CS-438 - Decentralized Systems Engineering, Fall 2021

Homework 1

Gossip, Routing, and Private messaging

Author	DEDIS lab, EPFL
Revision	September 2021 - 1.0.0
Publish date	Friday, October 1, 2021
Due date	**Tuesday, October 26, 2021 @ 23:55**

Table of contents

Table of contents 1

Introduction 1

- End result 1
- Objectives 1
- Broadcast with Gossiping: challenges 1
  - Using an unreliable transport layer 2
  - Nodes only know a subset of the participants 2
  - The topology is not fixed 3
- Routing 3

Your tasks 4

- Before you begin: New messages introduced 4
  - RumorsMessage 4
  - StatusMessage 5
  - AckMessage 5
  - EmptyMessage 5
  - PrivateMessage 6
- Task1: Implement the broadcast function 6
- Task 2: Process RumorsMessage 7
- Task 3: Implement the anti-entropy mechanism 7
- Task 4: Process StatusMessage 8
- Task 5: Process AckMessage 9
- Task 6: Implement the routing update 9
- Task 7: Implement the heartbeat mechanism 9
- Task 8: Implement the private message mechanism 10
- Hint: Sequence diagram 10

Try your program 12

Appendix A - Tests definitions 13

- Broadcast 13
- Anti-entropy 14
- Heartbeat 14
- ContinueMongering 14
  - Set to 1 14
  - Set to 0 15
- Catchup 15
- Ack 15
  - With Ack 16

- Without Ack 16

BigGraph broadcast 16

Private message 17

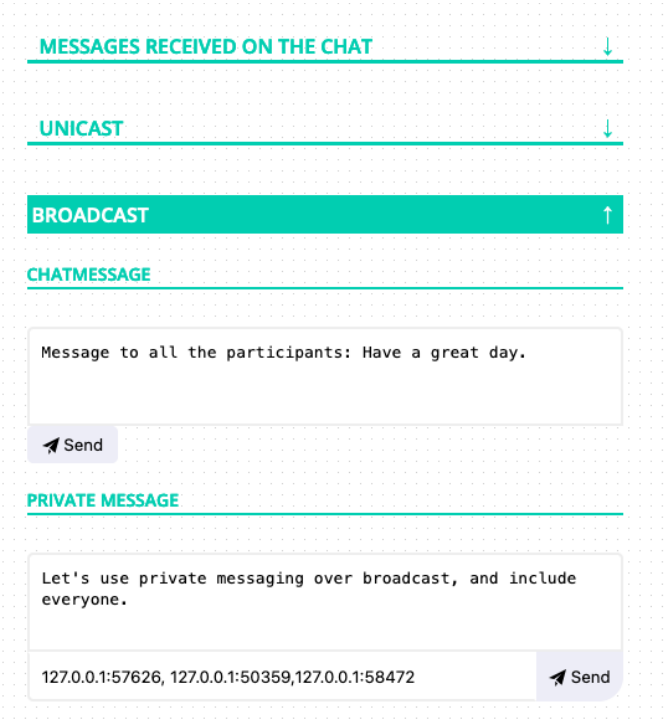
- Broadcast 17
- Unicast 17

Integration test 18

Introduction

End result

At the end of this homework, your Peerster system will be able to spread messages to all the peers using a gossip protocol. You will use this broadcast mechanism to exchange two types of messages: *chat messages* and *private messages*. Chat messages are sent to all participants. Private messages will allow you to target specific recipients. Concretely, Peerster will be able to not only send a unicast chat message to a specific peer, as was the case in HW0, but also to all other peers in a broadcast manner, by using the underlying broadcast mechanism and all the nice properties it offers. See figure 1 to get an idea of how the broadcast messaging will be used in practice with the web GUI, as well as the private one.



Message to all the participants: Have a great day.

Let's use private messaging over broadcast, and include everyone.

127.0.0.1:57626, 127.0.0.1:50359,127.0.0.1:58472

Figure 1: End-result of HW1. The peer is able to broadcast a message to all participants. It can also send private messages over the broadcast mechanism.

Objectives

Understand and implement a gossip protocol on top of what you implemented in HW0.

Broadcast with Gossiping: challenges

You are going to implement a gossip protocol. This kind of protocol is well suited in Peerster, given the following constraints of Peerster:

The transport layer is unreliable: In HW0 you implemented a socket using UDP. This protocol is unreliable in the sense it can lose packets without the sender being notified. This prevents us from making any assumption about a peer's message being correctly received by its neighbor[1].

Nodes only know a subset of the participants: Think of the Peerser system like a big party you've been invited to. You don't know all the participants personally, but the friends you know at that party can connect you to all the participants via one or more levels of transitive friendship. Therefore, if you wanted to spread a message to all the party's participants, you would only have to tell your friends to tell their friends, to tell their friends, and so on. This is handy because you don't need to speak to all participants directly, but just to your friends. However, this implies you rely on the cooperation of others to spread your message.

The topology is not fixed: Remember the party ? Well, it also happens that the participants can join as they want, in unpredictable ways. Therefore, it poses the problem of being sure the messages you spread also reach the participants joining late. We want all participants to eventually hear every message from everyone, even if it joins late. Since this is a great party, we assume that nobody leaves it. For simplicity, in Peerster if a peer doesn't respond we assume that the network is slow instead of the peer having failed, so nobody removes that peer.

Given those constraints imposed by our system, here are two properties we want to ensure:

1. Every participant must eventually receive every message, even if it joins late.
2. Participants don't have to know every other participant. As long as the peer graph is connected, this is sufficient for every node to send a message to all other nodes.

In the following section we discuss how to solve the aforementioned issues from a high-level perspective. Technical details will be discussed in the tasks section.

Using an unreliable transport layer

This problem will be fixed with the introduction of an acknowledgement mechanism. When peer A sends a message to peer B, B will have to send back an ack message to A, and A will have to wait until some timeout for an ack message from B. If not, A will have to consider the message as not being successfully delivered to B.

Nodes only know a subset of the participants

We considered the graph to be connected but not necessarily a complete graph. This means each participant only knows a subset of the other participants. Nevertheless, every participant should still be able to reach every other participant. This means two things: (1) A

[1] From a peer's point of view, its neighbors are all the peers it is directly connected to. If A knows B's contact details, then A has B as a neighbor.

participant must accept to relay a message from one of its neighbors to another, (2) a participant must accept to spread special messages from others that we call *rumors*. A *rumor* is a message that is intended to be spread to everyone in the system. Regarding (2), we will use a probabilistic mechanism to spread rumors, as well as an “anti-entropy” mechanism, where participants periodically share their view[2] on the system to others and update their view accordingly. The core idea is that nodes maintain for themselves their view on the system, randomly share new rumors to their neighbors, and periodically check that they are up-to-date with the latest “news”.

The topology is not fixed

Participants can join the network freely. Joining the network means for a new participant to connect to known participants. However, if A simply joins by connecting to a known participant, it doesn’t mean that A can immediately receive messages. If I know my friend John at a party, it doesn’t mean that John, nor the other participants, know that I arrived. To enable A to make itself known, we introduce the notion of “heartbeat” messages sent periodically by participants to announce their presence at the party, i.e. in the Peerster system. Sending it at a regular interval allows peers to keep a fresh state of the network in their routing tables (keep fastest relays).
For the sake of simplicity, we assume that participants never leave[3].

Routing

Every participant should know how to reach every other participant. If not by a direct link, it should know a node that can relay the message. To do so each participant maintains a next-hop routing table, as already set up in HW0. Until now the routing table could only be updated by adding a peer or setting manually an entry. Since we don’t want to manually update the routing table for each node, we are going to use a flavor of Destination-Sequenced Distance Vector routing - DSDV. This routing scheme offers a simple and efficient way to maintain a routing table. The main challenge is to avoid an endless loop: John’s messages to Bob are relayed by Alice, and Alice’s messages to Bob are relayed by John.

[2] A view on the system is all the rumors a peer has seen so far. It represents what a peer knows about the system’s state.

[3] In a realistic system, participants could leave, which in an extreme case could partition the network. To avoid that, participants usually keep enough neighbors so that it’s improbable that all their neighbors fail at the same time and disconnect the network.

Your tasks

In the following tasks you are first going to implement the broadcast function, and then all the mechanisms and messages needed to make it work. We describe how to process each kind of new message that your peer can receive.

The term “process” refers to the handler associated with a specific message type. Recall that your peer first receives Packets on its socket, and then uses the message registry to process that packet, which calls the handler associated with the message stored in the packet. Therefore, when we say “process message X”, we mean “implement and register the handler associated with message X”.

In summary, here is what needs to be done:

- □ Implement the broadcast function
- □ Process RumorsMessage
- □ Implement the anti-entropy mechanism
- □ Process StatusMessage
- □ Process AckMessage
- □ Implement the heartbeat mechanism
- □ Implement the routing update
- □ Implement the private message mechanism

Before you begin: New messages introduced

Before going into the technical details, here we introduce you to the new types of message Peerster is going to use. Those messages are defined in **`types/messaging_def`**. Until now you had only one kind of message, the **`ChatMessage`**. This homework introduces 5 new message types.
Recall that every packet has a unique identifier `PacketID` in the packet header.

RumorsMessage

A **`RumorsMessage`** contains one or more **`Rumor`** messages. A **`Rumor`** is a message a peer wants to broadcast to everyone. As illustrated in figure 2, it contains the initiator of the rumor, i.e. the peer’s address that created the rumor, a sequence number, and the message to be spread. A **`Rumor`** can be seen as a wrapper around any kind of message that needs to be eventually distributed to all peers. Therefore, once a **`RumorsMessage`** is sent by a peer, it can expect all the rumor messages from the list of **`Rumor`** to be eventually distributed and processed by every peer.

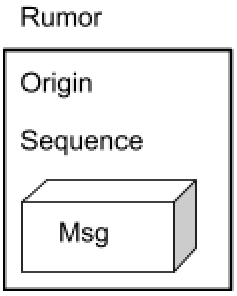


Figure 2: A rumor message is made of an origin, sequence number, and an embedded message. A rumor is eventually received by all peers and its embedded message processed.

The Origin of a Rumor is the socket's address of the peer that created the Rumor. The `Sequence` must always correspond to the number of rumors the rumor's creator has created so far, including the one being created. This number is equal to 1 when a peer creates its first rumor, 2 for its second rumor, etc… A `Rumor` has one and only one creator that never changes.

StatusMessage

This message contains a node's view on the system. By "view" we mean all the rumors the node has processed so far. That is, the node's view contains, for each known peer, the sequence number of the last Rumor processed from that peer. The status allows peers to compare their views and get/send updates from/to other peers.

AckMessage

This message is sent back by a peer to acknowledge that it correctly processed a RumorsMessage. As illustrated with Figure 3, it contains the identifier PacketID (a randomly generated string, see the code skeleton) of the RumorsMessage it acknowledges, as well as the status of the replying peer. This status is not useful for the acknowledgment itself. Instead, it is a handy way for peers to compare their views while doing an acknowledgement at the same time.

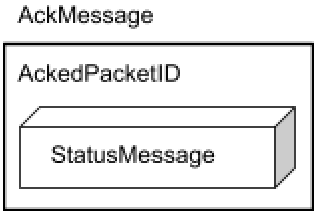


Figure 3: An AckMessage specifies the PacketID of the message to acknowledge, and a StatusMessage.

EmptyMessage

This message, as its name suggests, doesn't contain anything. It is used to send empty rumors to participants in order to periodically announce themselves:

- Eh, you !

- Me ?
- Yes, you
- Ok, what ?
- Nothing, bye
- *… Well, at least I know this annoying guy exists... Let’s spread the rumor!*

PrivateMessage

As illustrated in figure 4, a PrivateMessage specifies a bag of recipients, as well as the message recipients should process. In other words, this message wraps another message with a mechanism to tell who should process the embedded message. Like so, one can use a private message in a rumor. The PrivateMessage message type will be processed by all the participants, but the message embedded in the private message will only be processed by the recipients mentioned by the private message.

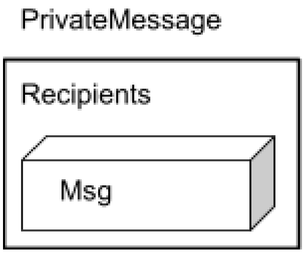


Figure 4: A private message specifies a bag of recipients and an embedded message.

Note that the notion of privacy here is pretty weak and simplistic, as it doesn’t prevent anyone from processing the embedded message even if it is not in the bag of recipients. Could you think of a better way[4] ?

Task1: Implement the broadcast function

This function is defined in the Messaging interface and is already added in the skeleton from HW0. Broadcasting a message means two things for a peer:

- Create a RumorsMessage containing one Rumor (this rumor embeds the message provided in argument), and send it to a random neighbor.
- Process the message locally.

To process the message locally, you should create a `transport.Packet` message and call the `registry.ProcessPacket` with it. See listing 1.

```
header := transport.NewHeader(node.addr, node.addr, node.addr, 0)

// let's execute the message (the one embedded in a rumor) "locally"
pkt := transport.Packet{
    Header: &header,
```

[4] The message should be encrypted with only the recipient’s private key being able to decrypt it. Shamir's Secret Sharing could offer that kind of primitives.

```
    Msg:    &msg,
}

err = m.node.registry.ProcessPacket(pkt)
if err != nil {...}
```

Listing 1: Processing the packet locally in the broadcast function.

Note that you will have to later update this function with the Process AckMessage task.

Task 2: Process RumorsMessage

When a peer receives a rumor message it must do the following:

- Send back an AckMessage to the source.
- Process each Rumor Ʀ by checking if Ʀ is expected or not. Ʀ is expected if Ʀ's sequence corresponds to the last Rumor's sequence + 1 we got from Ʀ's origin. If Ʀ is expected then process Ʀ's embedded message. If not, ignore it. For example, if Ʀ's sequence corresponds to the last Rumor's sequence **+ 2**, then the node still ignores Ʀ.
- Send the RumorMessage to another random neighbor in the case where one of the Rumor data in the packet is new.

When you process Ʀ's embedded message you must use the registry. To do so create a new `transport.Packet` using the same `Header` and the embedded message from the rumor, as illustrated in listing 2.

```
newPkt := transport.Packet{
    Header: pkt.Header,
    Msg:    rumor.Msg,
}

err := node.registry.ProcessPacket(newPkt)
if err != nil {...}
```

Listing 2: Processing the embedded message of a rumor using the registry.

Task 3: Implement the anti-entropy mechanism

Our only way so far to make nodes' views consistent are the statuses embedded in acks. However, if nodes don't send messages for a while, some nodes might wait too long to harmonize their views. The anti-entropy ensures that peers in Peerster eventually get the same view on the system. To do so, each peer must send a StatusMessage at a regular interval to a random neighbor. The interval is given in the configuration. If an interval of 0 is given then the anti-entropy mechanism must not be activated.

The StatusMessage must contain the last Rumor's sequence the peer received from each peer it received a rumor from, so far. It must also include an entry for the peer itself, telling the last Rumor's sequence the peer sent. Given you are the peer `127.0.0.1:1`, here is an example of a possible StatusMessage:

```
{
      "127.0.0.1:1": 2,
      "127.0.0.1:2": 1,
      "127.0.0.1:4": 7,
}
```

This StatusMessage is telling that your peer sent 2 Rumors, received 1 Rumor from `127.0.0.1:2`, and 7 from `127.0.0.1:4`. Note that there might be a peer `127.0.0.3` in the system. If that's the case then the status implicitly says that your peer didn't receive any rumors from it. Note that, because nodes process rumors in sequence per origin, the message above implicitly says that the peer received the rumor number. 1 from `127.0.0.1:2` and rumors 1 to 7 from `127.0.0.1:4`.

Task 4: Process StatusMessage

Status messages are used to sync Rumors between peers. Those messages are embedded in AckMessage and are also sent periodically by the anti-entropy mechanism.

When a peer P receives a status message from a remote peer it must compare the StatusMessage, which represents the remote peer's view, to its own view. There are four possible cases:

1. The remote peer has Rumors that the peer P doesn't have.
2. The peer P has Rumors that the remote peer doesn't have.
3. Both peers have new messages.
4. Both peers have the same view.

Here are the corresponding actions for each possible case:

1: The peer P must send a status message to the remote peer.

2: The peer P must send all the missing Rumors, in order of increasing sequence number and in a single `RumorsMessage`, to the remote peer. In this case peer P is not expecting an ack message in return. However, the remote peer could send back one, as the remote peer does not need to differentiate between "catch up" and "broadcast" RumorsMessages.

3: The peer P does actions 1 and 2.

4: With a certain probability, peer P sends a status message to a random neighbor, different from the one it received the status from. This mechanism is what we call "ContinueMongering". The probability is given in the configuration.

Task 5: Process AckMessage

When a RumorMessage is sent, the sending node expects an AckMessage to be received within a certain timeout. This ack message must contain the same PacketID that was used in the packet's header to send the RumorMessage.

When a peer receives an AckMessage, the peer should do two operations:

- The peer stops waiting (stops the timer) for the ack corresponding to that PacketID.
- Process the status message contained in the `AckMessage` (use the `registry.ProcessPacket` function).

If the peer does not receive an AckMessage after the timeout, then the peer must send the RumorMessage to **another** random neighbor that is different from the node it previously sent the rumor to.

If not already done, make sure that when your peer creates a rumor in the broadcast function, it also waits for the corresponding ack message. Waiting on an ack should not be a blocking operation. The timeout for an Ack is given in the configuration when a peer is created.

Task 6: Implement the routing update

Every peer in the system should have an entry in its routing table to every other peer. We are going to use the rumors to our advantage to update the routing table. The principle is rather simple: update the routing entry of a peer each time we process a new rumor from that peer and we are not already directly connected to that peer (i.e. this peer is not our neighbor). The routing's entry for a peer should be the relay address specified by the packet's header containing the new rumor. The sequence should be the rumor's sequence. Therefore, you should check for a routing update each time you process a rumor from a RumorsMessage.

If peer A receives an expected rumor with sequence N originating from peer C, and the rumor is relayed by peer B, then the routing table for peer A would be updated with the following entry:

```
{
	“Peer C socket address”: {“Peer B socket address”}
}
```

Task 7: Implement the heartbeat mechanism

The heartbeat mechanism makes peers announce themselves to every other peer and has the goal of keeping routing tables up-to-date. Let's use **Rumors** for that: When a peer

sends a rumor, it’s already the case that every other peer will eventually process that rumor and add an entry about that peer in their routing table. This is exactly what we need.
Because Rumors need to carry a message, but heartbeats don’t have any meaningful message to transmit, we simply embed an `EmptyMessage` in such “heartbeat” rumors. Processing the `EmptyMessage` should have no additional effect.

Your peer should send a rumor with an empty message at bootstrap and continuously at a regular interval. This interval is given in the config. If the interval is 0 then the heartbeat mechanism must not be activated, i.e. no rumors with `EmptyMessage` are sent. Note that it can be quite expensive to send a rumor, so the heartbeat interval should not be set to a high rate.

In short, the heartbeat mechanism consists of calling the broadcast function with an EmptyMessage at starting time and a regular interval.

Task 8: Implement the private message mechanism

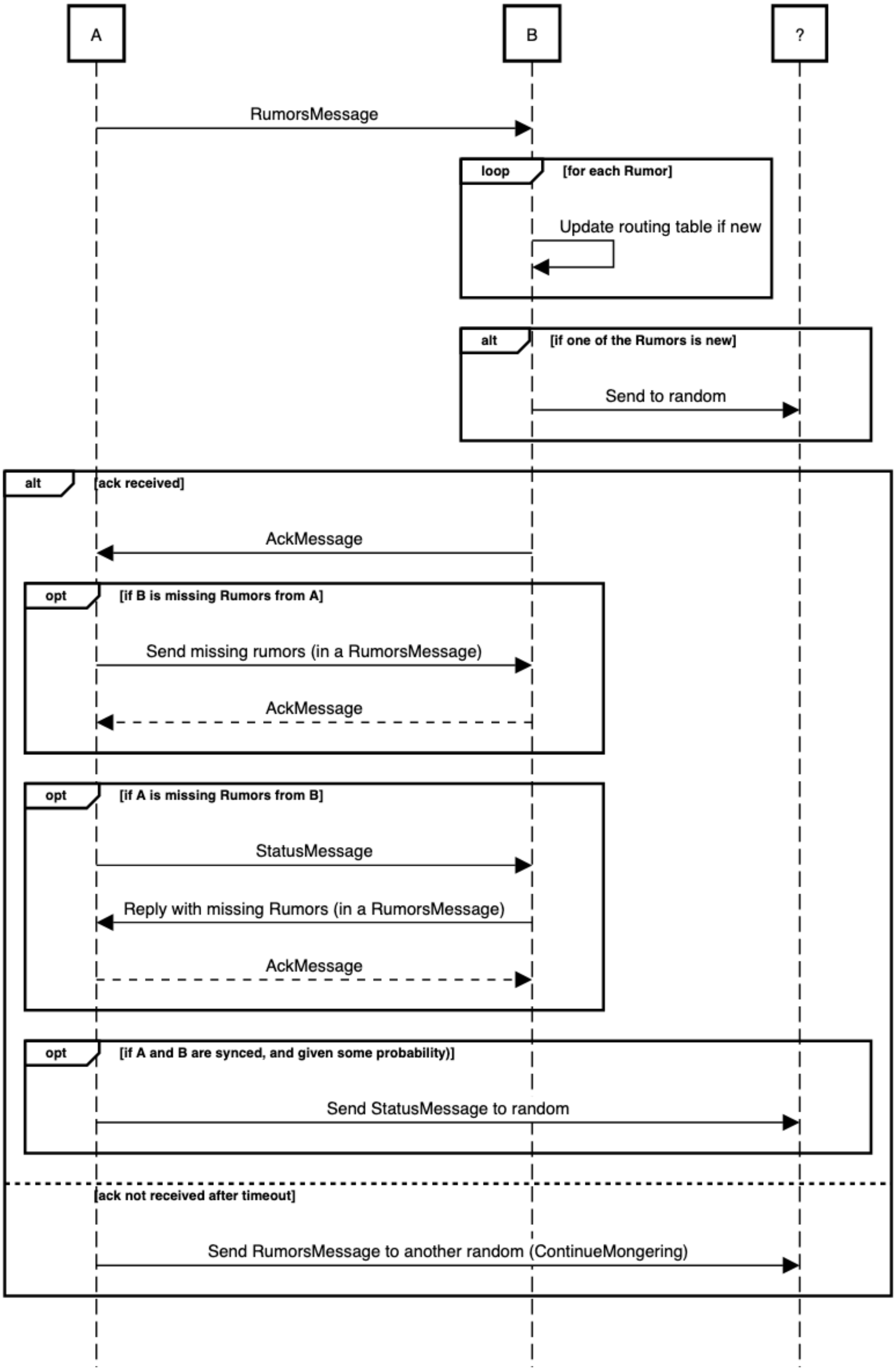
Implement the handler for the `PrivateMessage` message type. This handler is rather simple:

- Check if the peer’s socket address is in the list of recipients.
- If the previous condition is true, process the embedded message using the message registry.

Hint: Sequence diagram

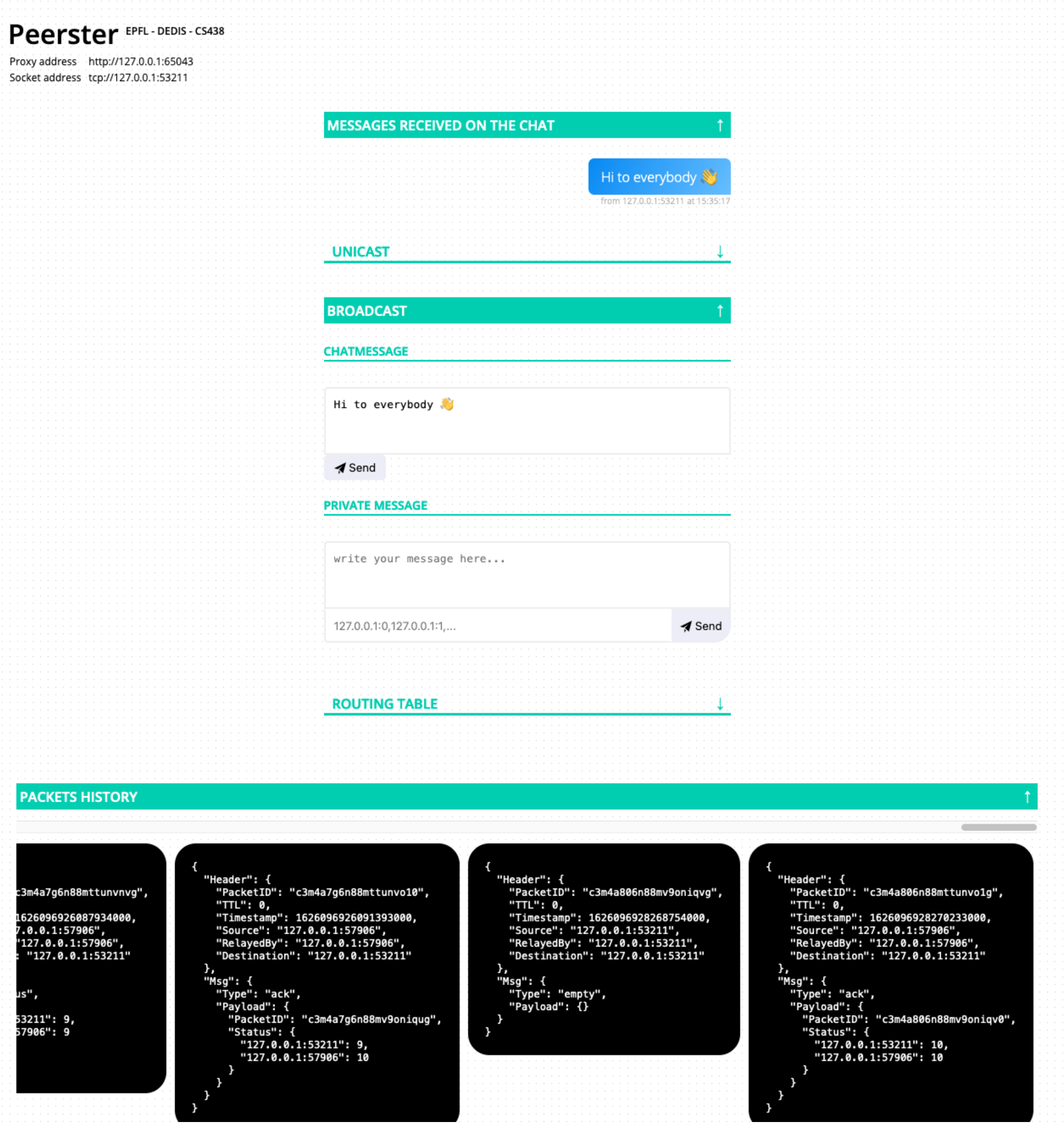
The following sequence diagram shows what happens when a peer sends a RumorsMessage to another peer.

Rumor sending with ack, status, and ContinueMongering


A
B
?
RumorsMessage
loop
[for each Rumor]
Update routing table if new
alt
[if one of the Rumors is new]
Send to random
alt
[ack received]
AckMessage
opt
[if B is missing Rumors from A]
Send missing rumors (in a RumorsMessage)
AckMessage
opt
[if A is missing Rumors from B]
StatusMessage
Reply with missing Rumors (in a RumorsMessage)
AckMessage
opt
[if A and B are synced, and given some probability)]
Send StatusMessage to random
[ack not received after timeout]
Send RumorsMessage to another random (ContinueMongering)

Try your program

As in HW1 and apart from the unit+integration tests, you can use the web GUI to try your program. Launch nodes with the CLI, connect peers together and start sending broadcast messages. Figure 2 illustrates what you should see.



Peerster EPFL - DEDIS - CS438

Proxy address http://127.0.0.1:65043
Socket address tcp://127.0.0.1:53211

MESSAGES RECEIVED ON THE CHAT ↑

Hi to everybody 👋

from 127.0.0.1:53211 at 15:35:17

UNICAST ↓

BROADCAST ↑

CHATMESSAGE

Hi to everybody 👋

Send

PRIVATE MESSAGE

write your message here...

127.0.0.1:0,127.0.0.1:1,...

Send

ROUTING TABLE ↓

PACKETS HISTORY ↑

```
c3m4a7g6n88mttunvnvg",

1626096926087934000,
7.0.0.1:57906",
"127.0.0.1:57906",
: "127.0.0.1:53211"


us",

53211": 9,
57906": 9
```

```
{
  "Header": {
    "PacketID": "c3m4a7g6n88mttunvo10",
    "TTL": 0,
    "Timestamp": 1626096926091393000,
    "Source": "127.0.0.1:57906",
    "RelayedBy": "127.0.0.1:57906",
    "Destination": "127.0.0.1:53211"
  },
  "Msg": {
    "Type": "ack",
    "Payload": {
      "PacketID": "c3m4a7g6n88mv9oniqug",
      "Status": {
        "127.0.0.1:53211": 9,
        "127.0.0.1:57906": 10
      }
    }
  }
}
```

```
{
  "Header": {
    "PacketID": "c3m4a806n88mv9oniqvg",
    "TTL": 0,
    "Timestamp": 1626096928268754000,
    "Source": "127.0.0.1:53211",
    "RelayedBy": "127.0.0.1:53211",
    "Destination": "127.0.0.1:53211"
  },
  "Msg": {
    "Type": "empty",
    "Payload": {}
  }
}
```

```
{
  "Header": {
    "PacketID": "c3m4a806n88mttunvo1g",
    "TTL": 0,
    "Timestamp": 1626096928270233000,
    "Source": "127.0.0.1:57906",
    "RelayedBy": "127.0.0.1:57906",
    "Destination": "127.0.0.1:53211"
  },
  "Msg": {
    "Type": "ack",
    "Payload": {
      "PacketID": "c3m4a806n88mv9oniqv0",
      "Status": {
        "127.0.0.1:53211": 10,
        "127.0.0.1:57906": 10
      }
    }
  }
}
```

Figure 2: Web gui

Appendix A - Tests definitions

Note to authors: Do not directly modify the test definitions without updating the implementation. Leave a comment if needed.

Broadcast

Pre-condition	**Action**	**Expectation**
(1-1) The following topology: A -> B.	Peer A broadcasts a message M.	A sent a rumor. B received a rumor with the correct header. B sent an ack. A received an ack. A and B processed M. A's routing table contains an entry for A and B. B's routing table contains an entry for A and B.
(1-2) The following topology: A -> B -> C. ContinueMongering set to 0.	Peer A broadcasts a message M.	A, B, and C processed M. C received a rumor from B. C sent an ack to B. A's routing table contains an entry for A and B. B's routing table contains an entry for A, B, and C. C's routing table contains an entry for A and C. RelayAddr for A is B.

Anti-entropy

Pre-condition	Action	Expectation
(1-3) The following topology: A -> B. (A knows B, but B doesn’t know A) A’s anti-entropy is set to 500ms. B has the default anti-entropy value of 0.	Wait 800ms.	A didn’t receive any packet. A sent at least 1 status message to B. B received at least 1 status message from A. B didn't send any packet.

Heartbeat

Pre-condition	Action	Expectation
(1-4) The following topology: A -> B. A’s heartbeat is set to 500ms. B has a default heartbeat value of 0.	Wait 800ms.	A sent at least 1 rumor to B. B received at least 1 rumor from A. B sent at least 1 ack to A. A received at least 1 ack from B.

ContinueMongering

Set to 1

Pre-condition	Action	Expectation
(1-5) The following topology: A -> B	Peer A broadcast a message M.	A received, in order: ack, status, ack. A sent, in order: rumor, status, rumor, status.

-> C ContinueMongering set to 1 for all peers.		One of B or C: Received, in order: rumor, status. Sent an ack. The other one of B or C: Received, in order: status, rumor. Sent, in order: status, ack. A, B, and C processed M.

Set to 0

Pre-condition	Action	Expectation
(1-6) The following topology: A -> B A -> C ContinueMongering set to 0 for all peers.	Peer A broadcast a message M.	A received an ack. A sent a rumor. One of B or C: Received a rumor Sent an ack. The other one of B or C: Received no messages. Sent no messages.

Catchup

Pre-condition	Action	Expectation
(1-7) The following topology: A -> B -> C. B is not started. All nodes have an anti-entropy of 50ms.	Peer A broadcasts message M1. Peer C broadcasts message M2. Peer B is started. Wait 200ms.	All peers received the same 2 rumors. All peers processed M1 and M2 with the respective messages.

Ack

With Ack

Pre-condition	Action	Expectation
(1-8) The following topology: A -> B A -> C A's ackTimeout set to 500. B and C are not sending back any messages (simulates network loss).	Peer A broadcasts a message M. Wait 800ms.	A sent two messages: one for B, one for C. A processed M. A received no messages. B and C received the rumor from A.

Without Ack

Pre-condition	Action	Expectation
(1-9) The following topology: A -> B -> C A's ackTimeout set to 0 (wait forever). B and C are not sending back any messages (simulates network loss).	Peer A broadcasts a message M. Wait 100ms.	A sent one message: one for B OR C. A processed M. A received no messages. B OR C received the rumor from A (same peer as in the first condition)

BigGraph broadcast

Pre-condition	Action	Expectation

(1-10) A random topology with 20 peers.	Every peer broadcasts a chat message.	Every peer processed the same 20 chat messages. Every peer has 20 entries in their routing table. An entry for each other peer.

Private message

Broadcast

Pre-condition	**Action**	**Expectation**
(1-11) The following topology: A -> B -> C -> D Anti-entropy is set to 50ms on all peers.	Peer A broadcasts a private message for B and D and with an embedded message M.	B and D have processed M. A and C have not processed M.

Unicast

Note: Sending a unicast private message is meaningless, but the system should allow it if it is implemented correctly. This is a sanity check.

Pre-condition	**Action**	**Expectation**
(1-12) The following topology: A -> B A -> C	Peer A unicasts a private message for C with an embedded message M to B Peer A unicasts a private message for B with an embedded message M to C	C has processed M. A and B have not processed M. C should have processed the embedded message. P{ H:{} M:{ M(e)}}

		M(e)

Integration test

Pre-condition	**Action**	**Expectation**
(1-13) A random topology with 10 reference peers and 10 students peers (10 instances of the peer implemented by 1 student).	Every peer broadcasts a chat message.	Every peer processed the same 20 chat messages. Every peer has 20 entries in their routing table. An entry for each other peer.