California State University Fullerton

CPP 551 Project 03 report

**by**

Rahul Chauhan (887448813,rahulchauhan@csu.fullerton.edu)

Harshit Singh Rathod (887446953, rathod10892@csu.fullerton.edu)

Contents

[1. Introduction 3](#_Toc27448374)

[2. Logical clock 4](#_Toc27448375)

[2.1. LAMPORT LOGICAL CLOCK 4](#_Toc27448376)

[2.2. IMPLEMENTATION 5](#_Toc27448377)

[3.ZEROMQ(ZMQ) 6](#_Toc27448378)

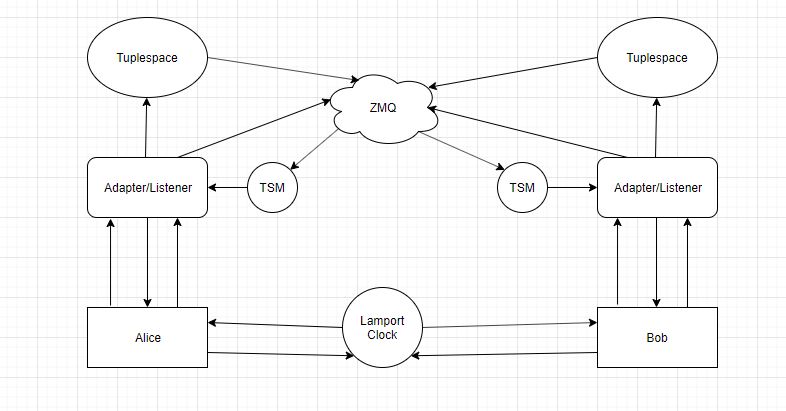
[3.1. IMPLEMENTAION 6](#_Toc27448379)

[4.Identify problem and their solutions 7](#_Toc27448380)

# Introduction

In previous project each user has their own tuplespace and they could post and read messages of each other via interacting with their own tuplespace. It tries to create an environment of distributed systems using variety concepts, including multicast, logging, recovery. While creating the previous project we faced some problem like duplication, inconsistency, sequencing, faults, replication. In this project we are trying to resolve two Major problem from the previous project Duplication, sequencing issue with a combination of Lamport logical clock and Zero MQ.

The basic design of Project 03 looks like below.



For simplicity we have only shown two clients, however we can add more. Clients are connected to Lamport clock which provide them a unique order. The client programs (e.g. client\_alice.py) have interactive console where users can input commands (e.g. write/read/take). The adapter from project 02 have been slightly changed to also behave as listener. When a client program runs it checks if adapter is up (ready for events), if not it waits for 99s and repeats once more.

Instead of using udp we are using tcp protocols along with ZMQ XPUB\_XSUB. This allowed us to keep only one listening/subscription port and one publishing/multicast port.

# Logical clock

 In distributed systems, physical clocks are not always precise, so we can't rely on physical time to order events. Instead, we can use **logical clocks** to create a partial or total ordering of events. Logical clocks to help us order events according to our 🡪 ordering. Unlike physical clocks which are physical entities that assign physical times to events, our clocks are simply a conceptualization of a mathematical function that assigns numbers to events. These numbers act as timestamps that help us order events. Since our clocks logically order events, rather than physically order them, we call them **logical clocks**.

## 2.1. LAMPORT LOGICAL CLOCK

Lamport defined a relation called happensbefore. The expression a 🡪 b is read “event a happen before event b” and means that all processes agree that first event a occurs, then afterward, event b occurs. The happens-before relation can be observed directly in two situations:  
1. If a and b are events in the same process, and a occurs before b, then a 🡪 b is true.  
2. If a is the event of a message being send by one process, and b is the event of the message being received by another process, and b is the event of the message being received by another process, then a 🡪 b is also true, A message cannot be received before it is sent, or even at the same time it is sent since it takes a finite, nonzero amount of time to arrive.

## 2.2. IMPLEMENTATION

Using Lamport clock we were able to remove duplicacy from the system as every event will generate a timestamp and this timestamp will always be greater than previous one. On restarts of tuplespace messages are replayed, but since they are old, are not logged in file.

Also, every message will come with unique timestamp and it will help to create ordering in the system due to the nature of Lamport clock. This helped us to resolve sequencing issue which is discussed below.

# 3.ZEROMQ(ZMQ)

The ZeroMQ lightweight messaging kernel is a library which extends the standard socket interfaces with features traditionally provided by specialized messaging middleware products. ZeroMQ sockets provide an abstraction of asynchronous message queues, multiple messaging patterns, message filtering (subscriptions), seamless access to multiple transport protocols and more.

## 3.1. IMPLEMENTAION

First use of ZMQ request reply pattern is to contact with Lamport clock, client send request to Lamport clock with is own timestamp and Lamport clock will reply the request by incrementing the highest timestamp by 1, and because of request reply pattern message will never get lost so it will be sure that client request will always get the messages from Lamport clock.

Second use of ZMQ is XPUB and XSUB pattern which is many to many connections because of that now there is no need to have different ports now only 2 port number is required, one for publisher and other for subscriber. When an event happens ts-adapter pairs will publish that message and all the TSM(tuplespace manager) will subscribe on that port. So, when message is published, all TSM is notified and respective logging and recovery happens.

# 4.Identify problem and their solutions

* What the problem is

**Duplicates:** Since we are logging the events of tuplespace suppose there are 6 logged events and tuplespace restarts. In this situation the logging and recovery mechanism recovers the tuplespace but tuplespace again notifies those events and duplicate logs (total 12) are generated in file.

**Sequencing:** In scenario where two users are constantly performing operations on their own tuplespace, the order of operations on both tuplespaces are not same.

* What triggers the problem, and how to reproduce it?

**Duplicates:** During recovery tuplespaces again notify those events. There is no way to block previous message from notification.

**Sequencing:** All the operations are unordered hence program can not determine which event happened before which event.

* What impact the problem has on the system

**Duplicates:** It makes logs inconsistent and when a tuplespace restarts twice those duplicates values are written again eventually making tuplespaces inconsistent. It also increases log size which is taking a lot of unnecessary space.

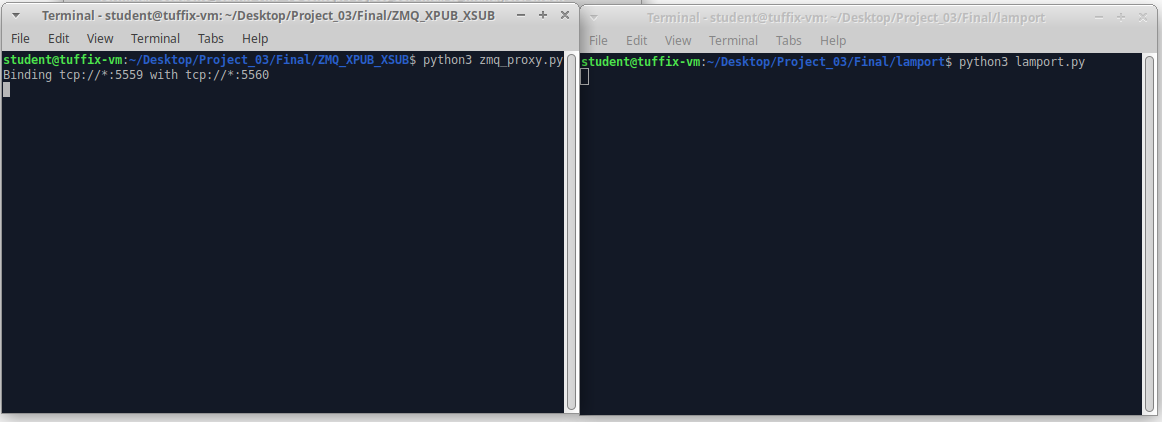
**Sequencing:** It results serious issues. Suppose we have one write and one take operations. In one tuplespace write is done before take and in other take happen before. The latter will block the tuplespace. In absence of ordering these issues continues to persist.

* What the correct behavior should have been

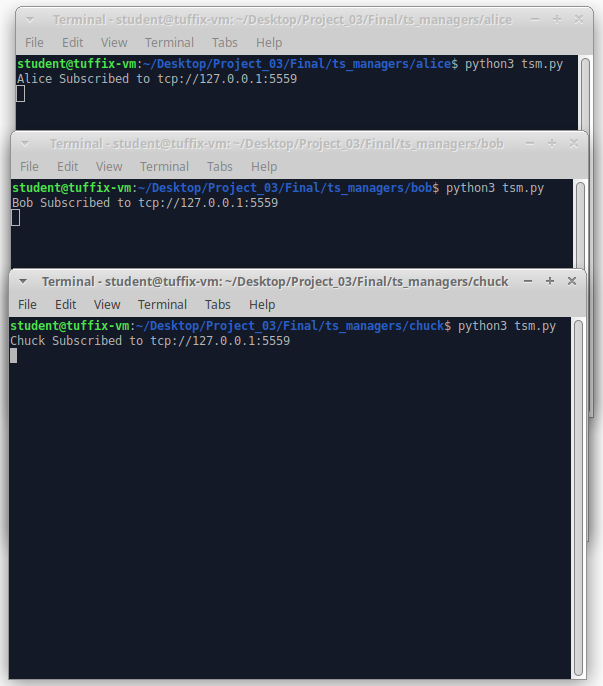
**Duplicates:** Previous messages should not be logged. There should be no message duplicacy in the system it should just avoid having duplicate message.

**Sequencing:** Messages should be written in ordered manner. There should be an order in the message so they can appear in order.

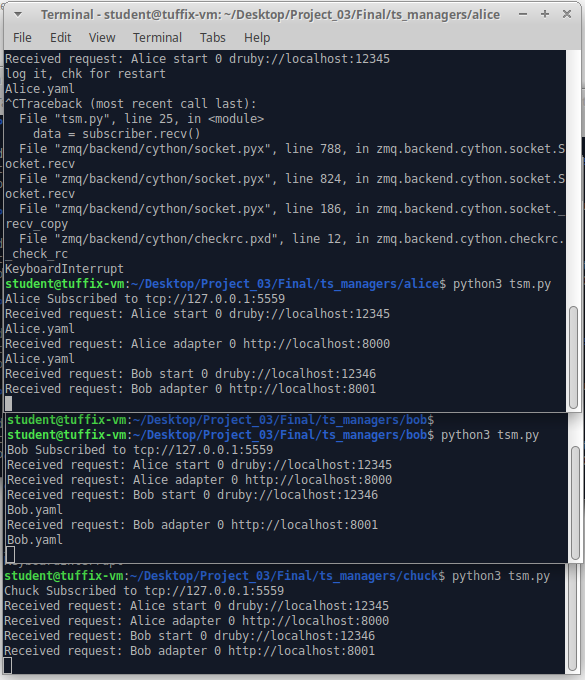
Run the common programs: zmq\_proxy.py and Lamport.py



Start ts managers:



Start ts\_ada pairs for alice and bob:



We did a slight modification while sending notification.

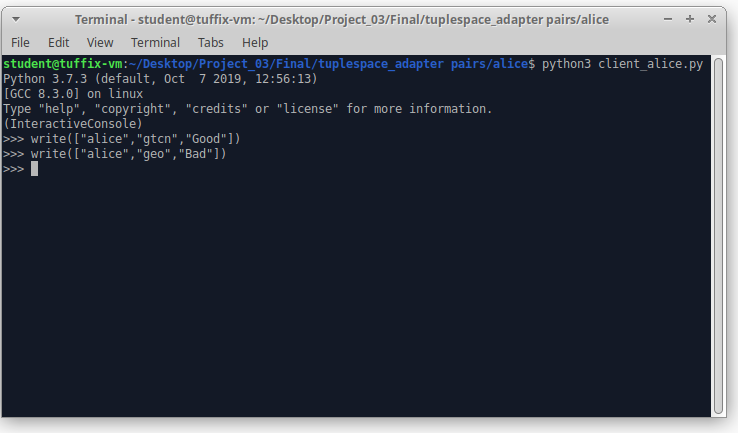
First: Name of ts

Second: Operation(start,adapter,write,read)

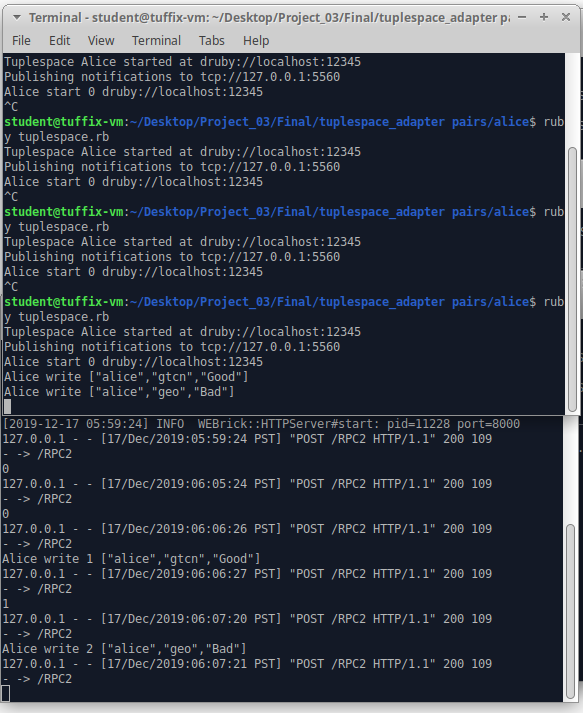
Third: Sequence of event(for ts and adapter its 0)

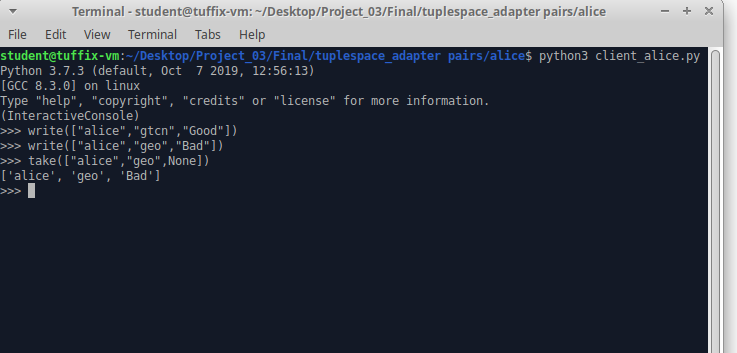
Fourth: Content

Start the client and write 2 events and then one take event: client\_alice.py

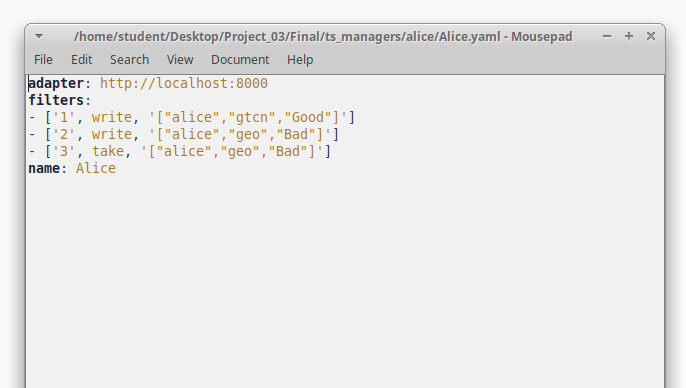


State of ts-adapter:

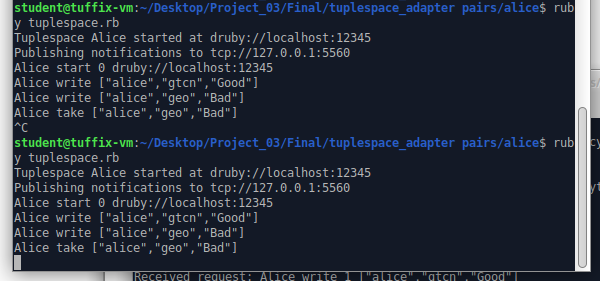




State of log file:

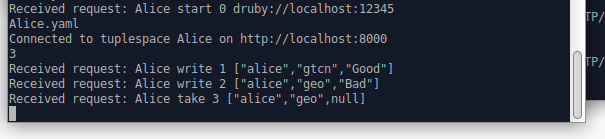


Let’s restart the tuplespace:

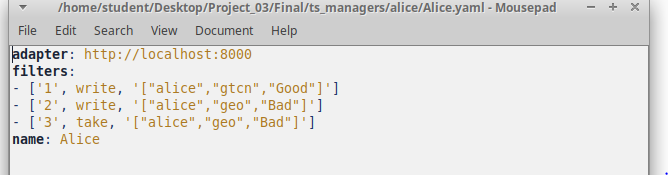


As we can see the state is recovered.

On receiver side they are received but are not logged to file since they are out of order:



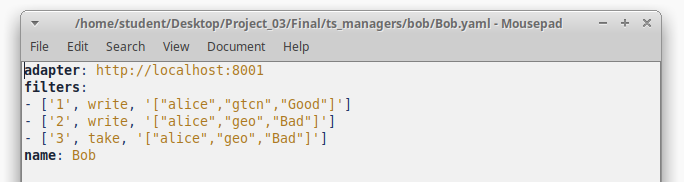
No change in log file:



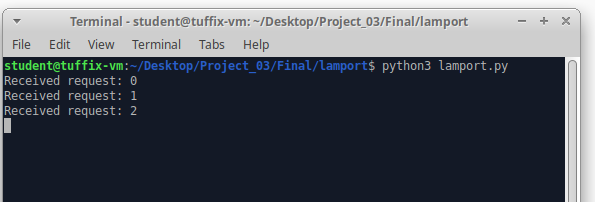
Hence by ordering we can remove duplicates.

Similarly using ordering these messages are propagated to other users in same order as well.

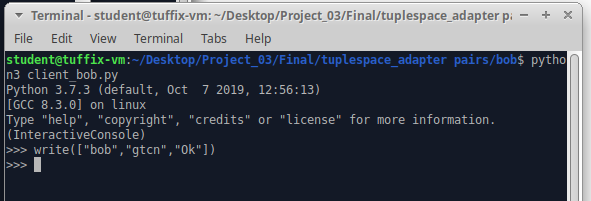
e.g. log of bob:

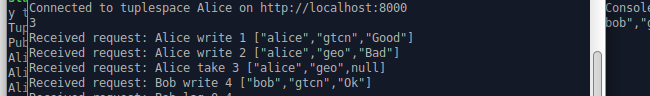


When bob tries to write new message then it contacts lamports whose current state is:

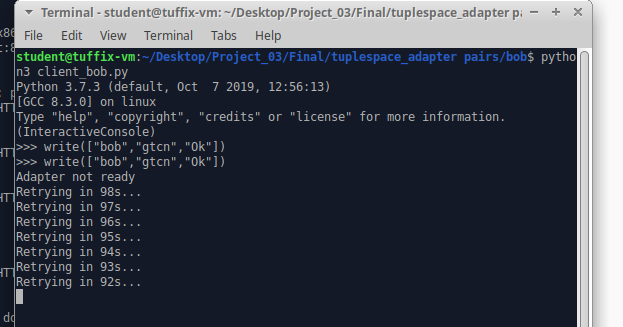


As we know we have three events before: when bob tries to write it will get a sequence as 4. Hence the order is maintained for all the events.





All messages are written in sequence. If the adapter is down and client notifies it, then it retries twice for 99s:



By doing this we are making sure that a message (more importantly sequence) is not lost.

Hence we can remove sequencing issue by this method.