Midterm Report

Andreas Bauer
andi.bauer@tum.de

Chung Hwan Han
hanc@in.tum.de

September 2022

1 Objective

This paper details the final state of the *Gossip-10* team project as part of the *Peer-to-Peer Systems and Security* course. The Gossip module is part of the *VoidPhone* Voice over IP system. VoidPhone provides anonymity and unobservability to voice communication facilitated through a peer-to-peer architecture. The Gossip module is used to spread information like availability information among communicating peers. Other modules from other teams might rely on the Gossip module to spread information required for their operations through the network. For this purpose, the project specification defines a socket-layer API for inter-module communication. The protocol spoken between Gossip instances is the core product of this project work. We follow the material and contents taught in the lecture – considering best practices, common attacks, and security measures – for the instantiation of our P2P protocol.

2 Requirements

Within the work spent on the project, we derived several requirements set against the system. This section provides an overview of what the Gossip is supposed to do and what we identify as essential for our ongoing design decisions.

As roughly outlined above, Gossip is responsible for spreading information in the peer-to-peer network. To do so, Gossip establishes connections with several other members of the network. Each peer maintains a public-private key pair[1] – referred to as *hostkey* – which is used for peer identification and trust verification. By specification, we assume that the public keys of the hostkeys are exchanged out-of-band beforehand between all members. The combination of hostkeys public key and the remotes ip address, form an identity on the Gossip layer.

Our developed protocol can currently be divided into two phases: *handshake* and *knowledge-spreading*. For the handshake, we identify the following requirements:

- Verify the identity and authenticity of the remote peer.
- Establish a secure channel, providing confidentiality and integrity to the communication and preventing typical attacks like message replays.
- Ensure that as much meta-information as possible is protected (e.g., outside attackers can't easily trace who is communicating/connecting with whom except through information leaked by lower layers).

While knowledge-spreading is specified to be best-effort and doesn't need to make any guarantees, we nonetheless want to deliver the best possible performance and protection against typical attacks. Some attacks are already out-ruled by design by the specification (e.g., information is always validated by the upper layer before continuing to spread information; or knowledge

[1] A 4096-bit long RSA keypair.

is spread to the whole network and not to single identities). Still, information may be lost if an Eclipse attack is staged and participating attacker peers drop all messages of a particular data type. Therefore, measures must be taken so that attackers don't have significant enough control over who is initiating a connection to whom. For routing itself, we must consider that we are maintaining an unstructured network and therefore need to take measures that routing packets aren't traveling forever in cycles.

Other modules interact with Gossip through the specification-defined TCP socket interface. API consumers will register data types they are interested in to be notified about. They may announce data into the network for those registered data types. For incoming data, Gossip delivers it to all registered API clients. Once each of those validated the data, information will be spread further into the network.

3 Architecture

In this section, we will detail the architecture of our implementation. Figure 1 provides a rough overview of our project's architecture. It uses UML-like syntax to highlight the core components of the module and its interfaces between each other and the outer world. Each of these components are explained in detail in the following subsections.

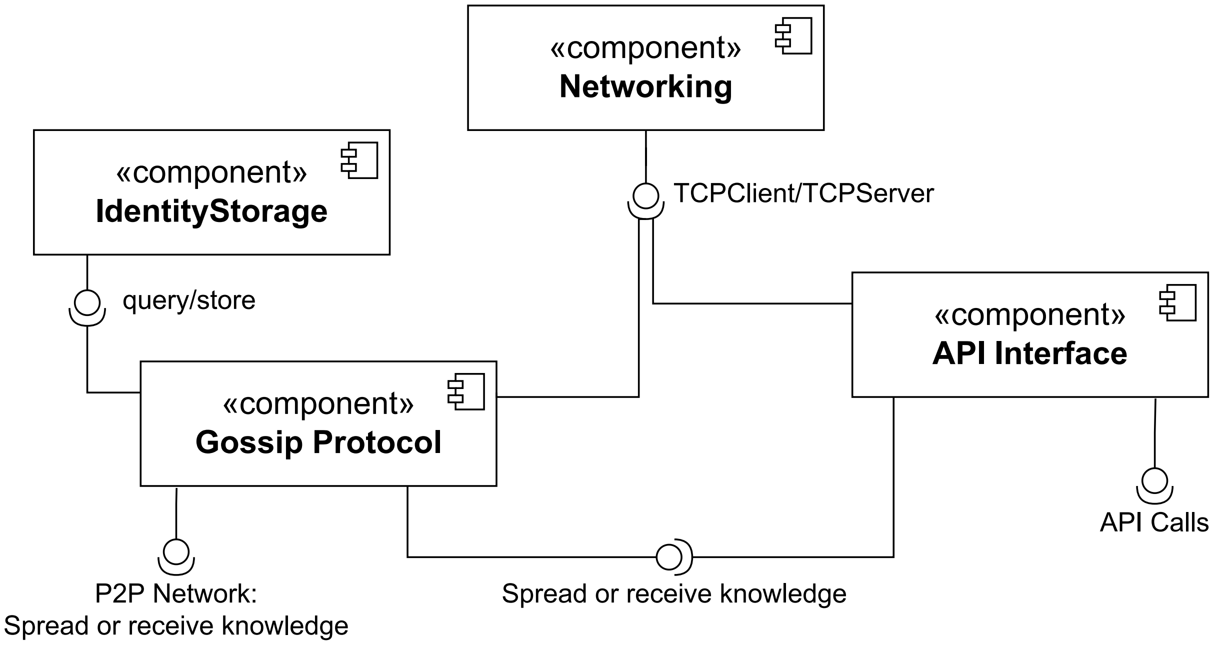

Figure 1: UML-like component diagram depicting a rough subsystem decomposition of our system. It highlights the fundamental structure and architecture of our project.

3.1 Networking

The **Networking** component is built on top of the *Netty 4*[2] networking framework. It provides common abstractions used by both the API and the Gossip component. Namely, it handles packet encoding and decoding, and connection establishment.

Packet serialization and deserialization can be implemented by conforming to the `OutboundPacket` or `InboundPacket` interfaces (defining a respective `InboundPacketHandler`, respectively). The set of supported packets can then be built using `ProtocolDescription` by supplying the packet implementations and their corresponding packet ids.

[2]https://netty.io

Together with an `EventLoopGroup` instance, `ProtocolDescriptions` can be used to instantiate a `TCPClient` or `TCPServer` to bind the respective TCP socket. `EventLoopGroup`[3] is a concept provided by Netty, which manages a thread pool to handle incoming connections and serialization of traffic asynchronously.

3.1.1 Common Packet Format

The `ConnectionInitializer` (automatically set up within the `TCPClient` or the `TCPServer`) constructs the channel pipeline for each connection. The resulting channel pipeline is depicted in Figure 2. Incoming traffic passes through the `LengthFieldBasedFrameDecoder`[4] (parsing the *size* field) and the `PacketDecoder` (parsing the *packet id*, assembling the typed packet instances) to the `InboundHandler`, calling the corresponding user code to handle the packet. Outbound traffic passes the `PacketEncoder` (serializing the packet contents and writing the *packet id*) and the `LengthFieldPrepender` (writing the *size* field).

The illustrated channel pipeline results in the general packet header depicted in Figure 3. The *size* field is an unsigned 16-bit integer, resulting in a maximum packet size of 65535 bytes (including the header size).

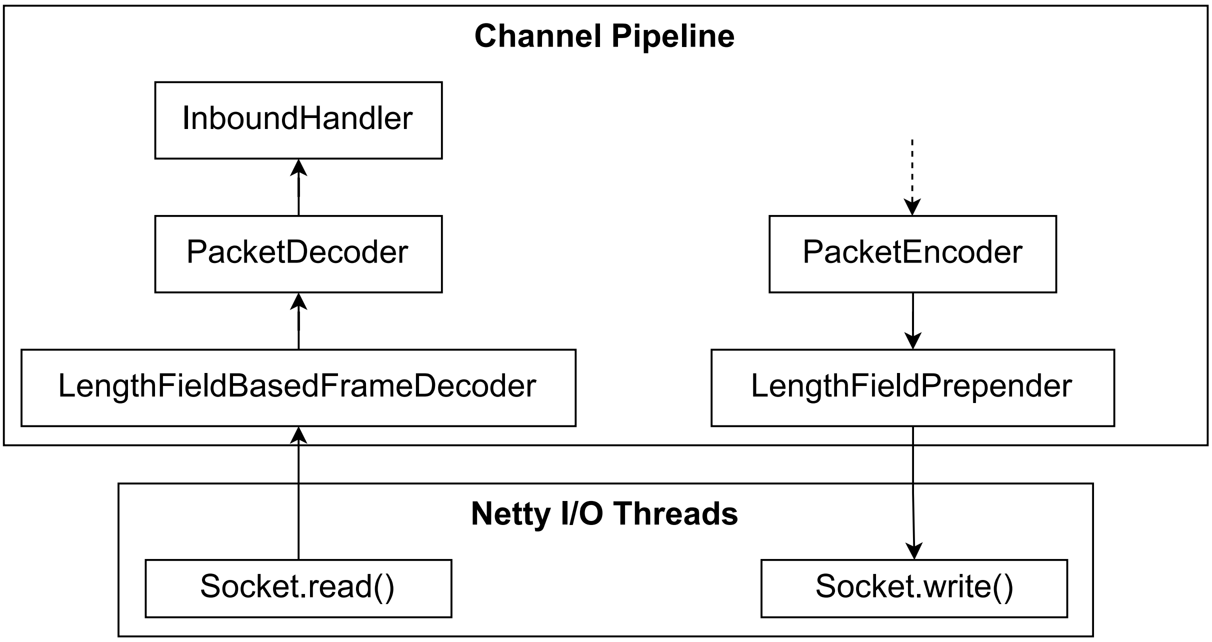

Figure 2: Diagram of the Channel Pipeline of an open connection.

3.2 API Interface

The **API Interface** implements the TCP socket-based API used by other modules to interact with Gossip (see section 2). Packet definitions are provided for the four API messages `ANNOUNCE`, `NOTIFY`, `NOTIFICATION`, and `VALIDATION`, as outlined in the specification. The component relies on the networking abstractions introduced in subsection 3.1.

Incoming API message calls are forwarded to the `GossipModule`, which is part of the *Gossip Protocol* component. The module is provided with a connection handle to send out potential notifications later.

[3] https://netty.io/4.1/api/io/netty/channel/EventLoopGroup.html
[4] Provided by Netty: https://netty.io/4.1/api/io/netty/handler/codec/LengthFieldBasedFrameDecoder.html

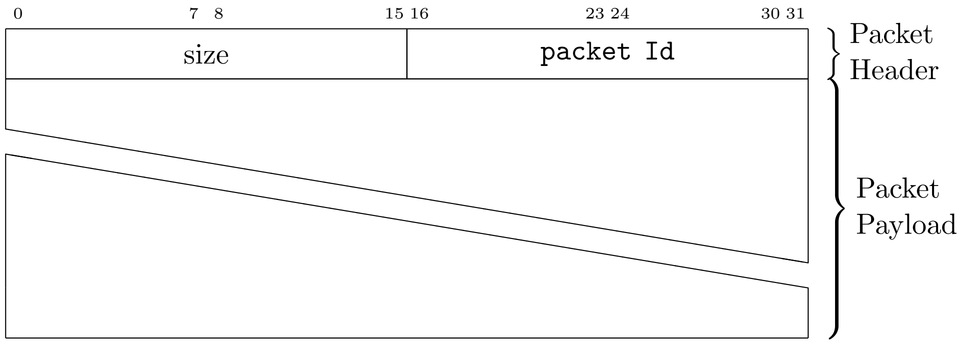

Figure 3: Packet header layout used within all packet types.

3.3 Gossip Protocol

The **Gossip Protocol** component implements the protocol spoken between individual Gossip peers. We rely on TCP to create a strongly connected network and rely on retransmissions and congestion control.

As outlined in the project specification, hostkeys are distributed out-of-band. Therefore, we currently don't consider identity exchange (see subsection 4.1). Known identities with their respective public keys and last known connection information are stored on disk inside the `identities` folder controlled by the **IdentityStorage** component. For testing purposes, we supply the `generateHostKey.sh` script to generate hostkeys for testing purposes. Further, the command line interface of our application provides means to generate and import identities in the required format (refer to the `README.md` for more information).

Our protocol consists of two phases, the initial *handshake* phase and the *knowledge-spreading* phase for established connections. The `DISCONNECT` is the only packet valid in both phases and which might be sent at any time. Its structure is depicted in Figure 4. It consists of a single *reason* field describing the disconnect reason. We currently maintain the following possible reasons: `NORMAL(0)`, `UNSUPPORTED(1)`, `AUTHENTICATION(64)`, `UNEXPECTED_FAILURE(65)`, `CANCELLED(66)`, `DUPLICATE(67)`, `BUSY(68)`, `NOT_ALLOWED(69)` and `TIMEOUT(70)`. Please derive their exact meaning from the technical documentation of the `GossipPacketDisconnect.Reason` class.

0 … 7	8 … 15	16 … 23	24 … 31
size		`DISCONNECT`	
reason	reserved		

Figure 4: GossipPacketDisconnect

The two protocol phases are described in detail in the following two subsections.

3.3.1 Handshake

For the handshake, we recall the requirements set in section 2. The primary purpose of the handshake is to create a secure channel for communication and verify and authenticate identity claims of the peers mutually.

We rely on *TLS 1.3*, serving us with a reliable, secure, and well-tested secure channel implementation. Consequentially, we employ ECDHE (Elliptic Curve Diffie Hellman with an Ephemeral key) for the key exchange, providing perfect forward secrecy for ongoing traffic. For application-data, encryption we configure the AEAD cipher *ChaCha20-Poly1305* supporting our confidentiality and integrity security goals.

Our TLS layer is configured to do mutual authentication, meaning both the server and the client provide respective certificates. The root of trust for a peer's certificate chain is derived from its hostkey (a self-signed certificate containing the public part of the hostkey). This certificate acts as a Certificate Authority to an intermediate certificate used within the TLS authentication phase. Within our `TrustManager` implementation, we verify the integrity of the certificate chain and ensure that the root certificate is self-signed with the expected hostkey identity. Certificate transmissions are encrypted with TLS 1.3. Therefore, this doesn't leak the peer's identity at the connection establishment. Each peer verifies that the TLS session was established with the expected hostkey ensuring that no man-in-the-middle attack is in progress.

After completing the TLS handshake, we continue with the gossip handshake used to establish our application-domain-specific constraints and expectations.

Handshake Hello After the TLS handshake completes, the connection-initiating peer sends the `HANDSHAKE HELLO` (see Figure 5) packet to the server. The packet specifies the used protocol *version* (currently always `VERSION_1(1)`). While the packet does not transport any elementary information, it marks the explicit begin of the gossip protocol.

Upon receiving the packet, we check our local identity storage, to ensure that we only accept connections from peers that are part of our identity storage. Further, we verify, that the remote peer's ip address matches the expected ip address of the locally stored identity (established out-of-band as explained section 2). A future version of our Gossip implementation might employ a Proof-of-Work (PoW) mechanism to update the expected ip address to provide a more robust peer-to-peer network implementation (see section 4). Lastly, we impose rate limiting on the amount of connection attempts that can be made from the same ip address. This is to reduce the attack surface of Sybill attacks, making it harder to stage such attacks from a single machine.

0 … 7	8 … 15	16 … 23	24 … 31
size		`HANDSHAKE HELLO`	
version	reserved		

Figure 5: GossipPacketHandshakeHello

Handshake Complete The `HANDSHAKE COMPLETE` (Figure 6) is the explicit signal to the client that the previous validation steps were successful and that the client should also switch to the *Knowledge Spreading* phase. The packet does not have any contents.

0 … 7	8 … 15	16 … 23	24 … 31
size		`HANDSHAKE COMPLETE`	

Figure 6: GossipPacketHandshakeComplete

3.3.2 Knowledge Spreading

Once clients complete the *Handshake* phase, they reside in the *Knowledge Spreading* phase and are active participants in the Gossip peer-to-peer network.

Once an API consumer subscribes to a data type (see subsection 3.2), it may announce data into the network. To do so, we overtime propagate the `SPREAD KNOWLEDGE` packet to all connected peers (see Figure 7). For each announced data point, we generate a random 64-bit *message id*. In combination with the *ttl* field (defined by the API consumer and decremented at every hop), it provides means to avoid cycles in packet routing.

On arrival of a `SPREAD KNOWLEDGE`, we provide the data contents to all API connections subscribed to the data type. If there are none, we discard the packet. Otherwise, we only forward the packet further into the network if all subscribed API connections report validity of the packet contents. This ensures that we don't propagate manipulated information into the network.

We impose some rate limiting restrictions on the amount of knowledge a remote peer can spread into the network. This is to reduce the attack surface on potential denial-of-service attacks (e.g., by changing the message id).

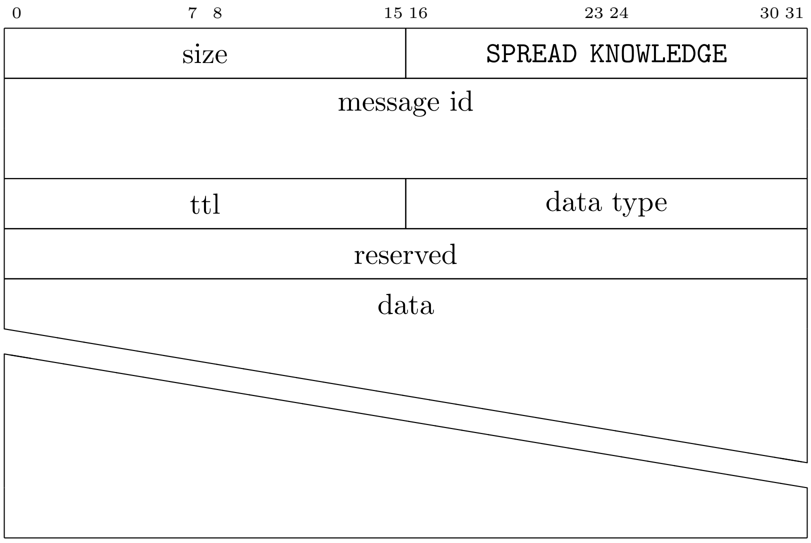

Figure 7: GossipPacketSpreadKnowledge

4 Future Work

This section details future work we see necessary to view the project as completed.

4.1 Gossip Protocol

Lastly, as specified, hostkeys and connection information are assumed to be exchanged out-of-band. Future work could research solutions for session onboarding as well as means to spread information about known peers into the network. We already discussed, that we could employ a PoW-based mechanism to update the peer's associated ip address (see subsection 3.3). Similarly, a PoW-based process could be implemented to do session onboarding. Considerations will have to be taken to reduce the effective attack surface of an Eclipse attack.

4.2 Infrastructure Related Improvements

The project setup needs some minor infrastructure-related adjustments. It was intended to employ a linter within the CI pipeline, however the respective PR was not completed (see #3).

5 Workload Distribution

We would like to refer to the milestones outlined in the initial report when explaining our workload distribution.

- **Project Setup**: The fundamental Java project setup including the Gradle build configurations were configured by Andreas. Additionally, Andreas built the CI pipeline to automate the tests and ensure the build integrity. We employed a PR-template to make pull request easier to draft and easier to review.

- **API**: Andreas has designed and implemented overall structures of the core networking layer, which are used by API and P2P connections. Chung Hwan implemented the serializing and deserializing methods for the API message formats.
- **P2P Protocol**: Andreas designed the handshake protocol and implemented it with including the definition of packet formats and handler logics. The business logic (implemented within the `GossipModule`) were implemented by Andreas.
- **Tests**: Chung Hwan wrote the in-memory test cases for the packet decoding and encoding. Andreas wrote the end-to-end test cases to ensure the network layer works as intended.
- **Documentation & Reports**: Chung Hwan and Andreas co-worked writing the midterm report. Andreas updated its contents to form the endterm report.

5.1 Individual efforts

The overall git commit history provides an additional source for the individual workload distribution in addition to the information outlined below.

Andreas: Since the initial report, I have roughly spent 85 hours in total on the project. This metric is provided by the time tracking tool of my IDE (code and reports). It, therefore, doesn't account for time spent outside the IDE (e.g., time spent with research, lecture material or online material around the topic or tools).

Chung Hwan: I invested about 30 hours for this project after submitting the initial report. Since this metric is calculated from commit history, it includes only the time the code was written.

6 Changes

With the final state of the project, we didn't identify any severe deviations from the project architecture we anticipated in the initial report.

However, there were slight changes in the list of dependencies we used within the project. Namely, we added the following new dependencies:

- **Guava**[5] is a commonly used utility library developed by Google, which we use in several locations.
- **Caffeine**[6] is a high-performance, near-optimal caching library that builds upon the Guava Cache API. We use it within the `GossipModule` for data structures to enforce a time- or space-based expiry on its contents.
- **JCommander**[7] is a library to parse command-line options in Java applications in a declarative way. (previously, in the midterm report, we used Apache Commons CLI before we switched to JCommander).
- **Gson**[8] is a library from Google to make JSON Serialization and Deserialization easy.
- **commons-lang**[9] is a library from the Apache Software Foundation that provides us with infrastructure to easily realize rate limiting.

[5] https://guava.dev
[6] https://github.com/ben-manes/caffeine
[7] https://jcommander.org
[8] https://github.com/google/gson
[9] https://commons.apache.org/lang

- **commons-configuration2**[10] is a library from the Apache Software Foundation that we use to parse the Windows INI file format. This replaces the *init4j* library, our previous choice for INI file parsing. We replaced it, as it wasn't able to parse properties on the global section (e.g., the *hostkey* option).

[10] https://commons.apache.org/proper/commons-configuration/