



**COMPSCI 453**  
**COMPUTER NETWORKS**

**Professor Parviz Kermani**

**Programming Assignment 2: A Reliable data transfer protocol**

**Acknowledgment:**

This assignment was designed by Professor Jim Kurose.

**Roadmap:**

The text of this assignment is rather long. I suggest that you start by reading Section 1 and then Section 5. Once you have a rough picture of what's ahead, then read the entire rest of the assignment carefully. **There is a flowchart on [the last section of the document](#) that may help you design and code your project efficiently.**



## Table of Content

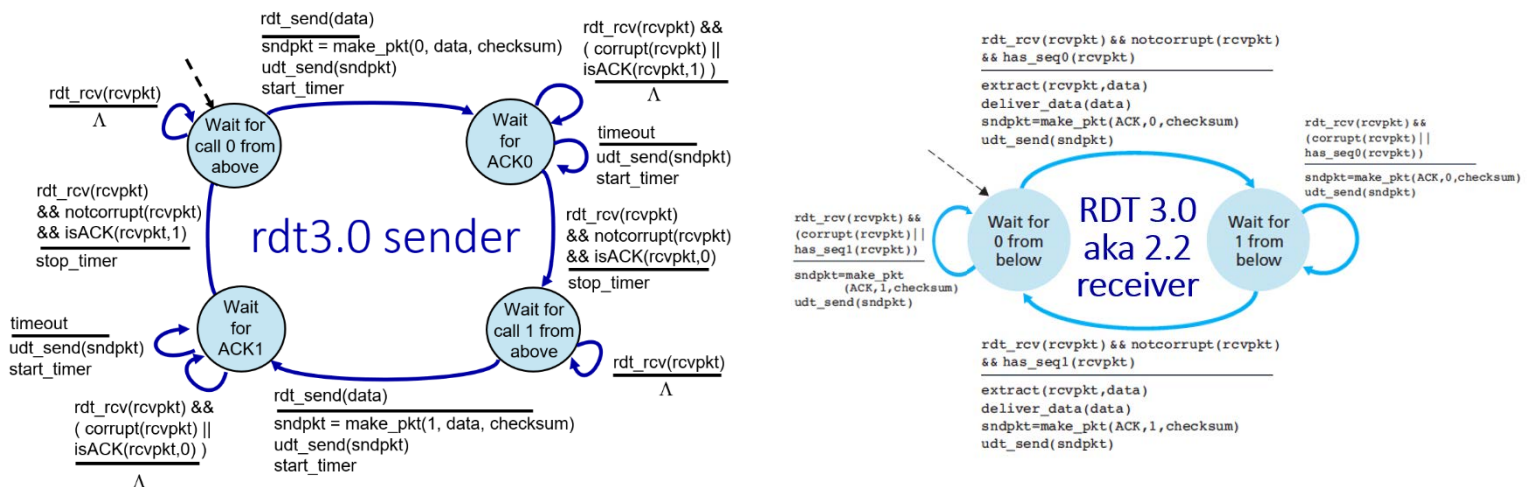
<b>PROGRAMMING ASSIGNMENT 2: A RELIABLE DATA TRANSFER PROTOCOL.....</b>	<b>1</b>
ACKNOWLEDGMENT: .....	1
ROADMAP: .....	1
<b>TABLE OF CONTENT .....</b>	<b>2</b>
<b>1. OVERVIEW AND INTRODUCTION .....</b>	<b>3</b>
<b>2. RDT 3.0 PROTOCOL DETAILS.....</b>	<b>3</b>
<b>3. THE NETWORK ENVIRONMENT .....</b>	<b>5</b>
<b>4. THE ASSIGNMENT .....</b>	<b>7</b>
<b>5. GETTING STARTED .....</b>	<b>8</b>
<b>6. CONNECTING YOUR SENDER AND RECEIVER TO A RECEIVER AND A SENDER WRITTEN BY SOMEONE ELSE. ....</b>	<b>9</b>
<b>7. OTHER POINTS OF INTEREST .....</b>	<b>9</b>
7.1 WHAT TO HAND IN.....	9
7.2 PROGRAMMING LANGUAGE.....	10
7.3 GRADING RUBRIC.....	10
7.4 FLOWCHART .....	10
7.5 REPRODUCTION OF SOME FIGURES IN LARGE SCALE.....	12



## 1. Overview and Introduction

In this second programming assignment, you'll be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol, specifically using an implementation of the rdt 3.0 protocol that we studied in class. It will operate over a channel that can corrupt, lose, or delay (but not reorder) messages. This should be fun since your implementation will differ very little from what would be required in a real-world situation.

The sending and receiving sides FSMs for the rdt 3.0 protocol are shown in Figure 1. The most important difference between rdt 3.0 and what you'll implement is that your sender will not be called via an `rdt_send(data)` from above, nor will your receiver deliver data to above via `udt_send(data)`. Instead, as discussed below, your sender will reliably deliver a file from sender to receiver. And, of course, your sender and receiver will be communicating over sockets, over a "live" Internet.



**Figure 1:** rdt 3.0 sender and receiver. The logic and data-transfer protocol behavior will be the same as show here, but the implementation details (e.g., socket calls) will differ from the abstract functions shown here.

In this assignment you will program a sender and receiver that will reliably transfer the first 200 characters of any text file (the example text of the Declaration of Independence is available here: [http://gaia.cs.umass.edu/cs453\\_fall\\_2020/files/declaration.txt](http://gaia.cs.umass.edu/cs453_fall_2020/files/declaration.txt), download this file to the machine where you sender is executing) from sender to receiver, using the rdt 3.0 protocol. The sender and receiver will communicate over the network environment described below in Section 3.

## 2. rdt 3.0 protocol details

Because your sender and receiver will operate over a channel that can corrupt, lose, or delay (but not reorder) messages, you'll want to use checksums, sequence, and acknowledgement numbers



(one-bit numbers will suffice; you are essentially programming a stop-and-go protocol), and timeout and retransmit mechanisms – all part of the rdt 3.0 protocol. Because your sender will need to interact with your receiver, as well as the receiver running on [gaia.cs.umass.edu](http://gaia.cs.umass.edu) and receivers written by other students, and your receiver will need to work with the senders of other students (as described below), we'll need to be incredibly precise about rdt 3.0 protocol packet formats.

To make things as easy as possible for you, a packet should be encoded as a single string. Thus, for example, rather than sending the binary encoding of an integer value such as 1 as a field in the packet, you would send the substring '1' as a field in the packet. The format of a packet sent between sender and receiver should be as follows:

```
<integer sequence number> <space> <integer ACK number> <space>
<20 bytes of characters - payload> <space>
<integer checksum represented as characters>
```

There <integer sequence number> is a '0' or '1' character. So, the string

```
'1 0 That was the time it <checksum represented as characters>'
```

would be a sender-to-receiver packet with a sequence number 1, ACK number 0 (note that the sender-to-receiver ACK value is never used by the receiver), a 20-byte payload of 'That was the time it' and the string representation of the computed checksum computed over the entire prefix up to the start of the checksum, i.e., '1 0 That was the time it'. For a receiver-to-sender ACK, the entire string except for the ACK value itself and the checksum should be blank, e.g.,

```
' 1 <checksum represented as characters>'
```

Note that the length of sender-to-receiver packets and receiver-to-sender packets are exactly 30 (1+1+1+1+20+1+5) bytes.

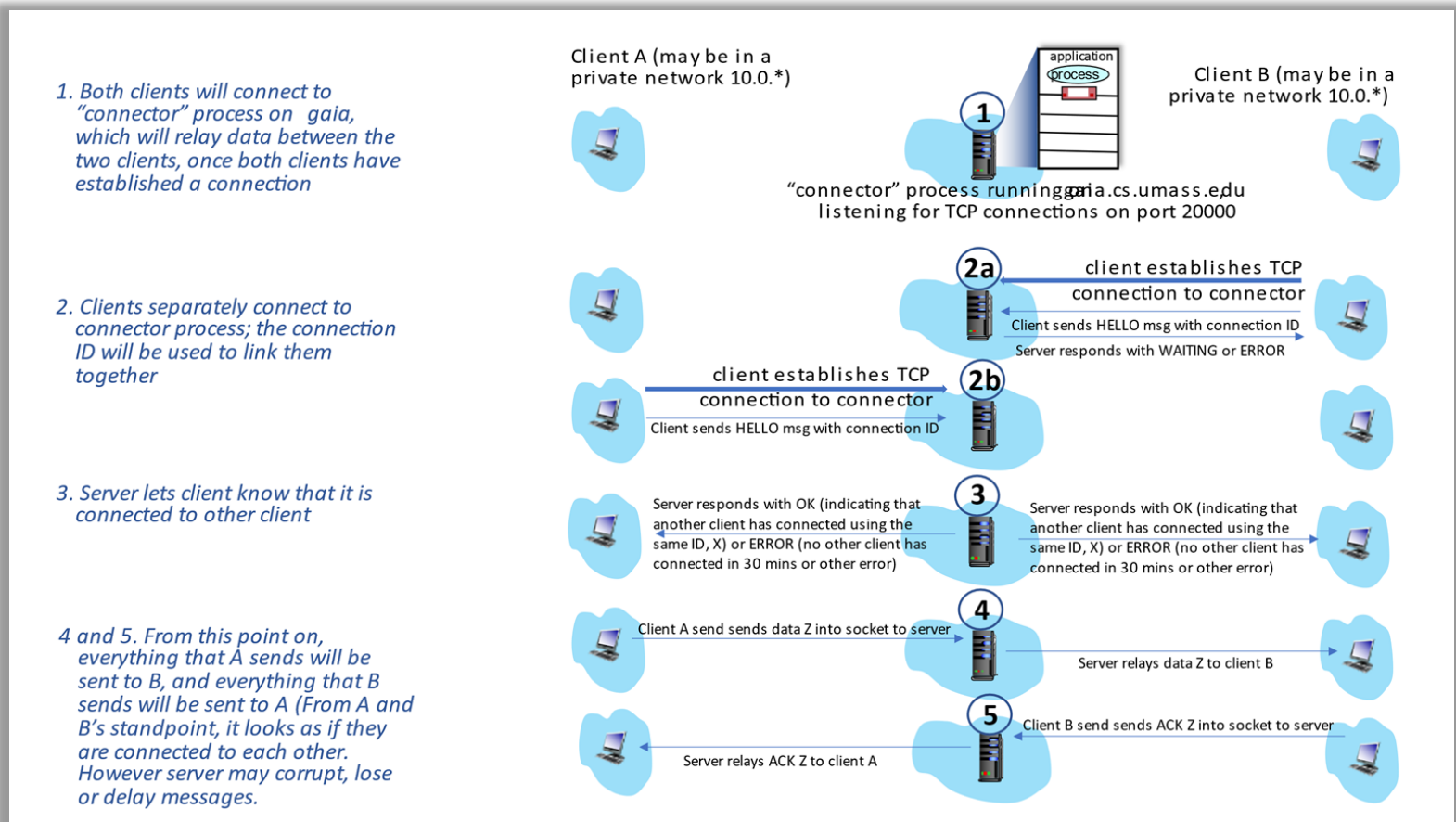
- **Checksumming.** To compute the checksum, add together the ASCII values of all of the bytes in the string as if they were integers, and include the string representation of that sum as the checksum at the end of the packet string. Note that the checksum is NOT computed over the checksum itself, but should be computed over all other bytes in the string, including the sequence number, ACK number, blank spaces, and 20-character payload. Our TA's have written checksumming routines that you can use in Python; you can pick them up at [http://gaia.cs.umass.edu/cs453\\_fall\\_2020/files/checksum.py](http://gaia.cs.umass.edu/cs453_fall_2020/files/checksum.py). One reason it is important to make sure everyone gets the checksum done correctly is that senders and receivers from different students will need to interoperate, and therefore will need to compute the checksum exactly the same way.
- **Timeout/retransmit.** You learned about how to use Python timers in the first programming assignment, so this will hopefully not be too much of a heavy lift. You will



set your timeout value as discussed below. There may be premature timeouts because of packets being delayed within the network environment discussed in section 3, but your protocol (rdt. 3.0) is (as we've learned) able to handle this scenario.

### 3. The network environment

Your sender and receiver will communicate with each other through an intermediary server running on [gaia.cs.umass.edu](http://gaia.cs.umass.edu), as shown below in Figure 2, that will function as an unreliable channel connecting your sender and receiver. Your sender and receiver will each send messages (see section 2 above) to the server, and the server will relay (after possibly corrupting, losing or delaying but not reordering) the message to your other side.



**Figure 2:** A server at [gaia.cs.umass.edu](http://gaia.cs.umass.edu) acting as a relay (possibly

Before your sender and receiver can communicate with each other, they must first each establish a connection with a [gaia.cs.umass.edu](http://gaia.cs.umass.edu) server as follows. Your sender and receiver (which are separate programs) should each:

- Create a TCP socket, as a client, and `connect()` it to port 20000 at [gaia.cs.umass.edu](http://gaia.cs.umass.edu).
- The sender side of your protocol would then send a string of the following form on this socket:



“HELLO S <loss\_rate> <corrupt\_rate> <max\_delay> <ID>”  
while the receiver side of your protocol would send a string of the form:

“HELLO R <loss\_rate> <corrupt\_rate> <max\_delay> <ID>”  
to the server where:

- <loss\_rate> <corrupt\_rate> are float values in the range [0.0, 1.0]. <loss rate> and <corrupt rate> are the probabilities that a message will be lost or corrupted, respectively. Set these values initially to 0.0 when you’re starting to work on this project, which simulates the case that there is no loss and no transmission corruption.
- <max\_delay> is an int value ranging [0, 5] representing maximum delay for your packet at the server. Set this initially to 0 also.
- <ID> is a string with four digits specifying the connection ID of the client. Your sender and receiver should specify the same connection ID (this is how the server will know to relay messages between these two TCP clients). Try to pick a “random” ID since if that number is currently in use by someone else, you’ll get an error message (see below).

An example HELLO message from a sender is

“HELLO S 0.0 0.0 0 1234”.

Your sender and receiver sides can set the parameter values differently, meaning that the sender-to-receiver receiver-to-sender directions of the channel may have different characteristics. Your sender and receiver must make a request with the same connection ID within 60 seconds of each other; otherwise, you’ll get an ERROR message (see below).

- The [gaia.cs.umass.edu](http://gaia.cs.umass.edu) server will respond to your client (your client must do a read from the socket to get this response) with a string with one of several different messages:
  - An “OK” message (containing the string ‘OK’) received by a client means that it is connected to another client that requested that same ID. This is the message you are hoping for! After this point, everything your sender or receiver sends – using the same socket that was used to communicate with the Gaia server – will be relayed to the other side by the [gaia.cs.umass.edu](http://gaia.cs.umass.edu) server.
  - A “WAITING message” (containing the string ‘WAITING’) means that the server is waiting for another client to connect within the next 60 seconds. After receiving a WAITING message, your client should then again read from the socket and will receive, either an OK or an ERROR message.
  - An “ERROR” message (containing the string ‘ERROR’) will include an error reason statement as part of the string beginning with ‘ERROR’ that will indicate what went wrong. Possible errors are an “Incorrect Parameter Values” (indicating one or more of the parameters specified in your HELLO message are invalid), “CONNECTION ID IN USE” (indicating that another sender/receiver pair are already communicating using



the Connection ID you specified) and “NO MATCHING CONNECTION REQUEST” meaning that no other client has made a request in the last 60 seconds for the same Connection ID, so your client could not be connected to another client.

When your sender and receiver are done with their task of reliably transferring a file, they must gracefully close the socket connection to the Gaia server. If your sender closes the TCP connection to the server when it’s done, the Gaia server will then close its two sockets that are used to communicate to your sender and receiver, and this, in turn, will cause your receiver to get a socket closed error message (check!) on a socket read, meaning that your receiver will know it’s done and can gracefully close its socket.

If you want to check out what’s happening at the [gaia.cs.umass.edu](http://gaia.cs.umass.edu) server at port 20000, you can check out its logfile of the requests it has received by checking out the following webpage [http://gaia.cs.umass.edu/cs453\\_fall\\_2020/files/PA2server\\_log\\_v2.php](http://gaia.cs.umass.edu/cs453_fall_2020/files/PA2server_log_v2.php).

## 4. The Assignment

Write a reliable data transfer protocol rdt 3.0 to transfer **the first 200 bytes** (characters) of any text file from sender to receiver. (Reminder: the example text “declaration of independence” is here: [http://gaia.cs.umass.edu/cs453\\_fall\\_2020/files/declaration.txt](http://gaia.cs.umass.edu/cs453_fall_2020/files/declaration.txt), download this file to the machine where your sender is executing).

Please begin the assignment by using the code templates that we provide (available on Moodle).

The sender file is “**PA2\_sender.py**” and invoked as follows from the command line

```
python3 PA2_sender.py <server_ip> <server_port> <connection_id> <loss_rate>
<corrupt_rate> <max_delay> <transmission_timeout> <filename>
```

The receiver file is named “**PA2\_receiver.py**” and invoked as follows from the command line

```
python3 PA2_receiver.py <server_ip> <server_port> <connection_id> <loss_rate>
<corrupt_rate> <max_delay>
```

When starting up, your client and server should then print out your name, the date and time. When the sender or server has set up the socket to the Gaia server and received an OK message from the Gaia server, your client and server should print out the date and time and a string indicating that the channel has been established.

After transferring the file, and before exiting, your client and server then print out the date and time, followed by the **checksum of the 200 bytes of data** from the text file that were sent (sender-side) or received (receiver-size), and statistics of the file transfer (sender-side only): **the total number of packets sent, the total number of packets received, the number of times a corrupted message was received, and the number of timeouts that occurred**. So you’ll need to gather these statistics as your rdt 3.0 protocol operates.





## 5. Getting started

This isn't a project you'll be able to do in an evening, as there are many pieces you'll need to get right: the channel establishment, your use of sequence numbers, ACKs, and timers, and the overall logic and correct operation of your rdt protocol. It is strongly recommended you proceed in steps:

1. Write code for two clients that establish a connection to each other through the [gaia.cs.umass.edu](http://gaia.cs.umass.edu) server, as discussed in section 3. Then abandon this code but keep it handy, as you can cut and paste it into step 3. Note you can do step 2 below before step 1 if you want to.
2. Write your rdt sender and receiver code using sequence numbers, ACK, timers using the message format described in Section 2, and implement the assignment in Section 4. Program your sender and receiver to communicate with each other via TCP sockets, using the loopback interface 127.0.0.1. That is, they'll connect directly to each other in this step, rather than via the Gaia server. This will allow you to get an interoperable sender and receiver up and running in this benign environment first.
3. Integrate your code from steps 1 and 2 into a sender and receiver that connect to the Gaia server and transfer the file over a reliable channel that you configure (via the initial HELLO message discussed in Section 3) with 0.0 loss, 0.0 corruption and 0 delay. When you've gotten this far, you're getting very close to done!
4. Now test and debug your code by selectively turning on more and more channel impairments.
  - a. Initially set channel loss and delay at zero, but channel corruption to a non-zero value, first in just the sender-to-receiver direction, and then in both directions. Does your protocol handle corrupted messages correctly, with no loss or delay?
  - b. Now turn off channel corruption, but turn on channel loss, first in just the sender-to-receiver direction, and then in both directions. Does your protocol handle lost messages correctly, with no corruption or delay?
  - c. Now turn on both channel loss and corruption. Does your protocol handle the occurrences of both corruption and loss in the same execution run?
  - d. Now turn off corruption and loss, but use a high channel delay (e.g., 4 or 5) and a small timeout. Does your protocol handle premature timeouts correctly (e.g., does it transfer the file with no missing or out-of-order data)?
5. If you've got 4a – 4d all working correctly, I'll give you 10-1 odds that when you turn channel corruption, loss, and delay on at the same time, that it will all work without errors. I'll give you 100-1 odds that if you programmed this all at once and turned everything on and started debugging that it would have taken at least 5 times as long to test and debug than having proceeded through steps 4a – 4d one at a time!





## 6. Connecting your sender and receiver to a receiver and a sender written by someone else.

One of the great things about well-specified protocols is that two people can independently write the two different “sides” of a protocol and those implementations should interoperate. That will be the case here as well.

First, you can test your sender by connecting to a receiver on [gaia.cs.umass.edu](http://gaia.cs.umass.edu) that has implemented the receiver-side of the assignment and is wait for you using connection ID 9999 (So please do not use this connection ID unless you are testing your sender with this receiver on [gaia.cs.umass.edu](http://gaia.cs.umass.edu)!). You can configure your channel (loss and corruption probabilities, delay) to this receiver however you’d like. The channel from the receiver to your server will not corrupt, lose or delay messages.

Secondly, you should find another student in the class who is willing to run their receiver to operate with your sender, and willing to their sender with your receiver. So actually, you’re both doing this for each other. You can find another student to connect with by writing and replying to Posts on the Piazza find-a-co-tester thread. Once you find someone willing to co-test with, please take your arrangements to co-test offline (i.e., not on Piazza broadcast) to email, texting or whatever communication mechanism you want. All students are required to demonstrate interoperability of their sender and receiver with another student (see section 7 below), so everyone will need to find at least one partner. And if you’re co-testing with them, then they’re co-testing with you. It’d be great if you’d co-test with multiple other students – you can help each other out, and get to meet other students in the class!

If you’d prefer not to advertise or respond about co-testing via Piazza, just email the instructor ([parviz@cs.umass.edu](mailto:parviz@cs.umass.edu)), and I’ll arrange something. But I encourage you to interoperate with other students in the class – it’ll give you a chance to meet some classmates.

## 7. Other Points of Interest

### 7.1 What to hand in

Please submit (i) your code (PA2\_sender.py and PA2\_receiver.py) on Gradescope for Autograding, and (ii) the screenshots of your sender and receiver in action in PDF file and submit it on Gradescope. Please make sure your programs work correctly in the following cases:

- 1) [Autograding] Set channel loss and delay at zero, but channel corruption to a non-zero value in both directions. Set the channel corruption level high enough so that both sender and receiver receive at least two corrupted messages during execution.
- 2) [Autograding] Set channel corruption and delay at zero, but channel loss to a non-zero value Set the channel corruption loss level high enough so that both sender timeouts out and retransmits at least two times.



- 3) [Autograding] Set channel corruption and loss to zero, but use a high channel delay and a small timeout so that your protocol times out at least two times (note that since no messages are lost or corrupted, these would be pre-mature timeouts).
- 4) [Screenshot] Your sender interoperating with another student's receiver. You should send this screenshot to the other students, so they can hand it in, and they should send you their screenshot so you can hand that in along with your screenshot here.
- 5) [Screenshot] Your receiver interoperating with another student's sender. You should send this screenshot to the other students, so they can hand it in, and they should send you their screenshot so you can hand that in along with your screenshot here.

Also, include text that explains what these screenshots are doing.

- For the autograding part, you can directly upload your sender and receiver code to the Programming Assignment 2 [Autograder] on Gradescope.
- For the screenshot part, put all screenshots into a **pdf file** (with the description for each screenshot). Please remember to put each scenario on a separate page. Name the pdf file as Firstname\_Lastname\_PA2.pdf. Upload the pdf file to Programming Assignment 2 [PDF] on Gradescope.

### 7.2 Programming Language

A note on programming languages: Please write your sender and receiver in **Python3** since that's the language used in our textbook for sockets and our autograder is using it.

### 7.3 Grading rubric

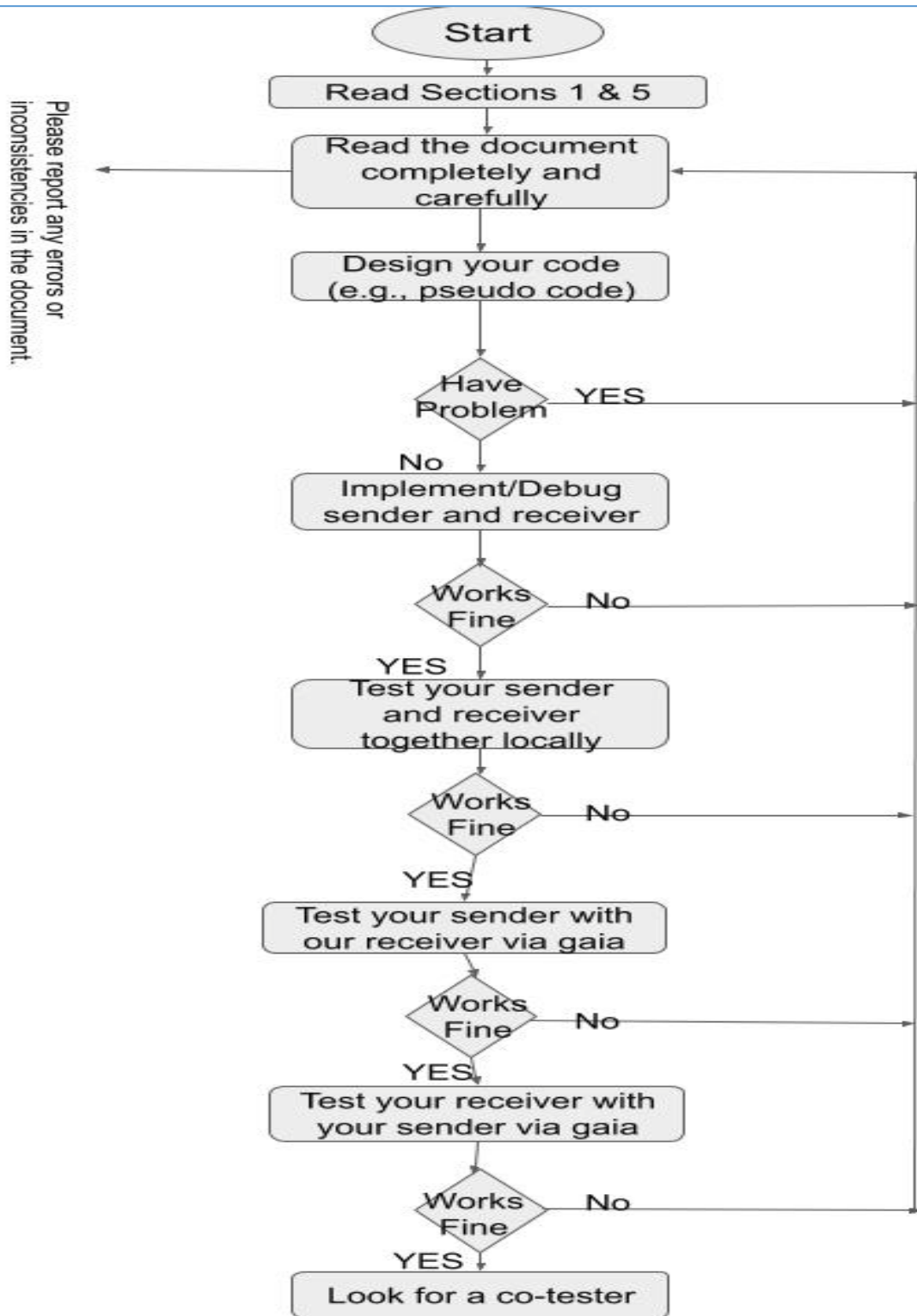
- **Autograder (70%)**: Your submit (both sender and receiver) will be tested with autograder. They will be connect to each other via Gaia server and local server. They will also be tested with our sender and receiver.

- **Screenshot (30%)**: We'll also manually grade the screenshots that you submit in the written document.

### 7.4 Flowchart

The following flowchart helps you implement your project efficiently. Please let us know if you have any questions on any steps.

**Note**: Our receiver is on connection\_id 9999.





## 7.5 Reproduction of some figures in large scale

