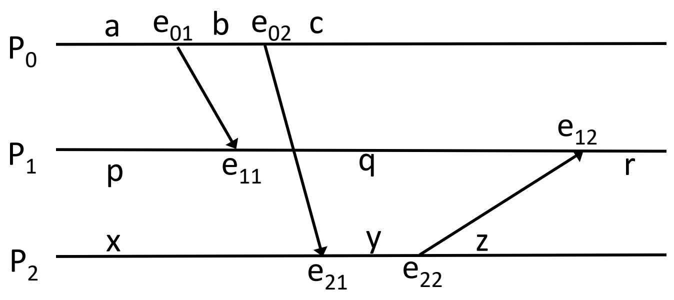
CSE535: Distributed Systems (Fall 2021)
Scott D. Stoller, Stony Brook University
Problem Set 1 (version 2021-10-16), Due 25 October 2021

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 by 11:59pm on the due date. The pdf file should be formatted for 8.5"×11" paper, should include the team name and names of team members at the top of page 1, and the filename should include the team name. 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 will help everyone 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.



Answer: (a) $LT(e_{01}) = (1, 0, 0)$. $LT(e_{02}) = (2, 0, 0)$. $LT(e_{21}) = (2, 0, 1)$. $LT(e_{22}) = (2, 0, 2)$. $LT(e_{11}) = (1, 1, 0)$. $LT(e_{12}) = (2, 2, 2)$. (b) The consistent global states are: (a, p, x), (b, p, x), (b, q, x), (c, p, x), (c, p, y), (c, p, z), (c, q, x), (c, q, y), (c, q, z), (c, r, z)

Problem 2

The reliable broadcast algorithm in the lecture slides does not consider failure detection. 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 pseudocode, your solution is probably unnecessarily complicated.

Answer:

Each process p maintains a matrix M such that M[r][q] is p's knowledge of how many messages from q have been received by r. Each process piggybacks this information on each message it sends. Additions to the original pseudocode are in red. max is computed element-wise on matrices. m.sender is the sender of message m.

```
Initialization at process p:
  rcvd := emptyset; G = the set of group members (including p)
Upon failed(q) at process p:
  remove q from G
  stable = {m for m in rcvd if m.ts[m.sender][m.sender] <= min({M[r][m.sender] for r in G})}
Upon R-multicast(m) at process p:
  M[p][p]++   # p received another message from p
  m.ts=M
  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
    M = max(M,m.ts)   # update M using received information
    M[p][q]++    # p received another message from q
    stable = {m for m in rcvd if m.ts[m.sender][m.sender] <= min({M[r][m.sender] for r in G})}
    for each r in G-{p,q}: send m to r   # relay the message
    R-deliver(m)
```

here is a simpler algorithm that might appear to be a solution: each process counts the number of times that it has received each message m. message m is stable when it has been received $N-1$ times, where N is the number of processes in the group; in other words, m is stable when it has been received from all other processes in the group. this algorithm is deficient, because some messages will never be classified as stable, because a process p does not relay a message m to the process q from which p first received m. this cannot easily be fixed without sending extra messages.

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.

Answer: Extend the recording phase so that, when a process p receives the mark for first time from (say) q, p records that q is its parent, and when p sends the mark to q, it includes a note that q is its parent (otherwise q would not know whether p first received the mark from another node). The edges $\{(p, \text{parent}(p)) \mid p \in P\}$ define a spanning tree rooted at the initiator. In the dissemination phase, each process waits for recorded-state messages from its children (i.e., the neighbors that told the process that it is their parent) and then sends to its parent its own recorded state together with the contents of the recorded-state messages from its children. A process's recorded state consists of its recorded local state and the recorded contents of its incoming channels.

this solution assumes there is one initiator. this assumption is implicit in the statement of the problem, which refers to "the initiator" not "the initiators". on the other hand, Chandy and Lamport's paper allows multiple initiators. If there are multiple initiators, our dissemination algorithm creates a spanning forest, not a spanning tree, and the collected global state is partitioned among the initiators. solutions that consider multiple initiators lose 1 point for not noticing the singular "initiator" in the statement of the problem, but lose no additional points for either leaving the recorded state partitioned among the initiators, or using additional messages (beyond $|P|$) to assemble the entire recorded state at one or more initiators.

Chandy and Lamport sketch a dissemination algorithm that ends with every process having a copy of the global state, if the network is strongly connected (see the last paragraph of section 3). that algorithm is not a solution to this problem, because it sends many more messages.

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).

Answer:

- 1 election timeout ET, to start new election.
- 1 broadcast time BT, for RequestVote and replies.
- 1 broadcast send time BST, to send heartbeat message to announce election result and disseminate

no-op log entry for the new term. Note: BST is less than broadcast time, because don't need to wait for replies.

- receive command c from client
- 1 broadcast time BT, for AppendEntries (containing c) and replies
- send reply to client

Total: ET + BT + BST + BT

Note that it is impossible for the leader to simply resume being leader in the same term as before the crash, without holding an election, because the "state" variable (with possible values Candidate, Leader, Follower) is set to Follower whenever a server starts up (whether for the first time or after a failure), as indicated in Figure 4 of the Raft paper; this implies the "state" variable is in volatile storage.

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.

Answer: The system starts in view 0, with r0 as primary.

1. client c sends request m0. r0, r1, and r2 receive it. r3 does not (the network drops it)
2. r0 sends pre-prepare for m0 to r1, r2, and r3. r1 and r2 receive it. r3 does not.
3. r1 sends prepare to r0, r2, and r3. r0 and r2 receive it. r3 does not.
4. r2 sends prepare to r0, r1, and r3. r0 receives it. r1 and r3 do not.
5. r0 has a pre-prepare and 2 matching prepares for m0, so r0 has a prepared certificate for m0, so it considers m0 to be prepared and (equivalently in this modified protocol) committed, and sends reply to c.
6. r2 has a pre-prepare and 2 matching prepares (its own and from r1) for m0, so r2 has a prepared certificate for m0, so it considers m0 to be prepared and (equivalently in this modified protocol) committed, and sends reply to c.
7. c receives both replies, which together form a reply certificate for m0, so c accepts the result in the replies.
8. r0 becomes partitioned (i.e., the network drops all messages to and from r0). the following description does not bother to mention messages sent to r0.
9. note: among the remaining replicas, only r2 knows that m0 prepared in view 0. however, r2 goes byzantine faulty hereafter and falsely claims that it never heard of m0.
10. r1 sends a view change message to r2 and r3 saying that m0 pre-prepared at r1 in view 0.
11. r2 sends a view change message to r1 and r3 falsely saying that no requests pre-prepared or prepared at r2 in view 0.
12. r3 sends a view change message to r1 and r2 saying that no requests pre-prepared or prepared at r3

in view 0.

13. all view change messages sent to r1, r2, and r3 are received.
14. r1 sends a new-view message establishing view 1. r2 and r3 receive it. r1 does not see a quorum of replicas that prepared m0, so it considers that m0 did not commit in view 0 and does not include m0 as a committed request in its new-view message establishing view 1. specifically, in Figure 4 in [Castro and Liskov 2002], in the conditions for checking whether m0 committed at sequence number n=0 in view v=0, condition A1 is not satisfied, because there are not $2f + 1$ view-change messages m' with an entry for m0 in $m'.\mathcal{P}$.
15. c sends a request m1.
16. r1, r2, and r3 process m1 correctly and send replies to c. c accepts the result in the replies. however, the reply does not reflect the updates performed by m0, and this violates linearizability.

the following scenario might appear to be a simpler solution:

1. r1 is the primary and broadcasts pre-prepare for message m as sequence n in view v.
2. r2, r3, and r4 receive the pre-prepare message and broadcast prepare.
3. Due to some network delay, only r2 receives 2f = 2 matching prepare messages in time and immediately commits message m as sequence n in view v.
4. r1, r3, and r4 do not receive the prepare messages in time and timeout. They initiate a view-change to view v + 1 which is successful.
5. In the new view v + 1, the replicas process message m' as sequence n.
6. r2 has message m processed as sequence n in view v and is now processing message m' as sequence n in view v + 1.

in this scenario, three replicas agree on the sequence of commits, and one replica disagrees. *this behavior is permitted when f=1.* of course, the algorithm guarantees that all replicas agree in failure-free executions. one might hope it guarantees that all replicas agree in executions containing only communication failures (no Byzantine failures). however, the paper does not claim that the algorithm has this property. the behavior exhibited in this scenario satisfies the safety property in the paper, which says, roughly, that clients see correct responses to their requests. in this scenario, the client ignores the reply that r2 sends when it commits m in view v, because the client accepts a result only when it is contained in f+1 matching replies. this scenario can perhaps be extended to become a solution to this problem but is not itself a solution.