ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Network Security Group
April 2025
Version 1.1

SCION JPAN Exercise

1 Introduction

In this exercise, you will use the JPAN[1] library to connect the global (or a local) SCION network. JPAN is full-stack endhost implementation of SCION in Java. It does not require any other SCION libraries to be installed. This exercise requires JPAN 0.5.1 or later.

There are several ways to connect to SCION

- Ideally, you have SCION connectivity. You can check here whether your ISP has connectivity: `https://scion-architecture.net/apps/`.
- You can also set up a local SCION network on your machine. Instructions are available here: `https://docs.scion.org/en/latest/dev/run.html`. This should work as is on Linux and MaxOS, on Windows it is recommended to use WSL.
- Another solution is to set up a private freestanding SCION network on multiple machines: `https://docs.scion.org/en/latest/tutorials/deploy.html`
- It is also possible to connect to SCIONlab, a global overlay network: `https://www.scionlab.org/`. Local SCION access is not required. SCIONlab allows you to create your own ASes.

Hints

- For debugging, it can be helpful to use Wireshark with the Scion plug-in, see `https://docs.scion.org/en/latest/dev/wireshark.html#wireshark`.
- With admin access to the control servers, or when running a local topology, additional path metadata can be added with `staticInfoConfig.json` files, see `https://docs.scion.org/en/latest/manuals/control.html#control-conf-path-metadata`

2 Task Preparation & Hints

2.1 Example Project

JPAN usage examples can be found here:

`https://github.com/netsec-ethz/scion-java-packet-example`

If you have a native SCION connection to the public SCION network, you can run a simple example as follows.

[1] `https://github.com/scionproto-contrib/jpan`

```
$ wget https://github.com/netsec-ethz/scion-java-packet-example/releases/
   download/v0.2.0/scion-packet-example-0.2.0-executable.jar
$ java -jar scion-packet-example-0.2.0-executable.jar "A␣personal␣message"
```

The example will send a packet to the SCION packet inspector: `https://scionpacketinspector.netsec.ethz.ch`

2.2 Configuring JPAN

In a normal setup with a local AS, JPAN should do everything automatically.

If you have multiple daemons running locally, for example, because you are running a topology locally on your machine, you need to specify which daemon you want to use. This can be done via an environment variable or directly in Java as follows:

```
// Conenct to the daemon of the AS "1-ff00:0:131" in the scionproto "default" topology.
System.setProperty(Constants.PROPERTY_DAEMON, "127.0.0.77:30255");
// Now we can use the ScionService and/or open a channel/socket
```

2.3 Establishing a connection to a remote host

Scenario 1: With remote IP + ISD/AS

If you know the IP/port and ISD/AS, the easiest way is to get paths directly from the `ScionService`: `Scion.defaultService().getPaths(ISD/AS, IP)` and then use that with `connect(path)` or `send(packet, path)`.

Scenario 2: URL/host name + DNS entry

If you do not know the IP or ISD/AS, JPAN may be able to look it up via DNS. However, this requires that the host name has a DNS TXT entry that maps to the IP + ISD/AS. Note that this IP may be different from the normal IP returned by a DNS lookup.

To check whether a SCION DNS TXT entry exists, check as follows (example for `ethz.ch`):

```
$ dig TXT ethz.ch | grep scion=
ethz.ch.  130 IN TXT "scion=64-2:0:9,129.132.230.98"
```

If such an entry exists, you can simply create an `InetSocketAddress` with your host name and use it with `connect(address)` or `send(packet, address)`. JPAN will then internally query DNS and request and use paths to the destination.

For example, to connect to ETH Zurich (AS `64-2:0:9`):

```
InetSocketAddress dummy = new InetSocketAddress(InetAddress.getLocalHost(), 12345);
long isdAs = ScionUtil.parseIA("64-2:0:9"); // ETH Zurich
List<Path> paths = Scion.defaultService().getPaths(isdAs, dummy);
Path destination = paths.get(0); // Just use the first.
try (ScionDatagramChannel socket = ScionDatagramChannel.open()) {
```

```
    socket.connect(destination);
}
```

3 Task Description

Your task is to create a simple client/server application that sends a text string back and forth. From all available paths, it should use the one with the lowest latency. The task is split into several subtasks that are partially independent.

The aim of the exercise is to learn:

- Connecting through a SCION network using an ISD-AS identifier.
- Sending and receiving packets using the PAN API.
- Using path metadata.
- Using SCMP `traceroute`.
- Implement a complex path policy that always uses the fastest (low latency) path.

Note The exercise can be split into two mostly independent parts:

- Ping-pong: The ping-ping application requires either access to two servers in the SCION network or it can be run locally using a local topology from scionproto.
- The `PathPolicy` task can be tested in the same setup as the ping-pong application, or it can be tested in the global scion network without access to a destination server. In the latter case, the `PathPolicy` can be used simply to print out the filtered paths.

3.1 Task: Implement ping pong

The first task is to implement a ping-pong client/server, that sends a text string back and forth once. The client and the server should be in different ASes. Sending and receiving can be implemented with JPAN's `ScionDatagramChannel` or `ScionDatagramSocket`.

Hints

There are some complexities regarding the SCION dispatcher and regarding DNS. To avoid these it is recommended to:

- Bind the server to a port in the range 31000-32000
- send/write to using an explicit path from `Scion.defaultService()` `.getPaths(<serverIsdAs>, <serverAddress>)`.
- Use `ScionDatagramChannel` instead of the socket.

Alternative Task

Instead of setting up two machines or a local scionproto topology, the task could be to simply send a packet to `https://echoscion.ddns.net/`. However, this requires access to the SCION production network. Furthermore, it may be difficult to see your packets if too many users send

packets at the same time.

3.2 Task: Implement custom path policy

- Implement a path policy that orders paths by MTU (descending) and excludes all paths with an MTU lower than or equal to the median MTU.
- Apply the path policy to the channel or socket of the ping-pong application.
- Print out the path that is actually used by the client.

Hints

- Examples of path policies are available in the `PathPolicy` class in JPAN.
- After `connect(..)`, the current path is available through `getConnectionPath()`.
- The pretty-printing of paths is available in the `ScionUtil` class.

3.3 Task: Implement traceroute to measure path latency

Use JPAN's `Scmp.newSenderBuilder()` or `Scmp.newAsyncSenderBuilder(...)` to measure traceroute latency from your device to the border router of a remote AS.

Recommended: Test this with large latencies, for example by sending a traceroute to any of the following remote ASes:

- 64-2:0:9 "ETH Zurich (ETHZ)"
- 65-2:0:13 "Anapaya Zurich"
- 65-2:0:19 "Anapaya Frankfurt"
- 66-2:0:10 "Anapaya CONNECT Singapore"
- 66-2:0:11 "Anapaya CONNECT Hong Kong"
- 71-0:0:22f "SWITCH Education"
- 71-2:0:35 "BRIDGES" – Has path metadata
- 71-2:0:48 "Equinix" – Has path metadata
- 71-0:0:51e5 "GEANT"
- 71-2:0:4a "Ovgu"
- 71-2:0:5c "Universidade Federal de Mato Grosso do Sul"

Hints

More ASes can be found here: `https://docs.anapaya.net/en/latest/resources/isd-as-assignments/`

Note: the path metadata also contains a latency field[2]. However, this is the static latency provided by the AS. The metadata latency is measured rarely and does not reflect degradation due to high traffic.

3.4 Task: Implement a low-latency `PathPolicy`

In the previous steps, we learned how to implement a custom `PathPolicy` class and how to detect the latency of a given path. Now we combine these and implement a policy that measures the latency of all available paths and returns the paths ordered by latency (lowest latency first).

The new policy can be used with the application that was developed in the first task.

Hints

- To get the original request path from a SCMP `TimedMessage` you can use `msg.getRequest().getPath()`.
- The solution will have two problems that should be ignored for now:
  - Discovering the fastest route takes some time, which causes a delay before the packet is sent.
  - `ScionDatagramChannel` and `ScionDatagramSocket` will (current implementation) use the `PathPolicy` only when the previous path expires. That means that the lowest latency path detection will run only once.

3.5 Bonus Task: Asynchronous latency measurements

In the previous task we implemented a low-latency `PathPolicy`. The implementation has two major problems: it is called only initially or when a path expires, and whenever it is called, it causes a delay of up to 1 second before it returns the fastest paths.

The current API of JPAN does not (yet) allow using filters that can execute at their own initiative. The task is now to implement a solution that avoids these problems. The solution should result in a `LowLatencyPathSupplier` that can be used as follows:

```
LowLatencyPathSupplier supplier = new LowLatencyPathSupplier(2000, ...);
try (ScionDatagramChannel channel = ScionDatagramChannel.open()) {
  channel.configureBlocking(true);
  String msg = "Hello there!";
  ByteBuffer sendBuf = ByteBuffer.wrap(msg.getBytes());

  for (int i = 0; i < 10; i++) {
    Path path = supplier.get();
    // We use send() so we can explicitly specify the path
    channel.send(sendBuf, path);
    if (path.getMetadata() != null) {
      println("Sent via " + ScionUtil.toStringPath(path.getMetadata()) + ": " + msg);
```

[2]The field may not be present or it may contain a value of -1 if the latency is unknown.

```
        } else {
          println("Sent via " + ScionUtil.toStringPath(path.getRawPath()) + ":" + msg);
        }
        Thread.sleep(100);
      }
    }
```

The idea of the program is that we can sent a message every 100ms while the path supplier determines every 2000ms what the currently fastest path is. The path supplier should work asynchronously so as not not cause any wait when calling `supplier.get()`.

The `LowLatencyPathSupplier` should internally do the following:

1. Get paths from the `ScionService`.
2. Execute tracroute for every path.
3. Order paths by latency.
4. Wait for a configurable delay, for example 2000ms.
5. Repeat with item 1.

Bonus tasks

- The supplier should not request paths every 2000ms; instead, the path should be cached until at least one path expires.
- The `Path` object returned by `supplier.get()` should have metadata.

Hints

The task requires some Java concurrency programming.

- `java.util.Timer` can be useful to schedule regular tasks.
- `CountDownLatch` can be used to wait until we receive responses (or timeout!) for all tracerout requests.
- `ConcurrentSkipListMap` can be used as a thread-safe and ordered 'HashMap'.
- The traceroute sequence ID is unique can be used to map requests to responses.