Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 14 Exercise Solutions

14.1 Three computers together provide a replicated service. The manufacturers claim that each computer has a mean time between failure of five days; a failure typically takes four hours to fix. What is the availability of the replicated service?

14.1 Ans.

The probability that an individual computer is down is 4/(5*24 + 4) ~ 0.03. Assuming failure-independence of the machines, the availability is therefore $1 – 0.03^3$ = 0.999973.

14.2 Explain why a multi-threaded server might not qualify as a state machine.

14.2 Ans.

The order in which operations are applied within such a server might differ from the order in which they are initiated. This is because operations could be delayed waiting for some other resource, and the resource scheduling policy could, in principle, reverse the order of two operations.

14.3 In a multi-user game, the players move figures around a common scene. The state of the game is replicated at the players' workstations and at a server, which contains services controlling the game overall, such as collision detection. Updates are multicast to all replicas.

(i) The figures may throw projectiles at one another and a hit debilitates the unfortunate recipient for a limited time. What type of update ordering is required here? Hint: consider the 'throw', 'collide' and 'revive' events.

(ii) The game incorporates magic devices which may be picked up by a player to assist them. What type of ordering should be applied to the pick-up-device operation?

14.3 Ans.

i) The event of the collision between the projectile and the figure, and the event of the player being debilitated (which, we may assume, is represented graphically) should occur in causal order. Moreover, changes in the velocity of the figure and the projectile chasing it should be causally ordered. Assume that the workstation at which the projectile was launched regularly announces the projectile's coordinates, and that the workstation of the player corresponding to the figure regularly announces the figure's coordinates and announces the figure's debilitation. These announcements should be processed in causal order. (The reader may care to think of other ways of organising consistent views at the different workstations.)

ii) If two players move to pick up a piece at more-or-less the same time, only one should succeed and the identity of the successful player should be agreed at all workstations. Therefore total ordering is required.

The most promising architecture for a game such as this is a peer group of game processes, one at each player's workstation. This is the architecture most likely to meet the real-time update propagation requirements; it also is robust against the failure of any one workstation (assuming that at least two players are playing at the same time).

14.4 A router separating process *p* from two others, *q* and *r*, fails immediately after *p* initiates the multicasting of message *m*. If the group communication system is view-synchronous, explain what happens to *p* next.

14.4 Ans.

The case of partitions was excluded from the description of view-synchronous communication in the chapter, but we can describe "reasonable" behaviour for this case. Process *p* must receive a new group view containing only itself, and it must receive the message it sent. The question is: in what order should these events be delivered to *p*?

The answer is that they may be delivered in any order. In view-synchronous communication, a process may conclude that another process received a given message, if that other process is in the view following the delivery of that message. However, there is no converse implication: a process that is absent from the next view may or may not have received a message delivered just before that view.

14.5 You are given a group communication system with a totally ordered multicast operation, and a failure detector. Is it possible to construct view-synchronous group communication from these components alone?

14.5 Ans.

If the multicast is reliable, yes. Then we can solve consensus. In particular, we can decide, for each message, the view of the group to deliver it to. Since both messages and new group views can be totally ordered, the resultant communication will be view-synchronous. If the multicast is unreliable, then we do not have a way of ensuring the consistency of view delivery to all of the processes involved.

14.6 A *sync-ordered* multicast operation is one whose delivery ordering semantics are the same as those for delivering views in a view-synchronous group communication system. In a *thingumajig* service, operations upon thingumajigs are causally ordered. The service supports lists of users able to perform operations on each particular thingumajig. Explain why removing a user from a list should be a sync-ordered operation.

14.6 Ans.

Sync-ordering the remove-user update ensures that all processes handle the same set of operations on a thingumajig before the user is removed. If removal were only causally ordered, there would not be any definite delivery ordering between that operation and any other on the thingumajig. Two processes might receive an operation from that user respectively before and after the user was removed, so that one process would reject the operation and the other would not.

14.7 What is the consistency issue raised by state transfer?

14.7 Ans.

When a process joins a group it acquires state *S* from one or more members of the group. It may then start receiving messages destined for the group, which it processes. The consistency problem consists of ensuring that no update message that is already reflected in the value *S* will be applied to it again; and, conversely, that any update message that is not reflected in *S* will be subsequently received and processed.

14.8 An operation *X* upon an object *o* causes *o* to invoke an operation upon another object o'. It is now proposed to replicate *o* but not o'. Explain the difficulty that this raises concerning invocations upon o', and suggest a solution.

14.8 Ans.

The danger is that all replicas of *o* will issue invocations upon o', when only one should take place. This is incorrect unless the invocation upon o' is idempotent and all replicas issue the same invocation.

One solution is for the replicas of *o* to be provided with smart, replication-aware proxies to o'. The smart proxies run a consensus algorithm to assign a unique identifier to each invocation and to assign one of them to handle the invocation. Only that smart proxy forwards the invocation request; the others wait for it to multicast the response to them, and pass the results back to their replica.

14.9 Explain the difference between linearizability and sequential consistency, and why the latter is more practical to implement, in general.

14.9 Ans.

See pp. 566-567 for the difference. In the absence of clock synchronization of sufficient precision, linearizability can only be achieved by funnelling all requests through a single server – making it a performance bottleneck.

14.10 Explain why allowing backups to process read operations leads to sequentially consistent rather than linearizable executions in a passive replication system.

14.10 Ans.

Due to delays in update propagation, a *read* operation processed at a backup could retrieve results that are older than those at the primary – that is, results that are older than those of an earlier operation requested by another process. So the execution is not linearizable.

The system is sequentially consistent, however: the primary totally orders all updates, and each process sees some consistent interleaving of *reads* between the same series of updates.

14.11 Could the gossip architecture be used for a distributed computer game as described in Exercise 14.3?

14.11 Ans.

As far as ordering is concerned, the answer is 'yes' – gossip supports causal and total ordering. However, gossip introduces essentially arbitrary propagation delays, instead of the best-effort propagation of multicast. Long delays would tend to affect the interactivity of the game.

14.12 In the gossip architecture, why does a replica manager need to keep both a 'replica' timestamp and a 'value' timestamp?

14.12 Ans.

The value timestamp reflects the operations that the replica manager has applied. Replica managers also need to manage operations that they cannot yet apply. In particular, they need to assign identifiers to new operations, and they need to keep track of which updates they have received in gossip messages, whether or not they have applied them yet. The replica timestamp reflects updates that the replica manager has received, whether or not it has applied them all yet.

14.13 In a gossip system, a front end has vector timestamp (3, 5, 7) representing the data it has received from members of a group of three replica managers. The three replica managers have vector timestamps (5, 2, 8), (4, 5, 6) and (4, 5, 8), respectively. Which replica manager(s) could immediately satisfy a query from the front end and what is the resultant time stamp of the front end? Which could incorporate an update from the front end immediately?

14.13 Ans.

The only replica manager that can satisfy a query from this front end is the third, with (value) timestamp (4,5,8). The others have not yet processed at least one update seen by the front end. The resultant time stamp of the front end will be (4,5,8).

Similarly, only the third replica manager could incorporate an update from the front-end immediately.

14.14 Explain why making some replica managers read-only may improve the performance of a gossip system.

14.14 Ans.

First, read operations may be satisfied by local read-only replica managers, while updates are processed by just a few other replica managers. This is an efficient arrangement if on average there are many read operations to every write operation. Second, since read-only replicas do not accept updates, they need no vector timestamp entries. Vector timestamp sizes are therefore reduced.

14.15 Write pseudocode for dependency checks and merge procedures (as used in Bayou) suitable for a simple room-booking application.

14.15 Ans.

```
Operation: room.book(booking).
    let timeSlot = booking.getPreferredTimeSlot();
    Dependency check:
    existingBooking = room.getBooking(timeSlot);
    if (existingBooking != null) return "conflict" else return "no conflict";
    Merge procedure:
    existingBooking = room.getBooking(timeSlot);
    // Choose the booking that should take precedence over the other
    if (greatestPriority(existingBooking, booking) == booking)
        then { room.setBooking(timeSlot, booking); existingBooking.setStatus("failed");}
        else {booking.setStatus("failed");}
```

– in a more sophisticated version of this scheme, bookings have alternative time slots. When a booking cannot be made at the preferred time slot, the merge procedure runs through the alternative time slots and only reports failure if none is available. Similarly, alternative rooms could be tried.

14.16 In the Coda file system, why is it sometimes necessary for users to intervene manually in the process of updating the copies of a file at multiple servers?

14.16 Ans.

Conflicts may be detected between the timestamps of versions of files at a Coda file server and a disconnected workstation when the workstation is reintegrated, Conflicts arise because the versions have diverged, that is, the version on the file server may have been updated by one client and the version on the workstation by another. When such conflicts occur, the version of the file from the workstation is placed in a covolume – an off-line version of the file volume that is awaiting manual processing by a user. The user may either reject the new version, install the new version in preference to the one on the server, or merge the two files using a tool appropriate to the format of the file.

14.17 Devise a scheme for integrating two replicas of a file system directory that underwent separate updates during disconnected operation. Use either Bayou's operational transformation approach, or supply a solution for Coda.

14.17 Ans.

The updates possible on a directory are (a) changing protection settings on existing entries or the directory itself, (b) adding entries and (c) deleting entries.

Many updates may be automatically reconciled, e.g. if two entries with different names were added in different partitions then both are added; if an entry was removed in one partition and not updated in the other then the removal is confirmed; if an entry's permissions were updated in one partition and it was not updated (including deletion) in the other, then the permissions-update is confirmed.

Otherwise, two updates, in partitions A and B, respectively, may conflict in such a way that automatic reconciliation is not possible. e.g. an entry was removed in A and the same entry in B had its permissions changed; entries were created with the same name (but referring to a different file) in A and B; an entry was added in A but in B the directory's write permissions were removed.

We leave the details to the reader.

14.18 Available copies replication is applied to data items *A* and *B* with replicas A_x, A_y and B_m, B_n. The transactions *T* and *U* are defined as:

T: *Read*(*A*); *Write*(*B*, 44). *U*: *Read*(*B*); *Write*(*A*, 55).

Show an interleaving of *T* and *U*, assuming that two-phase locks are applied to the replicas. Explain why locks alone cannot ensure one copy serializability if one of the replicas fails during the progress of *T* and *U*. Explain with reference to this example, how local validation ensures one copy serializability.

14.18 Ans.

An interleaving of T and U at the replicas assuming that two-phase locks are applied to the replicas:

T		U	
x:= Read (Ax)	lock Ax		
Write(Bm, 44)	lock Bm		
		x:= Read (Bm)	Wait
Write(Bn, 44)	lock Bn	•	
Commit unlock	Ax,Bm,Bn	•	
		Write(Ax, 55)	lock Ax
		Write(Ay, 55)	lock Ay

Suppose Bm fails before T locks it, then U will not be delayed. (It will get a lost update). The problem arises because Read can use one of the copies before it fails and then Write can use the other copy. Local validation ensures one copy serializability by checking before it commits that any copy that failed has not yet been recovered. In the case of T, which observed the failure of Bm, Bm should not yet have been recovered, but it has, so T is aborted.

14.19 Gifford's quorum consensus replication is in use at servers *X*, *Y* and *Z* which all hold replicas of data items *A* and *B*. The initial values of all replicas of *A* and *B* are 100 and the votes for *A* and *B* are 1 at each of *X*, *Y* and *Z*. Also *R* = *W* = 2 for both *A* and *B*. A client reads the value of *A* and then writes it to *B*.

(i) At the time the client performs these operations, a partition separates servers *X* and *Y* from server Z.

Describe the quora obtained and the operations that take place if the client can access servers *X* and *Y*.

(ii) Describe the quora obtained and the operations that take place if the client can access only server *Z*.

(iii) The partition is repaired and then another partition occurs so that *X* and *Z* are separated from *Y*. Describe the quora obtained and the operations that take place if the client can access servers X and Z.

14.19 Ans.

i) Partition separates X and Y from Z when all data items have version v0 say:

X	Y	Z
A= 100 (vo)	A= 100(vo)	A= 100(vo)
B= 100(vo)	B= 100(vo)	B= 100(vo)

A client reads the value of A and then writes it to B:

read quorum = 1+1 for A and B - client Reads A from X or Y

write quorum = 1+1 for B client Writes B at X and Y

ii) Client can access only server Z: read quorum = 1, so client cannot read, write quorum = 1 so client cannot write, therefore neither operation takes place.

iii) After the partition is repaired, the values of A and B at Z may be out of date, due to clients having written new values at servers X and Y. e.g. versions v1:

X	Y	Z
A= 200(v1)	A= 200(v1)	A= 100(vo)
B= 300(v1)	B= 300(v1)	B= 100(vo)

Then another partition occurs so that X and Z are separated from Y.

The client *Read* request causes an attempt to obtain a read quorum from X and Z. This notes that the versions (v0) at Z are out of date and then Z gets up-to-date versions of A and B from X.

Now the read quorum = 1+1 and the read operation can be done. Similarly the write operation can be done.