IN2194
Peer-to-Peer Systems and Security
Summer Semester 2023

Midterm Report

Team Gossiphers (Gossip-1):
Yannis Matezki and Dominik Ritzenhoff

Date Submitted: Jul. 04, 2023
Date Updated: Jul. 27. 2023

Changes to Initial Assumption

Use of external libraries:
- go.uber.org/zap - Structured Logging library for more convenient logging
- gopkg.in/ini.v1 - Ini Parser for the given configuration file

All other original assumptions from the Initial Report still stand

Architecture of the Module

Logical Structure

Go does not offer classes, but structs with method implementations offer a similar functionality.
Our module is separated into a few logical units:
- API
    - TCP Server that answers API requests defined in the given specification
    - API Packet serialization/deserialization
    - Basic logic for API clients to listen to events of a certain type
    - Structs used like classes:
        - *Server*
            - Stateful TCP API server, stores connections registered for events
- Config
    - Configuration values are read from a .ini file, which serve as adjustable parameters within the program.
- Gossip
    - UDP Server and Client that answer and initiate P2P push and pull requests defined within the Brahms algorithm, respectively.
    - P2P Packet serialization/deserialization.
    - Structs used as classes:
        - *View* - Represents a view within Brahms algorithm.
        - *Sampler* - Represents a sampler within Brahms algorithm.
        - *SamplerGroup* - Represents a group of samples belonging to one node.
        - *Node* - Represents a node (i.e. peer) within the network.
        - *Gossip* - Represents the collection of data structures needed to make the Brahms algorithm function.
        - *Server* - Stateful TCP P2P API server. Stores connections registered for events.
- Challenge
    - Module to generate computational challenges in the required format and check the validity of the produced solutions
    - Utilized a ring of rotating keys to invalidate old challenges after a given timeframe without storing each generated challenge
    - Structs used like classes:
        - *Challenger*
            - Stores internal key ring and automatically rotates keys in a given time frame
            - Generates challenges and validates solutions using the internal keys

Process Architecture

Goroutines are utilized for all internal processes acting as lightweight threads managed by the Go runtime. Each TCP server is running in its own Goroutine and spawns a new Goroutine for each incoming connection to allow for multiple parallel connections. Goroutines are also utilized for periodically executed tasks like the key rotation of the challenge generator.

Networking

Packets that are not formatted correctly are simply dropped, a flooding of well formatted packets is currently not prevented, but the local view should not be impacted due to other security measures.
Communication between peers is UDP-based and assumes the network to be unreliable.
Each peer locally tracks the state of communication flows with other peers and which packets to expect within a certain time frame. Packets that are not expected (i.e. unsolicited Pull Responses) are simply ignored.
Each packet is encrypted using the receiver's public key and additionally contains the sender's identity in the header and a signature, signed with the sender's private key in the last 64 bytes.
The receiver decrypts the packet and verifies the signature. Only if the signature is valid and the packet is expected, it is further processed.
The signature signs the entire packet, excluding only the last 64 bytes (the signature itself).
Each row in one of the following tables is 32 bits wide.

General Structure:

<table>
<tr><td>size</td><td>Packet Type</td></tr>
<tr><td colspan="2">Sender Identity (32 bytes)</td></tr>
<tr><td colspan="2">Data (optional)</td></tr>
<tr><td colspan="2">Signature (64 bytes)</td></tr>
</table>

Gossip Ping:

- Sent to probe another node. Another node should reply with a GOSSIP PONG if available.
- Packet Type: GOSSIP PING = 0x00 0x30
- No data

Gossip Pong:

- A response to the ping, indicating that this node is still alive.
- Packet Type: GOSSIP PONG = 0x00 0x31
- No data

Gossip Pull Request:

- Sent to request a list of nodes from another node, the other node should respond with a GOSSIP PULL RESPONSE
- Packet Type: GOSSIP PULL REQUEST = 0x00 0x40
- No data

Gossip Pull Response:

- Response to the pull request with the node's view included in the body.
- Packet Type: GOSSIP PULL RESPONSE = 0x00 0x41

nodes (List of peers, Identity and Address separated by tab (\t), peers separated by newline (\n)) → <Identity>\t<Address>\n

Gossip Push Request:

- Request a challenge to push own address to another node, other node should respond with a GOSSIP PUSH CHALLENGE to this
- Packet Type: GOSSIP PUSH REQUEST = 0x00 0x50
- No Data

Gossip Push Challenge:

- Response to the push request. Includes a challenge token for the destination node to 'solve' in order to have its gossip push request acknowledged.
- Packet Type: GOSSIP PUSH CHALLENGE = 0x00 0x51

Difficulty
Challenge (32 bytes)

Gossip Push:

- After solving the challenge received in a GOSSIP PUSH CHALLENGE packet, a node can then continue to send its own ID/address to another node with the nonce that acts as the solution to the challenge
- Packet Type: GOSSIP PUSH = 0x00 0x52

Challenge (32 bytes)
Nonce (8 bytes)
<Identity>\t<address>\n Of own peer

Gossip Message:

- Contains a gossip message that should be spread to all nodes in the local view when received from a known peer
- TTL should be decreased every time the message is forwarded, a message with a TTL of 1 is not forwarded further
- Packet Type: GOSSIP MESSAGE = 0x00 0x60

TTL	reserved	data type
data		

Security Measures

Every message exchanged between peers contains the sender's identity and is signed by the sender using its public key. Additionally, it is encrypted using the receiver's public key. The receiver discards any packet that it cannot decrypt and consecutively verify the signature of. An attacker can still guess the packet type based on its size, but both integrity and confidentiality of the contents is guaranteed.
As seen in the data blocks above, we also include a gossip push challenge to make it more difficult for a group of malicious peers to push-flood another peer using i.e. a sybil attack.
This challenge is generated using the other peers' identity.
Additionally only one push from each identity is allowed in one Brahms Gossip cycle.
This limits the amount of pushes an attacker can send with respect to available compute and network addresses.
Finally the parsing is made to expect malformatted messages and deal with them safely by only receiving and parsing an expected, limited amount of bytes.

Future Work

Gossip implementation is currently being finalized, and it will adhere to the previously mentioned specifications. Additionally, we would like to adjust the size of the view every round by integrating with the Network Size Estimator.

Workload Distribution

Together:

- We collectively wrote the reports, designed the protocol and came up with a design for the gossip implementation specification.
- Reviewing each other's work in the form of Merge Requests

Yannis:

- API Server (Packet parsing, Client registration, etc.)
- Small Config module providing parsed configuration values and defaults
- Brahms Sampler and SamplerGroup for historical view samples
- Challenge module for generation, solving and validation of PoW challenges
- GitLab CI/CD setup for compilation and container creation

Dominik:

- Brahms View
- Gossip rounds to update the view with received Pushes and requested Pulls (WIP)
- P2P API implementation as previously outlined.
- The rest is currently W.I.P.

Effort spent for the Project

In total we spent a combined amount of roughly 10 working days on the project. Currently 7 of those fall on Yannis, while 3 fall on Dominik. We expect this to even out in the near future, since Dominik currently has a lot of his work in progress that depended on some implementations done by Yannis.