Interim report

Group 16 #Chili, RPS module, Alexandru Obada and Nicolas Frinker

1. Architecture

The current implementation of the RPS module adopts a layered architecture. It consists of the following layers:

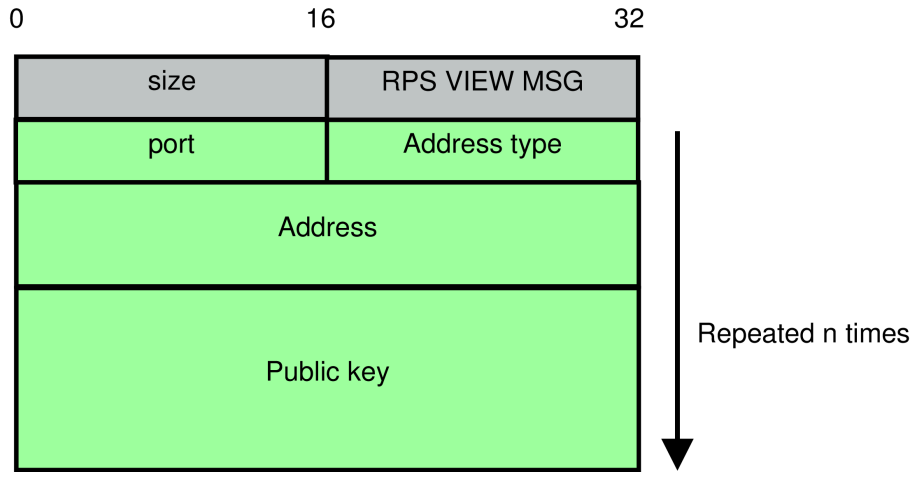
- Sampling layer:
  The services that are part of the sampling process are part of this layer.
- Communication layer:
  Implements NSE, GOSSIP interfaces, as well as RPS interface. Moreover, implements the RPS P2P interface. This layer is available as a service from the sampling layer. The communication layer implements the Observer Design Pattern. The components of the upper layer interested in specific messages have to register to the communication layer for those messages, as a result they are going to be notified.

For the communication with the underlying modules, an event driven TCP client is used. The RPS module consumes the service provided by NSE and GOSSIP modules, for this purpose, two instances of the aforementioned TCP client run in separate threads. The RPS messages are handled by the RPS server, an event driven TCP server, also runs in a dedicated thread. Besides the RPS API, the server also implements RPS P2P API. Both client and server are implemented on Netty framework.
The view persistence is implemented on top of the embedded database H2 with Hibernate as ORM. For increased testability, dependency injection via Spring has been used.

2. Protocol

Our protocol for the RPS module consistst of one message, the RPS View Message.
This message contains the view of the sending peer that can be merged into the view of the receiving peer.
This message will be either sent directly to another peer (as response to our message, pull) or enclosed in a gossip announce message as payload (push).
Every peer will listen on a configurable port for incoming RPS peer messages.

RPS View Message



RPS VIEW MESSAGE will have a value of 542.
Each message contains the common header (size and message type) followed by a list of peers.
The first peer is the source peer, all following peers are peers from the originator's view.

Each peer consists of the listening address (might be IPv4 or IPv6) and port and the public key of the peer. The identity can be derived from the public key.
The address type contains the number (integer) of the used version of the IP protocol (currently either 4 or 6).

Authentication and error handling

We decided to not require any authentication of the peer.
Our own listening socket is just used for receiving RPS view messages from other peers. If we send a message to such a port, we do not care, if this message is acutally received (we do not wait for an acknowledgement) and therefore, we can just send and forget. If this port was the wrong one, our message will just be garbage for the receiver and will be dropped.

In the case of an exception, we will just log the event (if detected) and ignore the incoming packet, if there is any. Since we will receive a lot of rps view messages periodically, we do not have to rely on every single message to be received successfully. Missing messages will not do any real harm.

Possible future extensions

Even though not implemented and planned yet, there are somesecurity improvements, we might consider to include in the protocol in further development. But as we are concentrating on the random sampling feature of the module first, we will look at them in more detail at a later time in our development cycle.
First, we consider encrypting the RPS view message send to our listening port using the receivers public key (that is included in the message).
Secondly, we are evaluating the usage of a secret token contained in each push message that each answer to this push (i.e. each pull message) is required to send back to the originator. This way we would make sure, that only identified peers in the network may send us their view, as only these peers will receive our push message and the token that is included.
Thirdly, we consider adding a signature to each message.