CHAPTER 23

Parallel and Distributed Transaction Processing

Solutions for the Practice Exercises of Chapter 23

Practice Exercises

23.1

Answer:

Data transfer is much faster, and communication latency is much lower on a local-area network (LAN) than on a wide-area network (WAN). Protocols that require multiple rounds of communication maybe acceptable in a local area network, but distributed databases designed for wide-area networks try to minimize the number of such rounds of communication.

Replication to a local node for reducing latency is quite important in a wide-area network, but less so in a local area network.

Network link failure and network partition are also more likely in a wide-area network than in a local area network, where systems can be designed with more redundancy to deal with failures. Protocols designed for wide-area networks should handle such failures without creating any inconsistencies in the database.

23.2

Answer:

a. The types of failure that can occur in a distributed system include
   i. Site failure.
   ii. Disk failure.
   iii. Communication failure, leading to disconnection of one or more sites from the network.

b. The first two failure types can also occur on centralized systems.

23.3

Answer:

A proof that 2PC guarantees atomic commits/aborts in spite of site and link failures follows. The main idea is that after all sites reply with a <**ready** *T*> message, only the coordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a site can happen only after it ascertains the coordinator's decision, either directly from the coordinator or indirectly from some other site. Let us enumerate the cases for a site aborting, and then for a site committing.

a. A site can abort a transaction *T* (by writing an <**abort** *T*> log record) only under the following circumstances:

   i. It has not yet written a <**ready** *T*> log record. In this case, the coordinator could not have got, and will not get, a <**ready** *T*> or <**commit** *T*> message from this site. Therefore, only an abort decision can be made by the coordinator.

   ii. It has written the <**ready** *T*> log record, but on inquiry it found out that some other site has an <**abort** *T*> log record. In this case it is correct for it to abort, because that other site would have ascertained the coordinator's decision (either directly or indirectly) before actually aborting.

   iii. It is itself the coordinator. In this case also no site could have committed, or will commit in the future, because commit decisions can be made only by the coordinator.

b. A site can commit a transaction *T* (by writing a <**commit** *T*> log record) only under the following circumstances:

   i. It has written the <**ready** *T*> log record, and on inquiry it found out that some other site has a <**commit** *T*> log record. In this case it is correct for it to commit, because that other site would have ascertained the coordinator's decision (either directly or indirectly) before actually committing.

   ii. It is itself the coordinator. In this case no other participating site can abort or would have aborted because abort decisions are made only by the coordinator.

23.4

Answer:

Site *A* cannot distinguish between the three cases until communication has resumed with site *B*. The action which it performs while *B* is inaccessible must

be correct irrespective of which of these situations has actually occurred, and it must be such that *B* can re-integrate consistently into the distributed system once communication is restored.

23.5

Answer:

We can have a scheme based on sequence numbers similar to the scheme based on timestamps. We tag each message with a sequence number that is unique for the (sending site, receiving site) pair. The number is increased by 1 for each new message sent from the sending site to the receiving site.

The receiving site stores and acknowledges a received message only if it has received all lower-numbered messages also; the message is stored in the *received-messages* relation.

The sending site retransmits a message until it has received an ack from the receiving site containing the sequence number of the transmitted message or a higher sequence number. Once the acknowledgment is received, it can delete the message from its send queue.

The receiving site discards all messages it receives that have a lower sequence number than the latest stored message from the sending site. The receiving site discards from *received-messages* all but the (number of the) most recent message from each sending site (message can be discarded only after being processed locally).

Note that this scheme requires a fixed (and small) overhead at the receiving site for each sending site, regardless of the number of messages received. In contrast, the timestamp scheme requires extra space for every message. The timestamp scheme would have lower storage overhead if the number of messages received within the timeout interval is small compared to the number of sites, whereas the sequence number scheme would have lower overhead otherwise.

23.6

Answer:

In remote backup systems, all transactions are performed at the primary site and the entire database is replicated at the remote backup site. The remote backup site is kept synchronized with the updates at the primary site by sending all log records. Whenever the primary site fails, the remote backup site takes over processing.

The distributed systems offer greater availability by having multiple copies of the data at different sites, whereas the remote backup systems offer lesser availability at lower cost and execution overhead. Different data items may be replicated at different nodes.

In a distributed system, transaction code can run at all the sites, whereas in a remote backup system it runs only at the primary site. The distributed system

transactions needs to follow two-phase commit or other consensus protocols to keep the data in consistent state, whereas a remote backup system does not follow two-phase commit and avoids related overhead.

23.7

Answer:

Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Suppose the copy of the balance at one of the sites, say s, is not consistent – due to lazy replication strategy – with the primary copy after transaction T_1 is executed, and let transaction T_2 read this copy of the balance. One can see that the balance at the primary site would be \$110 at the end.

23.8

Answer:

Let us say a cycle $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_m \rightarrow T_i$ exists in the graph built by the controller. The edges in the graph will either be local edgem (T_k, T_l) or distributed edges of the form (T_k, T_l, n). Each local edge (T_k, T_l) definitely implies that T_k is waiting for T_l. Since a distributed edge (T_k, T_l, n) is inserted into the graph only if T_k's request has reached T_l and T_l cannot immediately release the lock, T_k is indeed waiting for T_l. Therefore every edge in the cycle indeed represents a transaction waiting for another. For a detailed proof that this implies a deadlock, refer to Stuart et al. [1984].

We now prove the converse implication. As soon as it is discovered that T_k is waiting for T_l:

a. A local edge (T_k, T_l) is added if both are on the same site.

b. The edge (T_k, T_l, n) is added in both the sites, if T_k and T_l are on different sites.

Therefore, if the algorithm were able to collect all the local wait-for graphs at the same instant, it would definitely discover a cycle in the constructed graph, in case there is a circular wait at that instant. If there is a circular wait at the instant when the algorithm began execution, none of the edges participating in that cycle can disappear until the algorithm finishes. Therefore, even though the algorithm cannot collect all the local graphs at the same instant, any cycle which existed just before it started will be detected.

23.9

Answer:

a. The lock can be released only after the update has been recorded at the tail of the chain, since further reads will read the tail. Two phase locking may also have to be respected.

b. The overhead of recording chains per data item would be high. Even more so, in case of failures, chains have to be updated, which would have an even greater overhead if done per item.

c. All nodes in the chain have to agree on the chain membership and order. Consensus can be used to ensure that updates to the chain are done in a fault-tolerant manner. A fault-tolerant coordination service such as ZooKeeper or Chubby could be used to ensure this consensus, by updating metadata that is replicated using consensus; the coordination service hides the details of consensus, and allows storage and update of (a limited amount of) metadata.

23.10

Answer:

a. The old primary may receive read requests and reply to them, serving old data that is missing subsequent updates.

b. Leases can be used so that at the end of the lease, the primary knows that it if it did not successfuly renew the lease, it should stop serving requests. If it is disconnected, it would be unable to renew the lease.

c. This situation would not cause a problem with the majority protocol since the write set (or write quorum) and the read set (read quorum) must have at least one node in common, which would serve the latest value.

23.11

Answer:

a. We can have a special data item at some site on which a lock will have to be obtained before starting a global transaction. The lock should be released after the transaction completes. This ensures the single active global transaction requirement. To reduce dependency on that particular site being up, we can generalize the solution by having an election scheme to choose one of the currently up sites to be the coordinator and requiring that the lock be requested on the data item which resides on the currently elected coordinator.

b. The following schedule involves two sites and four transactions. T_1 and T_2 are local transactions, running at site 1 and site 2 respectively. T_{G1} and T_{G2} are global transactions running at both sites. X_1, Y_1 are data items at site 1, and X_2, Y_2 are at site 2.

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is nonserializable.

23.12

Answer:

a. The same system as in the answer to Exercise 23.14 is assumed, except that now both the global transactions are read-only. Consider the following schedule:

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the systemwide precedence graph must involve at least two different sites and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the

ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the systemwide precedence graph is eliminated.

23.13

Answer:

This is a very bad idea from the viewpoint of throughput. Most transactions would now update a few aggregate records, and updates would get serialized on the lock. The problem that due to Paxos delays plus 2PC delays, commit processing will take a long time (hundreds of milliseconds) and there would be very high contention on the lock. Transaction throughput would decrease to tens of transactions per second, even if transactions do not conflict on any other items.

If the storage system supported operation locking, that could be an alternative to improve concurrency, since view maintenance can be done using operation locks that do not conflict with each other. Transaction throughput would be greatly increased.

Asynchronous view maintenance would avoid the bottleneck and lead to much better throughput, but at the risk of reads of the view seeing stale data.