15440/15640 Distributed Systems

Homework 2 Solution

Q1. Distributed File Systems (16 Points)

Q1.1 (8 Points)

Answer the following questions with 1~3 sentences.

1. Does check-on-use always have lesser network traffic than broadcast invalidation? If it is, explain the reason, and if not, give a counter-example. (2 Point)

- No, For a server with a small number of clients with frequent read and a workload where writes are rare, broadcast invalidation wouldn't happen that much.

2. Briefly state an advantage of check-on-use over callbacks. (2 Point)

- The server is stateless, so it is easier to handle failures.

3. Briefly state an advantage of callbacks over check-on-use. (2 Point)

- For read-heavy applications, because check-on-use needs to check every time, it can lead to useless network communication.

4. How does NFS handle server failures? (2 Point)

- Because the NFS server is stateless, it could just simply restart. For the missed requests, the clients could simply resend the request, because it is idempotent.

Q1.2 (8 Points)

The TAs have built a distributed file system called Clever File System (CFS) based on their knowledge from 15-440. They plan to use CFS to post homework questions, and students can read and update these files with their answers.

Both clients of TAs and students have a cache, which is empty initially and has a large enough capacity. The cache implements **whole-file caching, session semantics, and check-on-use**.

Suppose only one TA and one student are accessing a shared directory of CFS (both of them have write access to all files). **At first, there is only 1 file named "Q1" in the directory and the file content is: "1+1=?"**.

Consider the TA and the student access the files in the following sequences:

Timestamp	TA	Student
1	fd1 = open(“Q1”, "w")	
2	write(fd1, "1*1=?")	
3	fd2 = open(“Q2”, "w")	
4	write(fd2, "2*2=?")	
5	close(fd2)	
6		fd1 = open(“Q1”, "r")
7		read(fd1)
8		close(fd1)
9		fd2 = open(“Q2”, "w")
10	fd3 = open(“Q3”, "w")	
11	write(fd3, "3*3=?")	
12	close(fd3)	
13		write(fd2, "2*2=4")
14	fd2 = open(“Q2”, "rw")	
15	read(fd2)	
16	write(fd2, "2*3=?")	
17	close(fd2)	
18		close(fd2)
19	close(fd1)	
20	fd2 = open(“Q2”, "rw")	
21	read(fd2)	
22	write(fd2, "Hmmm...")	
23	close(fd2)	
24	delete(“Q2”)	

Assume:

- Open operations with the “w” option will create a new file as well as a local cache copy, if the file does not exist.
- When the file is opened the offset is 0.
- The client’s cache is large enough that eviction never happens.
- Deleting a file also deletes its local cache copy.

Answer the following questions:

1. Identify which timestamp will cause the client to fetch file content from the server. (2 points)

- 1, 6, 9, 20

2. Identify which timestamp will cause the client to send file content to the server. (Use comma separated list) (2 points)
- 5, 12, 17, 18, 19, 23

3. What are the read results on timestamp 15 and 21 of TA, and line 7 of student? (3 points)
- 7: “1+1=?”
- 15: “2*2=?”
- 21: “2*2=4”

4. At the end, what are the file names and contents in TA client cache, student client cache, and the server, respectively? (3 points)
- TA: Q1 (“1*1=?”) Q3(“3*3=?”)
- Student: Q1 (“1+1=?”) Q2 (“2*2=4”)
- Server: Q1 (“1*1=?”) Q3(“3*3=?”)

Q2. Logging, Recovery, and ARIES (20 Points)

In lecture we discussed fault tolerance, using ARIES as a specific example.

Q2.1 (10 Points)

1. Explain why logging is a good practice, versus writing all changes to the corresponding disk pages immediately when they occur. (2 points)

   Disk IO is expensive, and storing the diffs is enough to recover all changes.

2. Suppose a machine is long-running and has already generated lots of log entries. You notice that the logs are taking up a lot of disk space. What is one idea from the lecture that solves this problem? (2 points)

   Checkpointing. Periodically, ensure all changes reflected in logs are applied to the disk pages. Then the logs up to the last commit can be truncated.

3. ARIES has three stages: analysis, redo, and undo. Your friend notices that the redo and undo operations are somewhat inverses of each other, so they suggest ignoring all the

recovered logs and just moving on. Give two reasons why your friend's suggestion is bad. (4 points)

Any two of the following:

- Committed transactions must be applied again and cannot be ignored.
- We don't know which changes are flushed to the disk already, so we must make sure the logs are in sync with the disk (e.g., some logs need to be undone, disk contents need to be rolled back).
- CLRs written during the undo step expedite the recovery process in the event of a second crash.
- (Any other reasonable answer)

4. Explain the purpose of the redoTheUndo field on a Compensation Log Record. (2 points)

   The redoTheUndo field comes into play during the redo phase, where it tells the system how the undo operation performed by the CLR should be 'redone'.

Q2.2 (10 Points)

The following table shows a table of log entries up to LSN 9, after which a crash occurs. We use the following notation for "update" and "commit" type log records:

LSN: [prevLSN, TID, Update, pageID, newVal, oldVal] # for "Update"
LSN: [prevLSN, TID, Commit] # for "Commit"

Notice that after LSN 6 and LSN 8, the logs up to that point are flushed to the disk. Note also that after LSN 3, all logs *as well as* changes to disk page 2 are flushed to the disk. At all other times, nothing is written to disk. Recall also that according to the Write-Ahead Logging Principle, it makes no sense to flush the changes to the disk without also flushing the corresponding logs.

LSN	Contents	Additional Actions
1	[-, 1, Update, P1, A = 10, A = 11]	
2	[-, 2, Update, P1, B = 20, B = 22]	
3	[2, 2, Update, P2, C = 30, C = 33]	Flush log and page 2 to disk
4	[1, 1, Update, P2, D = 40, D = 44]	
5	[4, 1, Update, P1, A = 13, A= 11]	
6	[5, 1, Commit]	Flush log to disk
7	[3, 2, Update, P1, B = 24, B = 20]	

8	[7, 2, Update, P2, C = 36, C = 30]	Flush log to disk
9	[8, 2, Update, P2, D = 42, D = 40]	
	~~CRASH~~	

1. **Analysis Phase**: The system restarts after the crash and performs ARIES. Fill out the contents of the Transaction Table and Dirty Page Table at the end of the analysis phase. Note that when a transaction is committed, it should be removed from the transaction table. (3 points)

    (a) Dirty Page Table - give your answer as tuples of the form (Page Number, Recovery LSN):

    (P1, 1), (P2, 3)

    (b) Transaction Table - give your answer as tuples of the form (Transaction ID, Last LSN):

    (T2, 8)

2. **Redo Phase**: Write down the LSN's (from 1 to 9) of log records that need to be reapplied, and explain why the others can be skipped. Account for all 9 entries above. (4 points)

    Need to be redone: 1, 2, 4, 5, 7, 8
    No need to redo: 3 (PageLSN >= RecoveryLSN), 6 (Nothing to do), 9 (does not exist in the logs after recovery)

3. **Undo Phase**: When writing down compensation log records (CLRs), do so in the following format (replacing *n* with the correct number).

    LSN *n*: [prevLSN, TID, Comp, pageID, redoTheUndo, undoNextLSN] # "Compensation"

    (a) Write down the first two CLR's you will append to the log, using the notation above. Make sure you indicate their LSNs. (2 points)

    LSN 9: [8, 2, Comp, P2, C = 30, 7]
    LSN 10: [9, 2, Comp, P1, B = 20, 3]

    (b) Suppose that right after the two CLR's were written and flushed to disk, a second crash occurs. Upon recovery, what is the *next log entry* you will append during recovery? (1 point)

LSN 11: [10, 2, Comp, P2, C = 33, 2]

Q3. Concurrency Control (18 Points)

In implementing Boogle's distributed database Bpanner, Boogle built a transaction system that implements 2PC. The central server that manages all the transactions is located in Mountain View and functions as the coordinator. The remaining servers across the world function as participants and must all be available to commit to log a transaction in their records. They are currently looking for new full-time software engineers to inspect and develop their coordinator code before the system is put into use. Assume the participant interface is properly working and already designed by another engineer on the team.

Inspect the Go pseudocode below that they are using to model the actual system.

```
01. const (
02.    ABORT = "ABORT"
03.    PREPARE = "PREPARE"
04.    COMMIT = "COMMIT"
05. )
06.
07. type coordinator struct {
08.    participants     map[int]*participant
09.    log              []*transaction
10.    listener         *net.Conn
11.    transferChan     chan *transaction
12.    committedChan    chan bool
13. }
14.
15. func (c *coordinator) main err {
16.    for {
17.       select {
18.       case transfer = <- c.transferChan:
19.          prepared := true
20.          msg := &message{PREPARE, transfer}
21.
22.          for conn, p := range s.participants {
23.             // sends a msg to participants and receive votes
24.             vote, err := p.canCommit(msg, conn)
25.             if err == TIMEOUT {
26.                prepared = false
27.                break
28.             } else if err != nil {
29.                return err
20.             }
31.             prepared = prepared | vote
32.          }
33.
34.          if prepared {
35.             msg := &message{COMMIT, transfer}
36.             log = append(log, transfer)
```

```
37.          } else {
38.              msg := &message{ABORT, transfer}
39.          }
40.
41.          for conn, p := range s.participants {
42.              ack, err := p.doCommit(msg, conn)
43.              if err != nil {
44.                  return err
45.              }
46.          }
47.
48.          c.committedChan <- prepared
49.      }
50.  }
51. }
52.
53. // assume this function is already implemented to take in a message and connID to
54. // return the participants vote
55. func (p *participant) canCommit(m *message, conn int) bool, err {
56.     ...
57. }
58.
59.
60. // assume this function is already implemented to take in a message and connID to
61. // return the participants response
62. func (p *participant) doCommit(m *message, conn int) bool, err {
63.     ...
64. }
65.
66.
67. // whenever a transfer is made, it will be sent to your coordinator server
68. // the coordinator is responsible for contacting the participant servers and unanimously
69. // decide to record the transaction in their logs
70. func (c *coordinator) Transfer(T *transaction) bool {
71.     c.transferChan <- T
72.     return <-c.commitedChan
73. }
```

Q3.1 (2 Points)

You have been invited to a first-round interview at Boogle! As a way to gauge your familiarity with the concepts you will be applying at work, your interviewer asks you to describe the steps to 2PL. You can write pseudocode or describe the process, but try to do so in a brief manner (<3 sentences).

1) Phase 1: Preparation, during which acquire or escalate locks and determine what has to be done, how it will change state, without actually altering it.
2) Phase 2: Commit or abort, during which decide whether to update global state based on if the transaction can be completed. Release all locks in either case.

Q3.2 (2 Points)

Nice, you made it to the final round interview. You are now asked to identify the steps to 2PC and make a note of the range of lines of code in the existing implementation that map to each of them.

1) Prepare and Vote - Participants figure out state changes required and determine if it can complete the transaction, send results to coordinator. (Lines 22-32)

2) Commit - Coordinator broadcasts to either commit or abort the transaction and its changes. (Lines 34-46)

Q3.3 (2 Points)

After successfully passing your final round interview, Boogle officially hires you as a distributed systems engineer. Congrats! Now it is your job to fix the buggy pseudocode implementation of 2PC that we have provided you above. All the previous engineer left as a note was “program seems to still commit when not all the participants are ready”. Identify the line that is causing issues in this code, propose a fix, and explain why this is a necessary change.

Line 31 is an issue, will commit if >= 1 participants vote yes, even if some vote no. Change it to prepared && vote.

Q3.4 (8 Points)

Boogle realizes that their server in Chicago (participant) has a problem crashing from time to time and loses connection with the Mountain View server (coordinator).

1. What should the coordinator do if it does not hear back from a participant? Describe your solution in terms of correctness and performance. (2 points)
    a. Timeout, and assume participant vote was ABORT. Ensures correctness, as we do not know what participants vote was (conservative). Ensures performance, so that we do not block indefinitely.

2. What should the Chicago server do if it crashed after sending “VoteAbort” in response to a PREPARE message from the coordinator server? (2 points)
    a. Safely abort.

3. What should the Chicago server do if it crashed after sending “VoteCommit” in response to a PREPARE message from the coordinator server? (2 points)
    a. Needs to learn whether the server committed or not. Can either query coordinator or other participants via gossip protocol to determine state.

4. Should the participants store tentative changes to disk before or after replying to the PREPARE message? Why? (2 points)
    a. Before replying to PREPARE message. If we store it after, the server could potentially time out immediately after sending a reply to the coordinator, leading to uncertainty upon reboot.

Q3.5 (4 Points)

Considering that the Chicago server has been crashing so often, the higher up executives at Boogle are starting to get worried about what would happen if the Mountain View server (the coordinator) was to crash.

1. What should the Mountain View server do if it crashes during, or immediately after the PREPARE phase? Is there any additional information the server needs on reboot? (2 points)
    a. Resend/initiate prepare phase on all servers it did not receive information from. This may require the server to employ a WAL log strategy in order to know what transaction it was currently working on.

2. What should the Mountain View server do if it crashes during the COMMIT phase (sending doCommit messages)? Is there any additional information the server needs on reboot? (2 points)
    a. Resend decision to any servers which did not receive the message. Optionally, the participants can query the server themselves to get the response, or potentially employ a gossip protocol to communicate with servers that have gotten the response. This may require the server to employ a WAL log strategy in order to know what transaction it was currently working on.

Q4. Distributed Replication (15 Points)

Q4.1 (6 Points)

In lecture we discussed the Basic Paxos algorithm. To test your understanding of the algorithm, let's go through correct and buggy implementations of Paxos.

"I➔J" denotes a successfully delivered message from server I to server J. There are 4 types of messages:

- Prepare(n), where n is a unique sequence (or round) number

- PrepareOK(n, (n_k, a_k)), where nk and a_k are the last sequence number and value, respectively, accepted by Acceptor k (if any) and stored at its stable storage
- Accept(n,v), where v = ak of highest nk seen among the promise responses, or any value if no promise response contained a past accepted proposal
- Accept_OK()

Each of the subquestions shows the sequence of messages from Paxos running on 3 servers (S1, S2, S3). The sequence number represents the order of arrival of the message. Assume that each server replies immediately after receiving a message. The messages not on the sequence are those which failed to deliver.

In each case, state whether the implementation of Paxos is buggy or correct. If buggy, identify the first step in the sequence that is incorrect and explain your answer briefly.

1. (2 points)

   1. S1 ➔ S1: Prepare(101)
   2. S1 ➔ S1: Prepare_OK(101, null)
   3. S1 ➔ S2: Prepare(101)
   4. S2 ➔ S2: Prepare(102)
   5. S2 ➔ S2: Prepare_OK(102, null)
   6. S2 ➔ S3: Prepare(102)
   7. S1 ➔ S3: Prepare(101)
   8. S3 ➔ S2: Prepare_OK(102, null)
   9. S3 ➔ S1: Prepare_OK(101, null)
   10. S2 ➔ S1: Prepare(102)
   11. S1 ➔ S2: Prepare_OK(102, null)

   Buggy, at line 9. Because S3 has received Prepare(102) first, it should ignore the Prepare message from S1 on line 7 and wouldn't be able to arrive at line 9.

2. (2 points)

   1. S1 ➔ S1: Prepare(101)
   2. S1 ➔ S1: Prepare_OK(101, null)
   3. S1 ➔ S2: Prepare(101)
   4. S2 ➔ S1: Prepare_OK(101, null)
   5. S1 ➔ S1: Accept(101, "v1")
   6. S1 ➔ S3: Accept(101, "v1")
   7. S3 ➔ S1: Accept_OK()
   7. S1 ➔ S1: Accept_OK()

   Correct

3. (2 points)

1. S1 ➔ S1: Prepare(101)
2. S1 ➔ S1: Prepare_OK(101, null)
3. S1 ➔ S3: Prepare(101)
4. S3 ➔ S1: Prepare_OK(101, null)
5. S1 ➔ S1: Accept(101, “v1”)
6. S1 ➔ S1: Accept_OK()
7. S2 ➔ S2: Prepare(102)
8. S2 ➔ S2: Prepare_OK(102, null)
9. S2 ➔ S1: Prepare(102)
10. S1 ➔ S2: Prepare_OK(102, null)
11. S1 ➔ S3: Accept(101, “v1”)
12. S3 ➔ S1: Accept_OK()

Buggy, at line 10, S1 should reply with the accepted value (101, “v1”) on the Prepare_OK message.

Q4.2 (9 Points)

1. Explain why the “prepare” phase is needed in Paxos algorithm? (3 Points)

- Helps filter out outdated requests.
- Helps servers find out accepted values in the system proposed by other servers.

2. In Basic Paxos protocol, could an acceptor accept multiple values? Provide a clear explanation of your answer. (3 Points)

- Yes, when the Accept message arrives later has a larger proposal number with a different value. Let’s say proposer S1 sent a Prepare message and received Prepare_OK from the majorities. Then S1 will broadcast an Accept message and assume only one of the acceptors, S5, received and accepted it. During that time, another S2 could also send a Prepare message and could get Prepare_OK from majorities without S5. (So, S2 will not learn the acceptedValue). Later when S5 receives the Accept message from S2, it should accept the new value.
- If acceptors could accept at most one value, the votes could be splitted and eventually no proposer would be able to get majority Accept_OK responses.

3. Three servers from S1 to S3 are using Basic Paxos protocol. The sequence of messages from Paxos running on 3 servers (S1, S2, S3) is like the below. Assume Server S1 was disconnected to all other nodes after timestamp 4, and then reconnected after timestamp 14. During that time Server S2 and S3 have made an agreement with value “v2”. How would S1 figure out what the agreed-upon value was after timestamp 14? (3 Points)

1. S1 ➔ S1: Prepare(101)
2. S1 ➔ S1: Prepare_OK(101, null)

3. S1 ➔ S3: Prepare(101)
4. S3 ➔ S1: Prepare_OK(101, null)
----------(network partition start)----------
5. S2 ➔ S2: Prepare(102)
6. S2 ➔ S2: Prepare_OK(102, null)
7. S2 ➔ S3: Prepare(102)
8. S1 ➔ S1: Accept(101, “v1”)
9. S1 ➔ S1: Accept_OK()
10. S3 ➔ S2: Prepare_OK(102, null)
11. S2 ➔ S2: Accept(102, “v2”)
12. S2 ➔ S2: Accept_OK()
13. S2 ➔ S3: Accept(102, “v2”)
14. S3 ➔ S2: Accept_OK()
----------(network partition end)----------

- S1 will continuously restart a new round by broadcasting a Prepare message to all with an increase in the proposal until it gets a Prepare_OK. It will be able to get a Prepare_OK response with acceptedValue “v1” from both S2 and S3. By looking at that the majority have agreed to the same value, it could figure out the agreed-upon value.