Homework 1
Gossip, Routing and Private Messaging

CS-438 - Decentralized Systems Engineering, Fall 2020

Publish date: Friday, October 02, 2020

Due date: Tuesday, October 27, 2020 @ 23:55 (Updated)

General Guidelines

In this homework, you will continue to build more functionalities to *Peerster* **on top of your Homework 0 implementation**. In this homework, you will add the functionalities to gossip messages, to use routing tables for message forwarding, and to send private messages.

We will provide you with the skeleton files that contain the main interfaces to be implemented, test framework, and some useful functionality and implementation hints. The skeleton files are implemented on top of the skeleton for Homework 0 which means that you should be able to merge it with your existing implementation. Specifically, you will receive access to a new repository cs438-hw1-student-*name* which you can add as a new git remote in the folder with your existing implementation and pull the updates into your master branch, e.g., by running

```
git remote add hw1
https://gitlab.epfl.ch/cs438/students/cs438-hw1-student-XYZ.git

git pull hw1 master
```

Then you should be able to merge the master branch into the branch with your implementation. We suggest that you use an automatic tool, such as KDiff3, for merging (it can automatically resolve the conflicts of modifying the same file as long as the changes do not touch the same source lines), although you will still likely have to manually resolve some of the conflicts, for which you need use your own reasoning.

Throughout this development and debugging process, you are welcome to discuss challenges and techniques with your fellow students, exchange pointers to relevant information or algorithms, debugging tips, etc., ***provided you each write your own code independently*****. Homeworks are individual per student.**

Teaching assistants will be available in room INJ 218 and on Zoom every Friday 15:15-17:00, to discuss with you how to architect your implementation. ***TAs are not going to debug your***

code, but they can help you ask the right questions just like your software engineer colleagues will do in the future. Room INF 1 is available every Monday from 13:15 to 15:00 for you to hack together and test your implementations, without TA supervision.

Second Life, a.k.a. Code Review

Every homework builds on the previous one and requires a full and working implementation of the previous homework. Thus, we strongly encourage you to fully implement every homework. However, if it so happens that: (1) You were unable to complete a homework assignment, or (2) You fully implemented your homework but the poor code design makes it hard for you to build on top, or (3) You are unsure why you don't pass all tests, we offer you an alternative.

After the deadline of Homework 0, you will receive 3 random submissions of your colleagues (by getting a read-only access to their HW0 repositories on Gitlab). You can choose to build on top of any of these 3 assignments for your next homework. **If you decide to do so, you need to specify that homework's identifier, i.e., the Gitlab repo link, in your future homework submission on Moodle.** Also, if you build on someone else's code, we *strongly suggest* that you review that colleague's code on Gitlab as a token of appreciation -- the review is, however, not graded.

However, be warned that there is no guarantee regarding the quality of these 3 randomly assigned submissions - they might be less good than your own implementation. Your best strategy, thus, is still to complete the homeworks yourself.

Introduction

Gossip protocols are distributed protocols for robust information exchange, typically deployed on dynamic network topologies, e.g, because nodes can join and leave the network (also called churn), they are mobile, their connectivity varies, etc. Examples of applications are ad-hoc communication between self-driving cars, peer-to-peer networks that broadcast a TV program, sensor nodes that detect fire hazard in remote areas.

Part 1 of this homework fixes the imperfect message broadcast protocol of Homework 0. In Homework 0, you implemented the simplest approach to deliver messages in a decentralized network – broadcasting them. Albeit simple, this approach was far from perfect: messages could be lost, duplicated, or could loop around the network indefinitely. Efficient gossip protocols spread information more like gossipping in real life: a rumor may be heard by many people, although they don't hear it directly from the rumor initiator. The core idea of the protocols is that nodes maintain the state of messages in the system, randomly share new messages and periodically check that they are up-to-date with the latest "news". In the first part of this homework, you will extend your Homework 0 broadcast implementation to support gossiping.

Part 2 of this homework adds support for unicast messaging. Although gossiping is a more efficient way to spread information, it is still a broadcast and it requires all the nodes in the network to be aware of every message. In reality, nodes in the network might need to exchange unicast point-to-point messages, rather than having to broadcast everything through gossip. The goal of the second part of this homework is to extend your gossip implementation with the support for point-to-point message exchange (private messaging). To achieve this, you will create a routing protocol on top of your gossip protocol and use that to implement point to point messaging.

We provide you with the GUI of this homework, which adds support for displaying rumors and private messages.

Part 1: A gossip protocol

There are two key problems with the trivial protocol of homework 0:

1. Although UDP is a standard communication protocol in decentralized networks, it is unreliable and offers only best-effort delivery, which means messages may get lost or duplicated in the network for a wide variety of reasons. This might be unlikely to happen when you're sending a message from one UDP socket to another on the same host, but becomes much more likely once you start sending messages between hosts: UDP datagram loss or duplication in the network may cause messages to be lost, or to appear multiple times.

2. Although in the Internet's original architecture the Network Layer (IP) underlying UDP was intended to guarantee universal “any-to-any” connectivity, so that any Internet host could communicate directly with any other, the modern Internet architecture is not a full mesh, i.e., hosts have to be able to communicate indirectly via each other. When the main objective is merely to ensure that a number of cooperating hosts or processes each obtain copies of whatever messages any of them send, as when implementing a chat room, one of the simplest yet also fastest and most reliable known algorithms for propagating those messages is known as a gossip protocol. USENET, the Internet's original widespread and decentralized public “chat room”, used such an algorithm. A few references for familiarizing yourself with gossip protocols (see the corresponding course lecture for an extensive overview) are available at the end of this document.

Gossip in Peerster

A good gossip protocol must provide efficient communication in terms of both bandwidth and information propagation, and must be robust to node and network failures. Moreover, the necessary message forwarding (as the network is in most cases not fully connectected) must not lead to infinite loops that you have probably spotted in Homework 0. The final desired property for the protocol is simplicity: constructing from simple but robust building blocks reduces the possibility of errors and simplifies design.

The two building blocks that will enable you to address the aforementioned challenges are the *rumormongering* and *anti-entropy* mechanisms. You can learn the details of how these mechanisms help in the course lecture. The gist is that rumor mongering facilitates the timely propagation of “hot” (in other words, fresh) messages by requiring nodes to distribute each new message to randomly picked neighbors until this message is heard by most. Anti-entropy ensures that all the nodes eventually receive all the distributed messages by requiring nodes to periodically compare their lists of “heard” messages. This is essentially a catch-up mechanism.

The turning wheels of the two mechanisms are **rumor messages** and **status messages**:

- **Rumor message:** Contains the actual text of a user message to be gossiped. These messages must be uniquely identifiable such that the nodes can keep track of them. Peerster addresses this by including a *sequence number* and the identity of the originating node in each rumor message. The sequence number distinguishes successive messages from a given client and enables peers to “compare notes” on which messages from which other peers they have or have not received. For example, if peer A has received messages originating from peer C up to sequence number 5, and compares notes with peer B that has received C's messages only up to sequence number 3, then A knows that it should propagate C's messages 4 and 5 to B. This convention essentially amounts to implementing a *vector clock* (see the additional readings at the end). The sequence numbers in Peerster start from 1.

- **Status message:** Summarizes the set of messages that the sending peer has seen so far from other peers and the messages that the peer itself has sent. A node sends a status message in two cases: (1) as an acknowledgement to a received rumor; and (2) when a periodic timer fires, as a part of the anti-entropy mechanism.

  The status message contains only one field: essentially a vector clock with other nodes’ IDs (origin IDs) that the peer knows about, and their associated values representing the lowest sequence number for which the peer has *not yet* seen a message from the corresponding origin node. For example, if A sends a status message to B containing the pair <C,4>, this means that A has seen all messages originating from C having sequence numbers 1 through 3, but has not yet seen a message originating from C having sequence number 4. *However, A may or may not have seen messages from C with sequence numbers higher than 4 --- <C,4> in itself says nothing about messages from C with IDs > 4.*

Rumormongering

We now describe the rumormongering protocol that you need to implement:

- Whenever a peer S (sending peer) obtains a new message **that it did *not* have before** – either a `Message` from the local client (in which case this peer becomes the message's origin), or a rumor message from another peer in a new rumor message – the peer picks a **random receiver peer R (from the list of all known peers, which includes peers given at bootstrap, as well as peers this node received messages from)** and sends a copy of the rumor to that target. The gossiper also sends a callback to the client informing the reception of a new message, with the origin name and the message as arguments.
- The receiver peer R **acknowledges the message by sending a status message back** to peer S.
- If the sending peer S **receives a status message (from R) acknowledging the transmission**, it compares the vector in the status message with its own status. There are three cases:
    1) S has *other* new messages that R has not yet seen, and, if so, S sends as rumors **one by one** all the missing messages **to the same receiving peer R**. Note that this message causes R to start rumormongering;
    2) S does not have anything new but sees from the exchanged status that R has new messages. Then, S itself sends a status message containing S's status vector, which causes R to send the missing messages back **(one at a time);**
    3) If neither peer has new messages, the sending peer S flips a coin (e.g., `rand.Int() % 2`), and either (heads) picks **a new random peer to send the same rumor message to**, or (tails) **ceases** the rumormongering process.
- If, after some time, e.g., a timeout of 10 seconds, the sending peer has not received a status message from the receiver peer, then it simply picks another peer to rumormonger its message. To implement this, you will need to set a timer to have a handler re-invoked after some period if no message has been received by then. The `time.NewTicker(some_value)` function is your friend.

To keep things simple, you should always send new rumor messages from a given origin in sequence number order. That is, if A is rumormongering with B and has new messages (C,3) and (C,4), then A should propagate (C,3) to B before propagating (C,4). This simplifies the program's logic by alleviating the need to buffer out-of-order messages: a peer can simply drop a message if it has not seen this message's predecessor. Sending a status message back will trigger the catch-up process and the dropped message eventually will come back in order.

An important question is how to make sure that a peer's name (identifier) is unique. Ultimately how you come up with an origin identifier is up to you, as long as you have a good reason to believe it will be unique (e.g., derive it from your SCIPER). A reasonable approach would be to pick a random number when the program starts and include that in the peer's name. Even better might be to use a long, cryptographically strong random number or cryptographic hash, but we will get to do that in later labs. **For now, let's keep the name given in the command line as the peer's name, as it's easy to use different names when all peers run on localhost.**

When a gossiper receives a message from a client, it needs to decide whether to broadcast it naively as per homework 0 or rumormonger it as per homework 1. To differentiate between the two, your gossiper program receives a "broadcast" flag, set to "true" for homework 0 and "false" for homework 1.

Anti-entropy

Rumormongering by itself is not guaranteed to ensure that all participating nodes receive all messages: the process might stop too soon. To ensure that all nodes eventually receive all messages, you will need to add an anti-entropy component. In Peerster, we will take a simple approach: just create a timer that fires periodically (there will be a new -antiEntropy flag in your gossiper; there's a default antiEntropy duration of 10 seconds if the flag is absent when the program is run) and that causes the peer to send a status message **to a randomly chosen peer from the list of known peers, which includes peers given at bootstrap, as well as peers this node received messages from**. Status messages are processed the same way as described above: the node does not need to distinguish what mechanism (rumormongering or anti-entropy) triggered the sending of a status message.

Message format

Just as in Homework 0, the client sends a type `ClientMessage` to the gossiper in Part 1 of Homework 1.

When a peer receives a message, it needs to be able to tell whether the message is a rumor or a status (or `SimpleMessage` from Homework). `RumorMessage` and `StatusPacket` are now options for encapsulation into a `GossipPacket`. You will find the new `RumorMessage`, `StatusPacket`, and updated `GossipPacket` structures in the skeleton code that we provide.

When decoding a `GossipPacket`, you could check which of the `Rumor`, `Status` or `Simple` fields is not `nil`, and know in that way what message you have received and process it accordingly. In a `GossipPacket`, one and only one field should be non-nil.

The message formats are below:

- **Rumor** message
  JSON format: `{origin: <value>, ID: <value>, Text: <value>}`

- **Status** message
  JSON format: `{Want:[PeerStatus]}`

  - **PeerStatus**

JSON format: `{Identifier: <value>, NextID: <value>}`

Printing format in the standard output and sending to the Watcher

Standard output

Your `gossiper` program should write at standard output:

- **`CLIENT MESSAGE <text_content>`** when receiving a message from a client
- **`SIMPLE MESSAGE origin <original_sender_name> from <relay_addr> contents <msg_text>`** when receiving a `SimpleMessage` from another peer
- **`RUMOR origin <original_sender_name> from <relay_addr> ID <msg_id> contents <msg_text>`** when receiving a `RumorMessage` from another peer
- **`MONGERING with <ip:port>`** when sending a `RumorMessage` to a peer at `ip:port`
- **`STATUS from <relay_addr> peer <name1> nextID <next_ID1> peer <name2> nextID <next_ID2> etc`** when receiving a `StatusPacket` from another peer with address `ip:port`
- **`FLIPPED COIN sending rumor to`** when rumormongering continues after coin flip
- **`IN SYNC WITH <ip:port>`** when the peer receives a status message and is up-to-date with all messages it contains
- **`PEERS <ip1:port1>,<ip2:port2>,etc` list of addresses of all known peers, printed every time the node receives a message**

Example of a running `gossiper` program:

```
RUMOR origin E from 127.0.0.1:5000 ID 1 contents Weather_is_clear
PEERS 127.0.0.1:5002,127.0.0.1:5000
MONGERING with 127.0.0.1:5002
STATUS from 127.0.0.1:5002 peer E nextID 1
PEERS 127.0.0.1:5002,127.0.0.1:5000
STATUS from 127.0.0.1:5002 peer E nextID 2
PEERS 127.0.0.1:5002,127.0.0.1:5000
CLIENT MESSAGE Winter_is_coming
STATUS from 127.0.0.1:5000 peer E nextID 2 peer B nextID 1
PEERS 127.0.0.1:5002,127.0.0.1:5000
STATUS from 127.0.0.1:5002 peer E nextID 2 peer B nextID 2
PEERS 127.0.0.1:5002,127.0.0.1:5000
STATUS from 127.0.0.1:5000 peer E nextID 2 peer B nextID 2
PEERS 127.0.0.1:5002,127.0.0.1:5000
IN SYNC WITH 127.0.0.1:5002
```

Watchers

In addition, you will find in your skeleton files that now your gossiper implements two watchers, for incoming gossip packets `inWatcher` and outgoing gossip packets `outWatcher` (we need those for testing). You need to first initialize them when you create your gossiper. Then, you need to send all the incoming packets to `inWatcher` and all the outgoing packets from your gossiper to `outWatcher` via the `Notify` method of the watcher. Specifically, you need to wrap the gossip packet in the `CallbackPacket` struct that you will find in `gossip/packets.go`. You do not need to modify the code of the watcher, simply send in the incoming and outgoing gossip packets.

Part 2: Routing and Private Messaging

In order to communicate with another node, any Peerster node has so far had to broadcast its messages into the network and to expect that these messages will eventually propagate to the destination. In addition to slower-than-desired message delivery, communicating via rumors requires nodes to store all the private messages in the network even if a node is not part of a conversation. In Part 2, you will address this issue by giving Peerster nodes point-to-point communication capability.

The difference between message spreading and point-to-point delivery is that in the latter case, nodes need to know *where* to send their messages. Recall that in most cases networks are not a full mesh, hence nodes need to be able to send messages *via* each other to communicate with their non-neighbors. The tool to enable such communication is a routing protocol that will define to which neighbor a node needs to forward each private message by computing routing tables and determining a unique path between pairs of nodes.

In Peerster, you will first implement a simple *destination-sequenced distance vector* (DSDV) routing scheme. This scheme has proven popular for routing in ad hoc mobile networks, due to its combination of simplicity and robustness against routing loops. You should first familiarize yourself with the general scheme via appropriate background readings (see some references at the end of the documents).

In this scheme, each node maintains a table of destinations and, for each destination, a “next hop” to reach that destination. **Peerster will piggyback its routing scheme on its gossip protocol**: each rumor message will double as a route announcement message, and the sequence numbering scheme you have already implemented for gossip purposes will act as the “destination sequence numbers” that the DSDV routing scheme requires.

Maintaining a routing table

Change your Peerster implementation to build and maintain a **next-hop routing table**: a key/value dictionary (e.g., a map) where keys are `Origin` identifiers and values are `(IP address, port number)` pairs (e.g., `map[string]string`). Whenever a Peerster node receives **the first rumor message from `Origin`**, record it in your next-hop table, at the key corresponding to the message's `Origin`, the `IP address` and `port number` from which that rumor message arrived. The tuple (`IP address`, `port number)` will be the next-hop on your route to `Origin`, which will remain in effect until you receive the next rumor message (**with a higher sequence number**) from the same node `Origin`. Of course, you still need to rumormonger the message to a random neighbor whenever you receive a Rumor message.

For example, at the beginning, Dave sends a Rumor message to Bob. We assume that Dave's IP is `"1.2.3.4"`, his port is `"43433"` and `ID` is `1`. Thus, Bob updates his DSDV routing table with item `[“Dave”:"1.2.3.4:43433”]`. Then, Bob forwards that Rumor message to Alice. After receiving the message, Alice also updates her DSDV routing table with item `[“Dave”:"5.6.7.8:33333”]`, where `"5.6.7.8"` is Bob's IP address and `"33333"` is Bob’s port. If Alice wants to send a message to Dave, she just needs to send a message to `"5.6.7.8"` (with port `33333`), although she does not know the concrete IP address of Dave. After several hours, if Eve sends another Rumor message to Alice through Bob, both Alice and Bob will add new items in their routing tables. Specifically, a new item `[“Eve”:"5.6.7.8:33333”]` will be added to Alice's routing table. (Of course, Bob will also update his routing table.) Please note that if Dave sends a new Rumor message with `ID 3` to Alice through Jack, Alice will have a new item `[“Dave”:"3.3.3.3:55555”]` instead of the old one (i.e., `[“Dave”:"5.6.7.8:33333”]`). We assume `"3.3.3.3"` is Jack's IP and his port is `"55555"`.

When updating a DSDV routing table entry (e.g. for peer "Dave" updated to `[“Dave”:"3.3.3.3:55555”]`), your program should print out the following to stdout:

```
DSDV <peer_name> <ip:port>
```

where <ip:port> is the new next-hop for <peer_name>. For instance :

```
DSV Dave 3.3.3.3:55555
```

Route Rumors

Unfortunately, if Peerster nodes only ever “announce” themselves to the network when the local user actually types in a message, nodes whose users are inactive for extended periods will never be announced in the network, and thus other nodes will not be able to find a route to them. We will fix this by ensuring that nodes always send Rumor messages occasionally, merely for the purpose of updating other nodes' routing tables, even when the local user is idle.

Add a periodic timer that generates a *route rumor* message. A route rumor message is just like the simple rumor messages you already generate, except it contains only the identity of the message's original sender and the message ID, whereas *its content is empty*. Modify your Peerster’s implementation message handler to accept and forward route rumors as well as regular rumors. *The processing should be essentially the same, except that you DO NOT display a DSDV message to the user when the content of the rumor message is empty.*

For testing purposes, your `gossiper` will get a new **`rtimer`** flag that indicates how many seconds the peer waits between two route rumor messages. The default value will be 0 seconds, which means disabled route rumors. **You need to add the implementation logic that disables sending all route rumors (including the one sent at startup, see below) if the value of `rtimer` is 0.**

Start-up Route Rumor

Besides the periodic route rumors, specified through the **`rtimer`** flag, a Peerster node should also **generate a single route rumor message when it first starts up** (either sent to a random neighbor or broadcast to all neighbors; your choice) to "prime the pump" and get itself known to other nodes quickly. Test your code to ensure that a routing table entry appears "quickly" in other Peerster instances after startup without having to manually type any chat messages.

Important: Route rumors and start-up route rumors **DO NOT** output DSDV messages.

Private Messages[1]

You will implement private messages as another option for `GossipPacket`, specifically as the `Private` field (see the homework skeleton). The private message is like a rumor message but it also includes the message destination and a hop limit, in addition to the identifier of the origin node, the message ID (set to 0, see below) and the message text.

The message destination is the identifier of the destination node (corresponding to the origin of a prior rumor message). The hop limit defines how far your private message should be able to reach. You should initialize with a constant default value 10. Every node on the forwarding path (**including the source peer**) will first decrement this value and either forward the message, or discard the message if the value reaches 0 before the message reaches the destination. Although routing loops should *normally* not arise in a DSDV protocol such as this, bugs or malicious behavior could still create loops, and the hop limit protects against this risk. Once the destination node receives the private message, a callback is sent to the controller and the cli and GUI will display the message.

For the sake of simplicity, we **do not** require you to implement correct sequencing of private messages between pairs of hosts - *which means there's no need to use vector clocks*. You can of course implement it if you want to, but keep in mind that our DEDIS peer will not implement it. For compatibility, we set the message ID field to 0, as mentioned above, to signal "*no order imposed*" for private messages.

[1] Note that, here, the use of the term "private" bears no connection to privacy. In Peerster, "private messages" are merely unicast messages.

Updating the client

There will be a new **`-dest`** CLI option that defaults to the empty string. If **`-dest`** is filled along with a **`-msg`** option, your program should send a private message to the destination **(and no rumor)**. Thus, you need to implement handling the following options in your CLI:

```
- UIPort string
        port for the UI client (default “8080”)
- dest string
        destination for the private message; can be omitted
- msg string
        message to be sent; if the -dest flag is present, this is
        a private message, otherwise it’s a rumor message
```

Printing format in the standard output and sending to the Watchers

When the gossiper receives a private message from a local client, it should write at stdout:

```
CLIENT MESSAGE <msg_text> dest <dst_name>
```

Every node on the routing path, **including the `Origin`**, must process the packet and:

- If the message destination refers to another node, first check the hop limit. If the hop limit is 0, then the node DOES NOT send the message out and the processing stops here. Otherwise, if the hop limit > 0, decrement its value just before forwarding the packet to the next hop.
- **If the `Destination` field indicates that the message is for the local node**, you need to write at stdout:

```
PRIVATE origin <origin> hop-limit <hop-limit> contents <contents>
```

As before, you should be sending every received and sent gossip packet to the watchers. The only difference is that now these gossip packets can include private messages.

Automatic testing through GitLab

We will use GitLab for testing. Each of you will be provided with a GitLab repository that you can access with your credentials for gitlab.epfl.ch. The CI (continuous integration) tool will automatically run tests every time you push your code in the repo. For your convenience in implementation, you can inspect the tests and change them to debug your program. Unit tests and integration tests will be available soon.

We will provide you with the basic code structure with interfaces, on top of which you can write your code. To facilitate smooth code merging and in order for your code to work with the testing infrastructure, ***we strongly advise you not to change the project structure***.

Hand-in Procedure and Grading

For each homework, you will receive a grade out of 6. To be considered for receiving full points (max. 6), you must upload and submit your fully-working code on Moodle (simply as a collection of source files) by the due date, which you can find at the beginning of each homework assignment document. **You can always update your submission on moodle until the deadline, so please start submitting early. Late submissions are not possible.**

We will grade your solutions via a combination of automatic testing, code inspection, and code plagiarism detection. We will run automated tests on your application and make sure it works as required, both when communicating with other instances of itself (unit tests) and when communicating with our own instances (integration tests). You will be provided with the test framework, so that you can test your code yourself before submitting. Unless otherwise specified, the tests assume a full implementation of the homework. If you implement only parts of a homework, we cannot guarantee there will be tests for that particular functionality that would give you points for it. For the actual grading, we will use the same tests but with *different* input data so only the implementations that correctly implement all the functionality will pass.

Our very first test is that your code must compile when ***go build*** is executed on your files. **No points will be given if your code does not compile.**

References

Gossip protocols

- The Wikipedia page offers a high-level summary, though probably not all the details you will need (perhaps depending on the mood of the current editors and the phase of the moon).
- The original Epidemic Algorithms research paper by Demers et al at Xerox PARC in the 1980s. Perhaps not the easiest read, but there's no more definitive source.
- RFC 1036, the standard describing the way USENET news messages were formatted and propagated gossip-style in USENET's heyday. Pay particular attention to section 5 at the end on propagation, and section 3.2 on the Ihave/Sendme protocol.
- Textbook: Tanenbaum 4.5.2 “Gossip-Based Data Dissemination”

Vector clocks

- Wikipedia
- Fidge, “Timestamps in Message-Passing Systems That Preserve Partial Ordering”

- Mattern, “Virtual Time and Global States of Distributed Systems”
- Textbook: Tanenbaum 6.2 “Logical Clocks”, especially 6.2.2 “Vector Clocks”
- Background: Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”

DSDV

- Wikipedia (very brief summary)
- Original DSDV paper by Perkins and Bhagwat
- Background: Tanenbaum “Computer Networks” 5.2 Routing Algorithms; Coulouris 3.3.5 Routing.

------------------------------------This completes the homework--