分布式系统作业

第 4 次作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

理论题：

1、Leader selection 在分布式系统中具有重要的用途，主要用于容错，即当主节点失效后能够从备份节点中选择新的 leader，但是新选择的 leader 需要得到其他节点的认同。主流的 leader 选择算法有： Bully、 Ring 算法，但不限于这些算法，调研以下软件，简述这些软件所采用的选举算法： Zookeeper、 Redis、 MongoDB、 Cassandra。

1. Zookeeper

Zookeeper 的 Leader 选举存在两个阶段：一是服务器初始化启动时，二是运行过程中 Leader 服务器发生宕机。

参数如下：

服务器 ID（myid）：编号越大在选举算法中权重越大；

事务 ID（zxid）：值越大说明数据越新，权重越大；

逻辑时钟(epoch-logicalclock)：同一轮投票过程中的逻辑时钟值是相同的，每投完一次，值会增加。

各节点的状态有以下 4 种：

LOOKING：竞选状态

FOLLOWING：随从状态，同步 Leader 状态，参与投票

OBSERVING：观察状态，同步 Leader 状态，但不参与投票

LEADING：领导者状态

对于第一种情况，服务器初始化启动时：

（1）每个 Server 发出一个投票。投票包含所推举的服务器的 myid 和 zxid，并各自将这个投票发给集群中其他机器。

（2）各服务器接受来自其他各个服务器的投票。集群的每个服务器收到投票后，需要判断该票是否有效。

（3）处理投票。选拔规则如下：

① 优先比较 zxid。zxid 较大的服务器优先作为 Leader。

② 若 zxid 相同，则比较 myid。myid 较大的服务器优先作为 Leader。

（4）统计投票。每次投票结束后后，服务器都会统计投票信息，判断是否已经有过半的机器接收到相同的投票信息。若已过半，则新的 Leader 被选出。

（5）修改各服务器状态。新的 Leader 确认后，每个服务器需要更新自己的状态（FOLLOWING，LEADING）。

对于第二种情况，运行过程中 Leader 服务器发生宕机时：

（1）非 Leader 服务器宕机或有新成员加入，仍然正常运行；

（2）若 Leader 服务器发生宕机，则整个集群需要暂停对外服务。各服务器需要更新自己的状态，其他非 OBSERVER 的服务器需要将自身状态更新为 LOOKING。

（3）进入到新一轮的投票阶段，规则和流程仍与第一种情况相同。

2. Redis

Redis 采用哨兵机制进行选举，其中每个集群有 master 和 slave，他们的关系为上下级。

其过程如下：

（1）slave 发现自己的 master 变为 FAIL。

（2）slave 将自己记录的集群 currentEpoch（选举轮次标记）加 1，并广播并 FAILOVER_AUTH_REQUEST 信息给集群中其他节点。

（3）其他节点收到该信息，只有 master 响应，判断请求者的合法性，并发送 FAILOVER_AUTH_ACK，一个 epoch 只发送一次 ack。

（4）尝试 failover 的 slave 收集 master 返回的 FAILOVER_AUTH_ACK，当某个 slave 收到超过半数 master 的 ack 后变成新 master。

（5）广播消息通知其他集群节点。

3. MongoDB

MongoDB 节点之间维护心跳检查，主节点选举由心跳触发。

（1）心跳检查

MongoDB 复制集成员会向自己之外的所有成员发送心跳并处理响应信息，因此每个节点都维护着从该节点 POV 看到的其他所有节点的状态信息。节点根据自己的集群状态信息判断是否需要更换新的 Primary。

在实现的时候主要由两个异步的过程分别处理心跳响应和超时，抛开复杂的条件检查，核心逻辑主要包括：

① Secondary 节点权重比 Primary 节点高时，发起替换选举；

② Secondary 节点发现集群中没有 Primary 时，发起选举；

③ Primary 节点不能访问到大部分（Majority）成员时主动降级；降级操作会断开链接，终止用户请求等。

（2）选举发起

发起选举的节点需要首先做一些条件判断，比如节点位于备选节点列表中、POV 包含复制集 Majority 等，真实情况的条件判断更加复杂。然后将自己标记为选举过程中，并发起投票请求。

（3）投票

投票发起者向集群成员发起 Elect 请求，成员在收到请求后经过会一系列检查，如果通过检查则为发起者投一票。一轮选举中每个成员最多投一票，在 PV0 中用 30 秒“选举锁”避免为其他发起者重复投票，这导致了如果新选举的 Primary 挂掉，可能 30 秒内不会有新的 Primary 选举产生；在 PV1 中通过为投票引入单调递增的 Term 解决重复投票的问题。

如果投票发起者获得超过半数的投票，则选举通过成为 Primary 节点，否则重新发起投票。

（4）注意事项

① MongoDB 选举需要获得大多数投票才能通过，在一轮选举中两个节点得票相同则重新选举，为避免陷入无限重复选举，MongoDB 建议复制集的成员个数为奇数个，当 Secondary 节点个数为偶数时，可以增加一个 Arbiter 节点。

② PV0 版本中，所有成员都可以投否决票，一个否决票会将得票数减少 10000，所以一般可以认为只要有成员反对，则该节点不能成为 Primary。PV1 版本取消了否决票。

③ 选举过程中，复制集没有主节点，所有成员都是只读状态。

4. Cassandra

Cassandra 主要使用了 Gossip 算法。

Gossip 算法，灵感来自办公室八卦，只要一个人八卦一下，在有限的时间内所有人都会知道该八卦的信息，这种方式也与病毒传播类似，因为 Gossip 有众多的别名"闲话算法"、"疫情传播算法"、"病毒感染算法"、"谣言传播(Rumor-Mongering)算法"。

但 Gossip 并不是一个新东西，之前的泛洪查找、路由算法都归属于这个范畴，不同的是 Gossip 给这类算法提供了明确的语义、具体实施方法及收敛性证明。

特点:

Gossip 算法又被称为反熵（Anti-Entropy），熵是物理学上的一个概念，代表杂乱无章，而反熵就是在杂乱无章中寻求一致。

这充分说明了 Gossip 的特点：在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致. 每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节可以通过网络连通，最终他们的状态都是一致的，当然这也是疫情传播的特点。

要注意到的一点是，即使有的节点因宕机而重启，有新节点加入，但经过一段时间后，这些节点的状态也会与其他节点达成一致，也就是说，Gossip 天然具有分布式容错的优点。

实验题：

2、可靠多播在分布式系统中具有重要的用途，比如传播选举消息等，可靠多播的实现方式有多种，请从以下软件中选择一种，编译运行，观察是否可以实现可靠多播，并撰写报告。

我选择第 4 个链接：

https://github.com/daeyun/reliable-multicast-chat

具体操作流程如下：

（1）下载链接中的压缩包并解压；

（2）在 C:\Users\93508\Desktop\reliable-multicast-chat-master\reliable_multicast_chat 目录下打开 4 个 cmd 窗口（相当于 4 个节点）

（3）运行方法为：

```
python main.py [process ID] [delay time] [drop rate]
```

所以在 4 个 cmd 窗口分别输入以下指令:

```
python main.py 0 0.1 0.1

python main.py 1 0.1 0.1

python main.py 2 0.1 0.1

python main.py 3 0.1 0.1
```

运行发现报错 AttributeError: module 'socket' has no attribute 'SO_REUSEPORT'

查询资料后可知不同操作系统设置 socket，SO_REUSEPORT 选项不同，Windows 只能识别 SO_REUSEADDR。

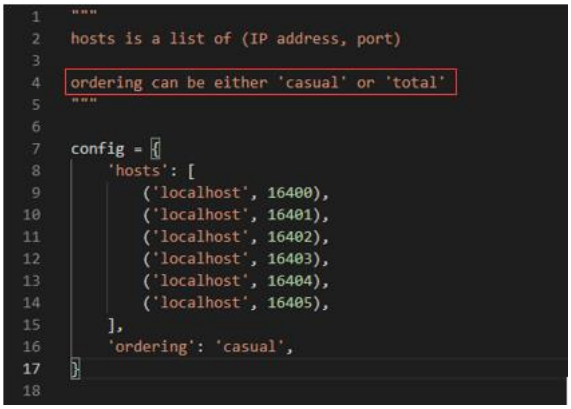
所以将 chat_process.py 中的第 39 行从：

```
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
```

修改为：

```
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

查看 config.py 可知：

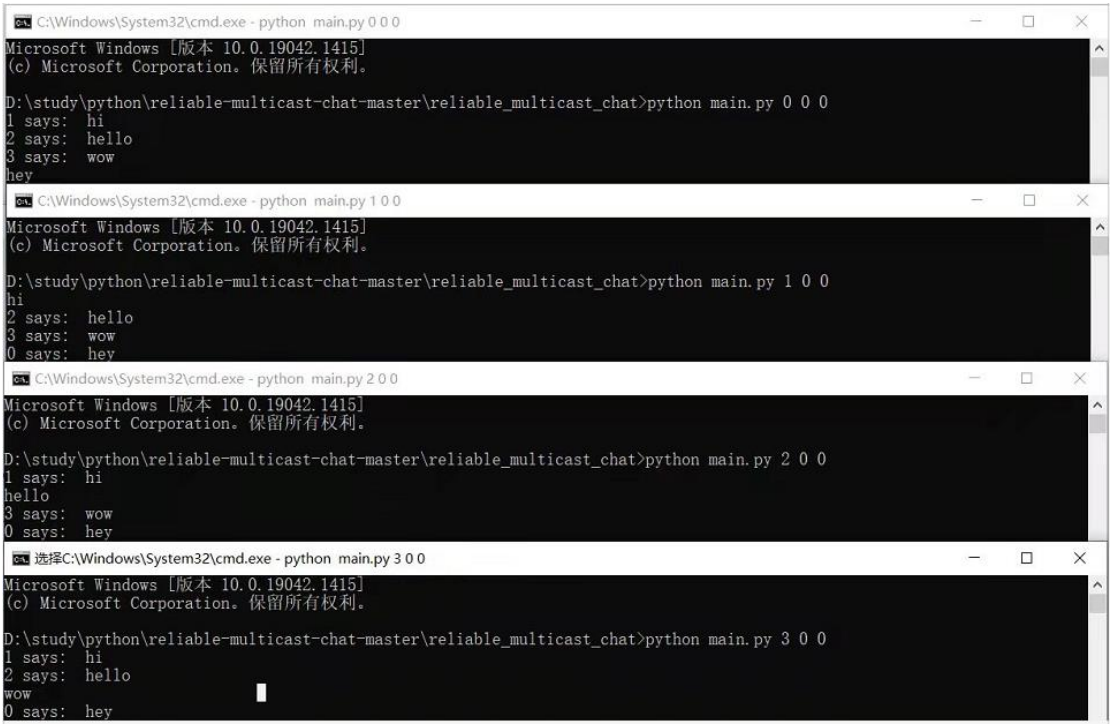


```
"""
hosts is a list of (IP address, port)

ordering can be either 'casual' or 'total'
"""

config = {
    'hosts': [
        ('localhost', 16400),
        ('localhost', 16401),
        ('localhost', 16402),
        ('localhost', 16403),
        ('localhost', 16404),
        ('localhost', 16405),
    ],
    'ordering': 'casual',
}
```

ordering 的方式为 casual，由注释和 github 链接可知，还有一种排序方式为 total。

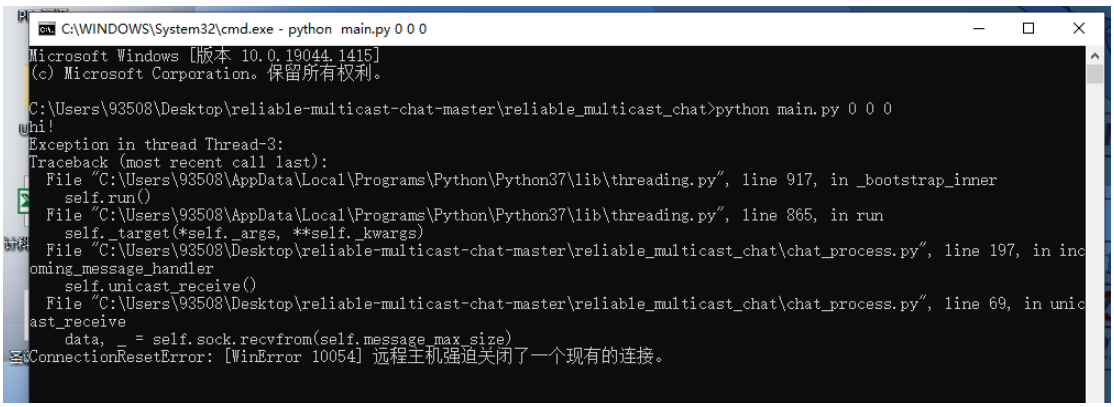
- **Casual Ordering**: When a process receives a multicasted message, it pushes the received message to a buffer queue with a vector timestamp from the sender. And then `update_holdback_queue_casual()` takes care of updating the buffer and actually delivering the message to the process. It compares the timestamps of the buffered messages with the process's timestamp to determine which messages should be kept in the buffer or be removed and delivered to the process.
- **Total Ordering**: Process 0 is always designated as the sequencer. When a process multicasts a message, other processes will hold that message in a buffer until they receive a marker message from the sequencer indicating the order of that message. When the sequencer receives a message, it increments an internal counter and assigns that number as the message's order number which will then be multicasted to all other processes.

但是在 Windows 系统运行时结果不稳定，有时候能实现多播（见下页）：



0 发送的 hey，1 发送的 hi，2 发送的 hello，3 发送的 wow，均被其他三个线程收到。

但有时候也会报错，且频率较高：



原因未知，经同学指导后得知：还是应该将原程序放到 ubuntu 上运行，而非 Windows。

所以将 chat_process.py 中的第 39 行重新从：

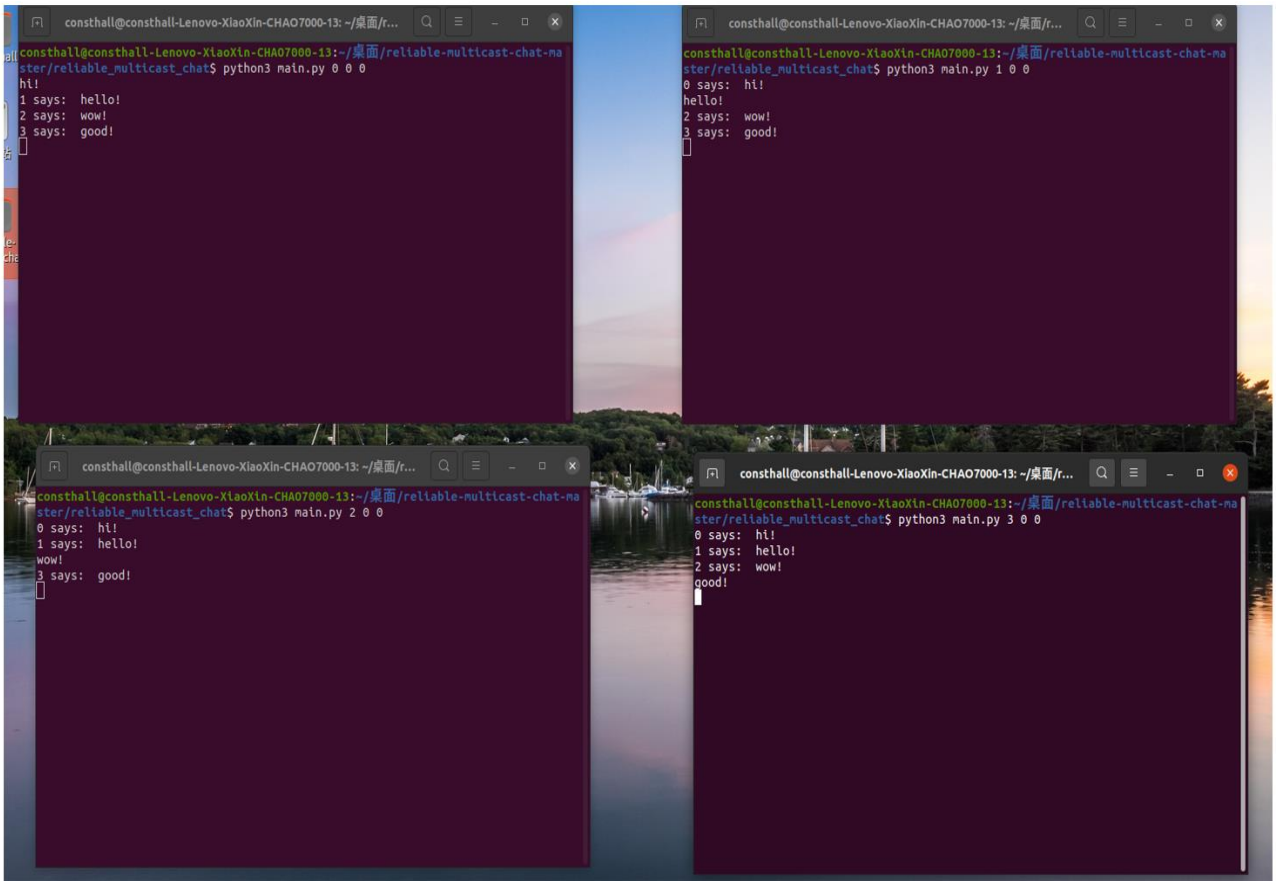
```
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

修改为：

```
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
```

在 ubuntu 上运行结果为：

（1）ordering：casual



0 发送的“hi!”，1 发送的“hello!”，2 发送的“wow!”，3 发送的“good!”，均被其他三个线程收到，所以可靠多播成功。

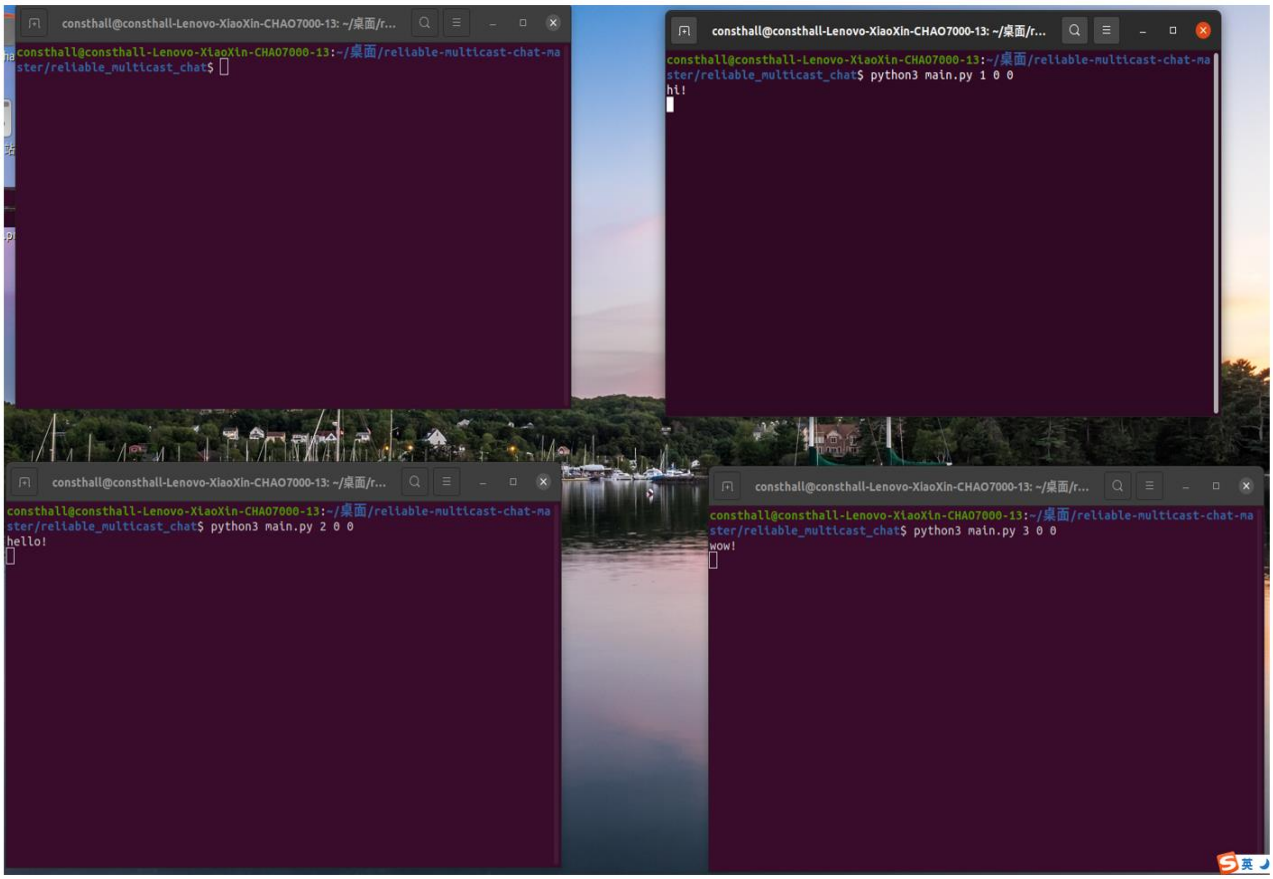
（2）ordering：total（图见下页）

```
"""
hosts is a list of (IP address, port)

ordering can be either 'casual' or 'total'
"""

config = {
    'hosts': [
        ('localhost', 16406),
        ('localhost', 16407),
        ('localhost', 16408),
        ('localhost', 16409),
        ('localhost', 16410),
        ('localhost', 16411),
    ],
    #'ordering': 'casual',
    'ordering': 'total',
}

```

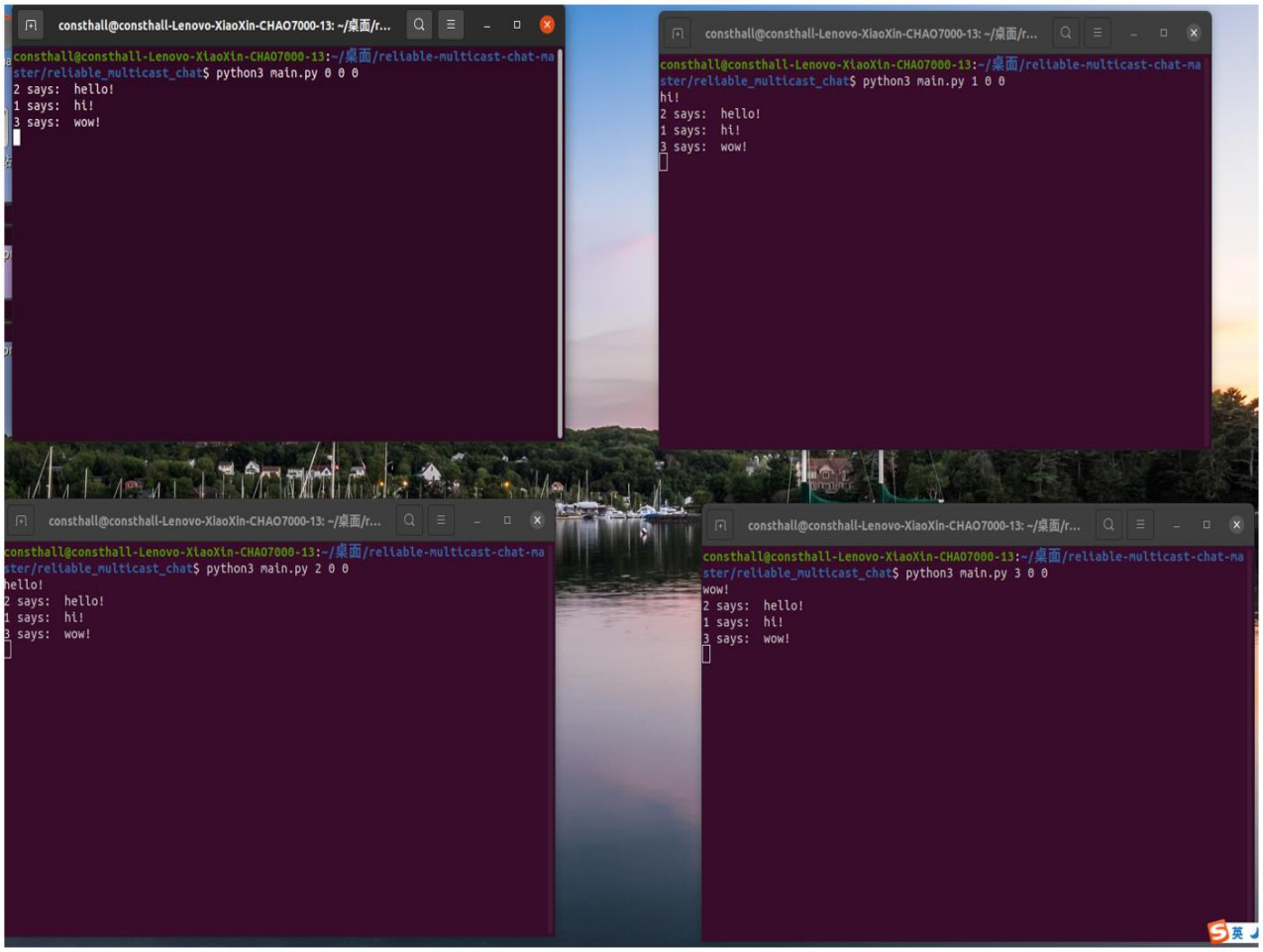
算法

- Casual Ordering：当一个进程接收到一个多播消息时，它会将接收到的消息推送到一个带有来自发送者的向量时间戳的缓冲区队列。然后`update_holdback_queue_casual()`负责更新缓冲区并将消息实际传递给进程。它将缓冲消息的时间戳与进程的时间戳进行比较，以确定哪些消息应该保留在缓冲区中或删除并传递给进程。
- **总排序**：进程 0 始终被指定为定序器。当一个进程多播一条消息时，其他进程会将该消息保存在缓冲区中，直到它们从定序器接收到指示该消息顺序的标记消息为止。当定序器接收到一条消息时，它会增加一个内部计数器并将该编号指定为消息的顺序号，然后将其多播到所有其他进程。

由上图可知，若为创建 0 号进程，则其余进程发送的消息都会保存在缓冲区中无法发送给其他进程。



创建 0 号进程后，发现缓冲区中的消息均已发送出去，可靠多播成功（图见下页）。


consthall@consthall-Lenovo-XiaoXin-CHAO7000-13: ~/桌面/r...
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面/reliable-multicast-chat-master/reliable_multicast_chat$ python3 main.py 0 0 0
2 says: hello!
1 says: hi!
3 says: wow!
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13: ~/桌面/r...
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面/reliable-multicast-chat-master/reliable_multicast_chat$ python3 main.py 1 0 0
hi!
2 says: hello!
1 says: hi!
3 says: wow!
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13: ~/桌面/r...
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面/reliable-multicast-chat-master/reliable_multicast_chat$ python3 main.py 2 0 0
hello!
2 says: hello!
1 says: hi!
3 says: wow!
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13: ~/桌面/r...
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面/reliable-multicast-chat-master/reliable_multicast_chat$ python3 main.py 3 0 0
wow!
2 says: hello!
1 says: hi!
3 says: wow!