# CSE535: Distributed Systems (Fall 2021)

Scott D. Stoller, Stony Brook University

**Problem Set 1** (version 2021-10-16), Due 25 October 2021

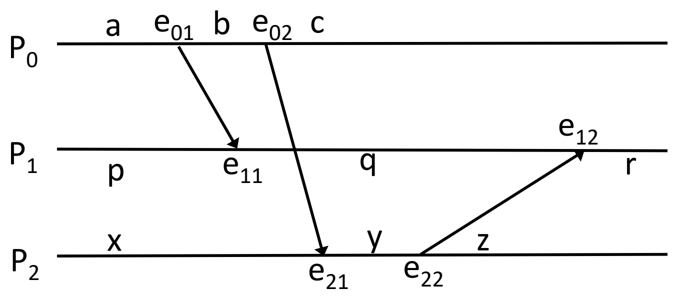
**Pratik Nagelia - [redacted] , Shubham Agrawal - [redacted], Sumeet Pannu - [redacted]**

**INSTRUCTIONS:** Do this assignment with your project teammates. Include the team name and the names of all team members in the submission. **Justify your answer for every problem**, except where directed otherwise. All problems have the same weight. Exactly one member of each team should submit the assignment on Blackboard, as a pdf file formatted for 8.5"×11" paper, by 11:59pm on the due date. I recommend that each team member work on all problems by himself/herself, then the team can meet to pool everyone's thoughts and organize the final write-up. This approach will help every team member be better prepared for the exam.

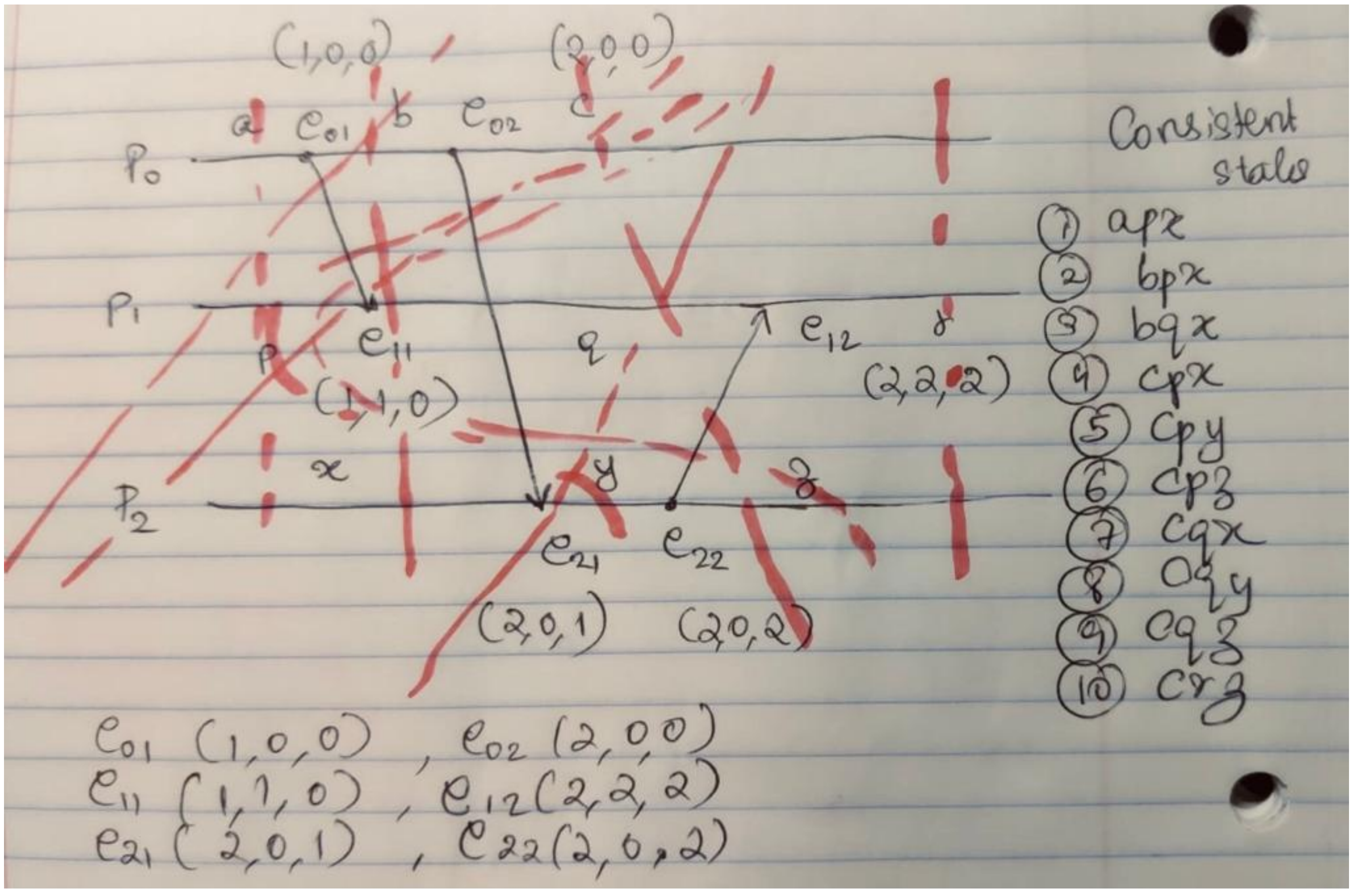
## Problem 1

(a) What is the vector time of each event $e_{ij}$ in the following computation? You do not need to give a justification for this answer.

(b) List all consistent global states of the following computation, in lexicographic order. Write each global state as a tuple of local states; note that a, b, c, p, q, r, x, y, and z are local states. For example, write the initial state as (a, p, x). You do not need to give a justification for this answer.



**Solution:**



Consistent state

1. apx
2. bpx
3. bqx
4. cpx
5. cpy
6. cpz
7. cqx
8. cqy
9. cqz
10. crz

$e_{01}$ (1,0,0), $e_{02}$ (2,0,0)
$e_{11}$ (1,1,0), $e_{12}$(2,2,2)
$e_{21}$ (2,0,1), $e_{22}$(2,0,2)

Figure 1: The vector time of each event and all the consistent global states

## Problem 2

The reliable broadcast algorithm in the lecture slides does not consider failures. Here is Python-style pseudocode for a modified version that considers crash failures with accurate failure detection, i.e., the fail-stop model. The algorithm removes failed processes from the group. For simplicity, assume there is only one (anonymous) group. `send` and `receive` are provided by an underlying reliable FIFO unicast algorithm that retransmits messages when necessary to hide network failures that drop messages.

```
Initialization at process p:
  rcvd := emptyset; G = set of group members
Upon failed(q) at process p:   # notification that q failed
  remove q from G
Upon R-multicast(m) at process p:
  add m to rcvd
  for each q in G-{p}: send m to q
  R-deliver(m)
Upon receive(m) from q at process p:
  if m not in rcvd
    add m to rcvd
    for each r in G-{p,q}: send m to r   # relay the message
    R-deliver(m)
```

Each process p's reliable FIFO unicast algorithm must store each received message m until it knows that every group member has received m, in case every other process that received m fails and p needs to retransmit m. A message is *stable* if all live group members received it. Give pseudocode for an extension of the above algorithm in which processes learn when messages are stable. The extension should satisfy the following design requirements: (R1) the algorithm may piggyback additional information on each message, but it may not send additional messages. (R2) a process should learn that a message is stable as quickly as possible, subject to the limitations implied by (R1).

To simplify the pseudocode, when a process learns that a message m is stable, it should simply include m in a set `stable` of stable messages (although, in reality, it would instead remove m from the unicast algorithm's buffer).

Note that the reliable FIFO unicast `send` statement may return before the destination actually receives the message, and that waiting for an acknowledgement and retransmitting the message if necessary, occur *in the background*. This is how sends on TCP sockets work, for example. With this assumption about `send`, the `for`-loop sends the message to all group members in parallel.

Hint: If you add more than about 10 lines of pseudo-code, your solution is probably unnecessarily complicated.

**Solution:**

```
Initialization at process p:
  rcvd := emptyset; #list of messages that are not yet stable and are pending
  G = set of group members
  stable_messages = {} #list of stable messages

Upon failed(q) at process p:   # notification that q failed
  remove q from G

Upon R-multicast(m) at process p:
  add m to rcvd
  #Piggyback the initiating sender of the message and the list of stable messages
  for each q in G-{p}: send (m, p, stable_messages) to q
  R-deliver(m)

Upon receive(m, sender_list, stableM) from q at process p:
    for each msg in rcvd: #Delete previously seen messages that have been stabilized until now
        if msg is in stableM:
            rcvd.delete(msg)

    if m not in rcvd
        add m to rcvd
        sender_list.append(p) #Append the current process in the sender_list
        for each r in G-{p,q}: send (m,sender_list, stable_messages) to r   # relay the message
        R-deliver(m)
    else if m is in rcvd
        allLiveProcessesRecvdTheMessage = true
        #Excluding p, check if all the remaining group nodes have received the message
        for each r in G-{p}:
            #If process r is absent, it means that process r has not yet seen the current message
            if r is not in sender_list:
                allLiveProcessesRecvdTheMessage = false

        if allLiveProcessesRecvdTheMessage=true: #all live processes have received the message
            rcvd.delete(m) #Remove m from rcvd when it has stabilized
            #Add the message that got stabilized to the stable message list
            stable_messages.append(m)
```

**Note:** We are dependent on the last message to piggyback the set of stable message Ids. During the last message from the client, it will not be able to detect stable message as no more messages will be coming in. To avoid such scenarios, we can add a timeout mechanism which stores the last message timestamp. If the last message timestamp is above this threshold, we can piggyback the stable list of messages IDs along with system level messages like heartbeat.

## Problem 3

Describe how to extend Chandy and Lamport's snapshot algorithm with a dissemination phase, after which the initiator has a complete copy of the recorded state. The dissemination phase should use at most $|P|$ messages, where $P$ is the set of processes.

Notes: The network connectivity graph is not necessarily a complete graph. A process can send a message only to a process with which it shares a communication channel. Each process initially knows only the identities of its neighbors. It does not know the set of all processes. Channels are unidirectional, but you can assume that channels come in pairs, i.e., if there is a channel from $p$ to $q$, then there is also a channel from $q$ to $p$.

**Solution:**

As the network connectivity graph is not necessarily a complete graph, concept of minimum spanning tree is used. Every node stores the parent of its node. Initiator is generally a root of the non-connected graph as it does not have any cycles. Once the initiator starts the message, it sends the request to direct neighbours (processes) using Chandy and Lamport's algorithm. This further traverses down until it reaches to the nodes with no more neighbours or processes left to send. In the dissemination phase, these last set of processes in the hierarchy send its recorded state to its direct parent. The direct parent waits for responses till it receives all the messages from its descendants, appends all the results along with its own and sends the result to its direct parent. This processes keeps on happening till the initiator or the root process gets recorded data of all its descendants. Once this state is reached, the initiator has a complete copy of all the recorded states. Since, the children is just sending one message to its direct parent which in turn sends one message to its direct parent, the whole dissemination phase takes a maximum of P messages where P is the number of processes in the distributed system.

## Problem 4

In Raft, how much delay is caused if the leader crashes briefly? Specifically, suppose the leader crashes, immediately recovers, and gets re-elected. What is the minimum time from when the leader recovers until the service can send a response to a request that has been executed in the new term? Express your answer in terms of the election timeout (ET), the broadcast time (BT), and any other relevant quantities. Recall that "broadcast time", in the paper's terminology, is the round-trip time for a set of parallel RPCs to all servers ("round-trip time" would be better terminology).

**Solution:**

System $->$ Servers: S1, S2, S3, S4, and S5, and no term has been done until now.
Since, there isn't any leader present, and it is given that S1 becomes the leader. Below-mentioned steps are in-line with this case:

**1**. S1 increments its term to A.
**2**. S1's status changes from being a follower to a candidate.
**3**. S1 then votes for itself and sends **RequestVote RPC** parallely to each of the other servers S2, S3, S4, and S5 in the cluster.
**4**. The **RequestVote RPC** includes information about S1's log before sending out the RPC. Here, voters would be the rest of servers, i.e. S2, S3, S4 and S5. If a voter does not have its own log ahead of S1's log, then they would cast their vote for S1's candidature as a leader.
**5**. When S1 receives a majority of votes (say 3 out of 5 servers) for its candidature as a leader, it wins the election and becomes the leader from term A.
**6**. Assuming S1 crashes now.
**7**. Periodic heartbeat message(**AppendEntries RPC**) containing no log entry should be sent from a leader to its followers in order to maintain its authority as a leader. S1 was supposed to do this, but since it crashed, no AppendEntries RPC messages were sent out.
**8**. In view of the fact that followers would not receive any communication from the leader within the window frame of time, i.e. **election timeout**, followers would timeout. Election timeout is specific to each server and is randomized so that not all of them would timeout at the same time and create a situation where a majority is not formed because of vote splitting. Election timeout is chosen randomly from a fixed interval, e.g. 150-300ms.
**9**. In the problem, it is mentioned that S1 recovers immediately, so S1 will repeat the procedure for getting elected again. Assuming S1 gets re-elected.
**10**. Now, S1 will send heartbeat messages, i.e., AppendEntries RPC, to other followers and receive their votes.
**Time Involved:** It would take **broadcast time (BT)** to ascertain leader's authority.
**11**. Now, it's time for S1 to service the client requests. The command to be executed by the replicated state machine is contained in the requests sent out by the client to the cluster.
**12**. S1 creates a new entry for appending the command to its own log, and send an **AppendEntries RPC** in parallel to other follower servers to replicate the entry on their logs. When S1 sends this RPC message, it contains the index and the term of the entry it made in its log preceding the latest entry.
**13**. If the follower server does not find an entry in its log with the same index and the term of the entry sent out by the leader, it would refuse the entry. This means that the follower's log is inconsistent with that of the leader's log.
**14**. This inconsistency needs to be handled by the leader by forcing the follower's logs to duplicate its own log. Conflicting entries in follower's logs must be overwritten with entries from the leader's logs.
**15**. We need to make the followers' log consistent with that of the leader. For that, the follower can refuse and AppendEntries RPC by including the term of the entry conflicting with that of the leader's and sending out the term and the the first index of the entry corresponding to that term. The leader then bypasses all the conflicting entries in that term, and again send AppendEntries RPC for making the logs

consistent. One AppendEntries RPC message is required per term to make the logs consistent.
**Time Involved:** If the number of conflicting terms are x, then it would require **x*BroadcastTime** to make logs consistent. And **one extra BroadcastTime** to append entries contained in the leader's log that are missing in the follower's logs and finally receive a vote.
**16**. When AppendEntries RPC message returns successfully from a majority of the followers, S1 can be sure that the client's entry has been replicated successfully.
Now, S1 responds back to the client with the result of the command's execution. **17**. S1 commits the entry to its local state machine. The highest index of the entry that needs to be committed is stored in a tracked index using which entries are applied to the state machine. This index is also included in the heartbeat messages(AppendEntries RPC) so that other follower servers can also apply the same entry to its local state machine.
**Total Time Taken**:
Assuming client has sent its request after S1 crashes and gets re-elected:
Number of round-trip taken to process a client's request (N) = 1 round-trip (to ascertain leader's authority) + x round-trip (to make a follower's log consistent with that of the leader's) + 1 round-trip (to append missing entries from the follower's logs and get a vote message)
Each round-trip takes BT (Broadcast time) to get completed. This time is calculated for the communication between the leader and a single follower.
If there are y number of followers that are active in the cluster when this communication is happening, it will take **N*y round trips, i.e. N*y*BT time**.

**Test Case Scenarios:**
**1**. Client sends its request to the leader. Leader gets elected. Leader appends the command in its own log. Leader crashes now. Gets re-elected again. Sends messages to follower node to replicate this entry in their log. Receives votes. Applies the transaction to its state machine and notifies the client of the execution.
**Total Time Taken**: 1 BT for a leader to get elected + 1 Leader-Election timeout for other follower nodes timing out in the absence of a leader + 1 BT for the same leader to get re-elected + x*BT (x is the number of terms for which the follower's log is inconsistent with respect to the leader's log) + 1 BT (for appending missing entries from the follower's logs and get a vote message).
BT − > Broadcast time

**2**. Client sends its request to the leader. Leader gets elected. Leader appends the command in its own log. Sends messages to follower node to replicate this entry in their log. Replicas haven't written anything in their log until now. Leader crashes now. Gets re-elected again. Receives votes. Applies the transaction to its state machine and notifies the client of the execution.
**Total Time Taken**: 1 BT for a leader to get elected + x*BT (x is the number of terms for which the follower's log is inconsistent with respect to the leader's log) + 1 Leader-Election timeout for other follower nodes timing out in the absence of a leader + 1 BT for the same leader to get re-elected + 1 BT (for appending missing entries from the follower's logs and get a vote message).

**3**. Client sends its request to the leader. Leader gets elected. Leader appends the command in its own log. Sends messages to follower node to replicate this entry in their log. Replicas haven't written anything in their log until now. Receives votes. Applies the transaction to its state machine and notifies the client of the execution. Leader crashes now. Gets re-elected again.
**Total Time Taken**: 1 BT for a leader to get elected + x*BT (x is the number of terms for which the follower's log is inconsistent with respect to the leader's log) + 1 BT (for appending missing entries from the follower's logs and get a vote message).

## Problem 5

The Practical Byzantine Fault Tolerance (PBFT) paper suggests that the commit phase is needed to ensure that replicas agree on the sequence of executed requests across view changes [Castro and Liskov 2002, Section 4.3]. To understand the need for the commit phase in more detail, consider a variant of PBFT without the commit phase. Specifically, modify the algorithm as follows:

- When replica $i$ has a prepared certificate for a request, it immediately considers the request to be committed at $i$.

Furthermore, the replica does not inform other replicas that it considers the request to be committed, i.e., it does not send commit messages. The rest of the algorithm is unchanged. For example, the following step described in the lecture notes still applies:

- When a request is committed at a replica, and all lower-numbered requests are committed and have been executed, the replica executes the request and sends a reply to the client.

Give a sample execution with $R = 4$ and $f = 1$ in which this algorithm violates the safety requirement (linearizability). Keep it simple. Unnecessarily complicated answers will receive partial credit.

**Solution:**
As per the theory, the "prepare" phase ensures that the replicas have a total order of requests in the same view, but does not guarantee for a total order across view changes since replicas may collect prepared certificates in different views with the same sequence number and different requests. The commit phase ensures consistent ordering of requests across views.

Hence if we need to incorporate a view change in out sample execution to show the loss of safety due to omitting the commit phase.

Considering a scenario, R = 4 and f = 1. Lets say there are replicas : R1, R2, R3, R4 and we see a scenario of view transition from 1 to 2.
Lets say in view 1, a non faulty node R4 gets prepared, is able to prepared certificate and hence commits a request (M1) for a sequence (S1) after receiving quorum (i.e 2f+1 replica declared they received pre-prepare for view 1). The other replicas 1-3 weren't able to receive prepared message and unable to commit (for any reason, let's say network lag, etc. owing to network being partial synchrony).
Now lets say, after sometime, timeout happens and a view change occurs to view 2. Since, in this scenario, no other replica was prepared, the new primary node would not get the quorum for the committed message (M1), and the sequence number S1. Hence the new primary will assign the same sequence number S1 for the newer pre-prepare messages for the view v2, and newer messages M2. Hence a committed request is present in one replica is not present in other replicas, thus it does not survive a view change, thus leading to a divergence in the commit log and inconsistency in the system. Hence the safety is violated in the given scenario.