle contacten. Dit propageert zich dan over heel het netwerk tot het de destination client 8 bereikt. Die zal op zijn beurt een RouteRequestAck terugsturen, maar nu naar de omgekeerde route uit het Request pakket.

RouteRelay

De source node weet nu welke route er gevolgd moet worden om de message bij de destination te krijgen. Deze zal dan een `RouteRelay` pakket aanmaken, waarin de hoplist en data verwerkt zitten. Nadat de source node dit pakket gemaakt heeft zal deze verstuurd worden naar de volgende hop in de lijst.

Wanneer een node dit pakket ontvangt, maar niet de destination is, zal deze het pakket gewoon verder sturen naar de volgende hop in de lijst. Als de destination-node dit pakket ontvangt, en dus de message ontvangen heeft, zal deze een `RouteRelayAck` pakket maken.

RouteRelayAck

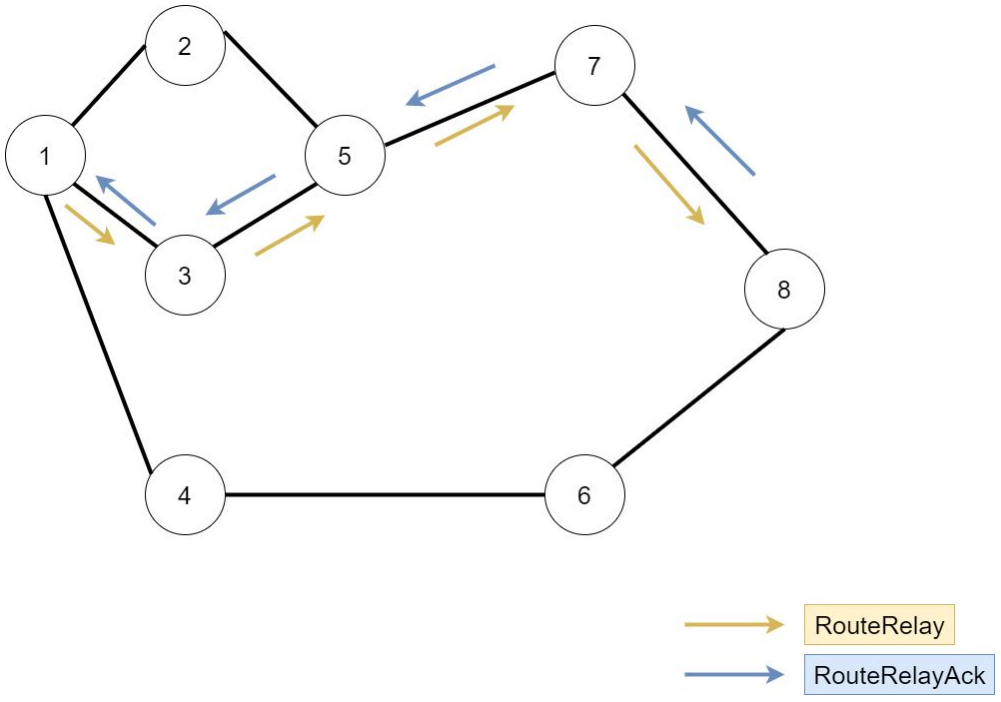
Wanneer we hier geraakt zijn, heeft een node een message ontvangen met behulp van een `RouteRelay`. Om hier een acknowledge op te sturen, maakt de node een `RouteRelayAck` die aan de source node zal melden dat de message succesvol is ontvangen. Dit pakket bevat de hoplist die gebruikt werd in het originele `RouteRelay` pakket, en zal deze hoplist gebruiken om de acknowledgement op terug te sturen.

Zoals bij de voorgaande pakketten, zal een tussennode dit pakket gewoon relayen naar de volgende node in de lijst. Wanneer de originele source-node dit pakket ontvangt, zal deze het originele message pakket verwijderen uit de lijst van expected acknowledgements en is de transmissie geslaagd.

Indien er niet tijdig een `RouteRequestAck` of `RouteRelayAck` ontvangen wordt op de source, gaat deze er van uit dat het pakket verloren ging en zal het een nieuwe `RouteRequest` opstellen en versturen. Door de relatief grote hoeveelheid van pakketten die deze methode teweeg brengt, is het zeker mogelijk dat een tussenliggende client op een gevonden route in tussentijd niet meer bereikbaar is.

RouteRelay: flow

In onderstaande afbeelding is een schematische weergave voorzien van het werkingsprincipe van een `RouteRelay`:



Na het ontvangen van een RouteRequestAck op client 1, wordt nu de eigenlijke message in een RouteRelay verstuurd naar client 8, volgens de omgekeerde route in dat pakket. Client 8 zal bij ontvangst een RouteRelayAck terugsturen.

Mesh Flow test

Op onderstaande afbeelding (volledige resolutie te vinden in het archief bij indienen), hebben we het netwerk aangemaakt zoals ook te zien was op de afbeelding van de netwerktopologie die we gebruikten.

In bovenstaande afbeelding zien we ons mesh netwerk in de praktijk. Hier staan de verbindingen tussen de verschillende clients aangeduid met een rode lijn. Enkel de clients die met zo een lijn verbonden zijn kunnen met elkaar communiceren omdat we het `WHITELIST` argument gebruikten om inkomende `DISCOVER` pakketten te filteren.

Om dit te illustreren sturen we een pakket van Client 1 (uiterst links) naar Client 8 (uiterst rechts). In de gele omkaderingen zien we dat de RouteRequest pakketten worden ontvangen en doorgestuurd naar de naburige clients. Wanneer deze request aankomt op de bestemming, zal deze antwoorden met een acknowledgement, welke zijn aangeduid met een paarse omkadering (RouteRequestAck). De gekozen route in dit geval om van 1 naar 8 te gaan is `1->3->5->7->8`. Deze route zal (omgekeerd wel te verstaan) gebruikt worden om de `RouteRequestAck` terug te sturen naar de source client (Client 1).

We zien bij Client 1 dat deze vervolgens een nieuw pakket aanmaakt: RouteRelay, met hier de uiteindelijke message in. Deze wordt naar Client 8 gestuurd met behulp van de gevonden route. Dit is te zien in de afbeelding in de lichtblauw-omkaderde gedeelten. We kunnen hier heel mooi waarnemen dat het pakket via de route `1->3->5->7->8` gaat en uiteindelijk aankomt bij de bestemming.

Tenslotte zal de destination antwoorden met een RouteRelayAck, welke omkaderd zijn met een oranje kleur. Ook hier wordt de origineel gevonden route hergebruikt om terug bij de source client te geraken. Moest er in de tussentijd een client weggevallen zijn, of iets anders misgelopen zijn zal de source client nooit een acknowledgement ontvangen. Wanneer dit niet gebeurt zal het hele proces herhaald worden, waardoor er dus een nieuwe route gezocht zal worden om de destination client te kunnen bereiken.