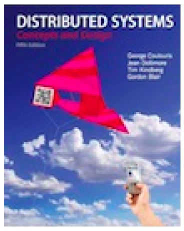

**Distributed Systems: Concepts and Design**

***Edition 5***

**By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair**
**Addison-Wesley ©Pearson Education 2012**

# Chapter 15 Exercise Solutions

15.1 Give an example where an unreliable failure detector produces a suspected value when the system is actually functioning correctly.

*15.1 Ans.*

For example, the process could be functioning correctly, but on the other side of a network partition; or it could be running more slowly than expected, and then the system may produce a suspected value.

15.2 For a system with N processes $p_i$, I = 1, 2, 3 … N, what are the essential requirements for the mutual exclusion conditions ME1 and ME2?

*15.2 Ans.*

The essential requirements for mutual exclusion are as follows:

ME1: (safety) At most one process may execute in the critical section (CS) at a time.

ME2: (liveness) Requests to enter and exit the critical section eventually succeed.

15.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of the synchronization delay.

*15.3 Ans.*

If $s$ = synchronization delay and $m$ = minimum time spent in a critical section by any process, then the maximum throughput is $1/(s+m)$ critical-section-entries per second.

15.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order.

*15.4 Ans.*

Process *A* sends a request $r_A$ for entry then sends a message *m* to *B*. On receipt of *m*, *B* sends request $r_B$ for entry. To satisfy happened-before order, $r_A$ should be granted before $r_B$. However, due to the vagaries of message propagation delay, $r_B$ arrives at the server before $r_A$, and they are serviced in the opposite order.

15.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any client (in any state), assuming that the server is correct and given a reliable failure detector. Comment on whether the resultant system is fault tolerant. What would happen if a client that possesses the token is wrongly suspected to have failed?

*15.5 Ans.*

The server uses the reliable failure detector to determine whether any client has crashed. If the client has been granted the token then the server acts as if the client had returned the token. In case it subsequently receives the token from the client (which may have sent it before crashing), it ignores it.

The resultant system is not fault-tolerant. If a token-holding client crashed then the application-specific data protected by the critical section (whose consistency is at stake) may be in an unknown state at the point when another client starts to access it.

If a client that possesses the token is wrongly suspected to have failed then there is a danger that two processes will be allowed to execute in the critical section concurrently.

---

15.6 Even without a deadlock, a poor algorithm might lead to starvation. Give an example of a system which leads to starvation.

*15.6 Ans.*

Consider a system with one process (*P1*) with minimum priority and all the processes present or coming to the system with higher priority than *P1*. As a result, *P1* will never be able to execute until all the other processes are complete. *P1* will therefore be starved.

---

15.7 In a certain distributed system, each process typically uses mutual exclusion to remove the critical section problem. Different algorithms are available for mutual exclusion. Explain how the central server algorithm is used to achieve mutual exclusion. How does an algorithm using multicast and logical clocks differ from the central server algorithm?

*15.7 Ans.*

In the central server algorithm, to enter a critical section, a process sends a request message to the server and awaits a reply from it. Conceptually, the reply constitutes a token signifying permission to enter the critical section. If no other process has the token at the time of the request, then the server replies immediately, granting the token. If the token is currently held by another process, then the server does not reply but queues the request. On exiting the critical section, a message is sent to the server, giving it back the token.

The basic idea for an algorithm using multicast and logical clocks is that processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.

---

15.8 Maekawa's voting algorithm does not need permission from all peer processes to gain access. Processes need only obtain permission from subsets of their peers to enter, as long as the subsets used by any two processes overlap. Is this algorithm deadlock-prone? Give an example.

*15.8 Ans.*

Consider three processes *p1*, *p2* and *p3* with *V1 = {p1,p2}*, *V2= {p2,p3}*and *V3= {p3p1}*. If the three processes concurrently request entry to the critical section, then it is possible for *p1* to reply to itself and hold off *p2*; for *p2* to reply to itself and hold off *p3*; and for *p3* to reply to itself and hold off *p1*. Each process has received one out of two replies, and none can proceed.

---

15.9 Provide a mechanism to overcome the limitations of Maekawa's voting algorithm.

*15.9 Ans.*

The algorithm can be adapted [Sanders 1987] so that it becomes deadlock-free. In the adapted protocol, processes queue outstanding requests in happened-before order, so that requirement ME3 is also satisfied.

---

15.10 Devise an algorithm to implement an unreliable failure detector.

*15.10 Ans.*

We can implement an unreliable failure detector using the following algorithm. Each process *p* sends a '*p* is here' message to every other process, and it does this every *T* seconds. The failure detector uses an estimate of the maximum message transmission time of *D* seconds. If the local failure detector at process *q* does not receive a '*p* is here' message within seconds of the last one, then it reports to *q* that *p is Suspected*.

However, if it subsequently receives a '*p* is here' message, then it reports to *q* that *p* is *OK*.

---

15.11 How is the fault tolerance of different mutual exclusion algorithms evaluated? Name the type of failure that cannot be tolerated by the ring-based algorithm.

*15.11 Ans.*

The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

a. What happens when messages are lost?

b. What happens when a process crashes?

The ring-based algorithm cannot tolerate a crash failure of any single process.

---

15.12 Explain why reversing the order of the lines '*R-deliver m*' and '*if* ( $q \neq p$ ) *then B-multicast*(*g, m*); *end if*' in Figure 11.10 makes the algorithm no longer satisfy uniform agreement. Does the reliable multicast algorithm based on IP multicast satisfy uniform agreement?

*15.12 Ans.*

Reversing the order of those lines means that a process can deliver a message and then crash before sending it to the other group members – which might, in that case, not receive the message at all. This contradicts the uniform agreement property.

The reliable multicast algorithm based on IP multicast does not satisfy uniform agreement. A recipient delivers a message as soon as it receives it; if the sender was to fail during transmission and that same message was not to have reached the other group members then the uniform agreement property would not be met.

---

15.13 Explain whether the algorithm for reliable multicast over IP multicast works for open as well as closed groups. Given any algorithm for closed groups, how, simply, can we derive an algorithm for open groups?

*15.13 Ans.*

In the case of an open group, senders are not necessarily members of the group. In the limiting case, no member sends messages to the group. In that case, there are no opportunities to piggyback information on top of the individual sequence number that any sender keeps for the group. However, a member will be able to detect a missing IP multicast message when it receives another message from the same sender. The protocol will still operate correctly but with reduced efficiency compared to the case of a closed group: it may take longer to detect a missing message.

In general, an open group algorithm can be obtained by considering the group to be closed and arranging for senders outside that group to pick a member of the group and unicast its message to it. The group member then multicasts it on the sender's behalf.

---

15.14 Explain how to adapt the algorithm for reliable multicast over IP multicast to eliminate the hold-back queue – so that a received message that is not a duplicate can be delivered immediately, but without any ordering guarantees. Hint: use sets of sequence numbers to represent the messages that have been delivered so far.

*15.14 Ans.*

Messages in the hold-back queue serve as markers indicating missing messages. In the original prootocol, missing messages are retrieved and once a message in the hold-back queue has a sequence number 1 greater than the last message R-delivered, it too is R-delivered and removed from the queue. However it is not necessary to hold such messages back. Instead, they can be R-delivered immediately and a record (a set) kept, for each group, of the sequence numbers of all missing messages. Once a missing message has been retrieved, it too is immediately R-delivered and its sequence number removed from the set.

15.15 Consider how to address the impractical assumptions we made in order to meet the validity and agreement properties for the reliable multicast protocol based on IP multicast. Hint: add a rule for deleting retained messages when they have been delivered everywhere, and consider adding a dummy 'heartbeat' message, which is never delivered to the application, but which the protocol sends if the application has no message to send.

*15.15 Ans.*

A process can delete a retained message when it is known to have been received by all group members. The latter condition can be determined from the acknowledgements that group members piggy back onto the messages they send. (This is one of the main purposes of those acknowledgments; the other is that a process may learn sooner that it has missed a message than if it had to wait for the sender of that message to send another one.)

A group member can send periodic heartbeat messages if it has no application-level messages to send. A heartbeat message records the last sequence number sent and sequence numbers received from each sender, enabling receivers to delete message they might otherwise retain, and detect missing messages.

---

15.16 Show that the FIFO-ordered multicast algorithm does not work for overlapping groups, by considering two messages sent from the same source to two overlapping groups, and considering a process in the intersection of those groups. Adapt the protocol to work for this case. Hint: processes should include with their messages the latest sequence numbers of messages sent to *all* groups.

*15.16 Ans.*

Let *p* send a message *m1* with group-specific sequence number 1 to group *g1* and a message *m2* with group-specific sequence number 1 to group *g2*. (The sequence numbers are independent, hence it is possible for two messages to have the same sequence number.) Now consider process *q* in the intersection of *g1* and *g2*. How is *q* to order *m1* and *m2*? It has no information to determine which should be delivered first.

The solution is for the sender *p* to include with its message the latest sequence numbers for each group that it sends to. Thus if *p* sent *m1* before *m2*, *m1* would include <*g1*, 1> and <*g2*, 0> whereas *m2* would include <*g1*, 1> and <*g2*, 1>. Process *q* is in a position to know that *m1* is to be delivered next; it would also know that it had missed a message if it received *m2* first.

---

15.17 Show that, if the basic multicast that we use in the algorithm of Figure 15.13 is also FIFO-ordered, then the resultant totally-ordered multicast is also causally ordered. Is it the case that any multicast that is both FIFO-ordered and totally ordered is thereby causally ordered?

*15.17 Ans.*

We show that causal ordering is achieved for the simplest possible cases of the happened-before relation; the general case follows trivially.

First, suppose *p* TO-multicasts a message *m1* which *q* receives; *q* then TO-multicasts message *m2*. The sequencer must order *m2* after *m1*, so every process will deliver *m1* and *m2* in that order.

Second, suppose *p* TO-multicasts a message *m1* then TO-multicasts message *m2*. Since the basic multicast is FIFO-ordered, the sequencer will receive *m1* and *m2* in that order; so every group member will receive them in that order.

It is clear that the result is generally true, as long as the implementation of total ordering guarantees that the sequence number of any message sent is greater than that of any received by the sending process. See Florin & Toinard [1992].

[Florin & Toinard 1992] Florin, G. and Toinard, C. (1992). A new way to design causally and totally ordered multicast protocols. Operating Systems Review, ACM, Oct. 1992.

15.18 Suggest how to adapt the causally ordered multicast protocol to handle overlapping groups.

*15.18 Ans.*

A process maintains a different vector timestamp *Vg* for each group *g* to which it belongs and attaches all of its vector timestamps when it sends a message.

When a process *p* receives a message destined for group *g* from member *i* of that group, it checks, as in the single-group case, that $Vg(message)[i] = Vg(p)[i] + 1$; also, all other entries in the vector timestamps contained in the message must be less than or equal to *p*'s vector timestamp entries. Process *p* keeps the message on the hold-back queue if this check fails, since it is temporarily missing some messages that happened-before this one.

---

15.19 In discussing Maekawa's mutual exclusion algorithm, we gave an example of three subsets of a set of three processes that could lead to a deadlock. Use these subsets as multicast groups to show how a pairwise total ordering is not necessarily acyclic.

*15.19 Ans.*

The three groups are $G1 = \{p1, p2\}$; $G2 = \{p2, p3\}$; $G3 = \{p1, p3\}$.

A pairwise total ordering could operate as follows: *m1* sent to *G1* is delivered at *p2* before *m2* sent to *G2*; *m2* is delivered to *p3* before *m3* sent to *G3*. But *m3* is delivered to *p1* before *m1*. Therefore we have the cyclic delivery ordering $m1 \to m2 \to m3 \to m1 \ldots$ We would expect from a global total order that a cycle such as this cannot occur.

---

15.20 Construct a solution to reliable, totally ordered multicast in a synchronous system, using a reliable multicast and a solution to the consensus problem.

*15.20 Ans.*

To RTO-multicast (reliable, totally-ordered multicast) a message *m*, a process attaches a totally-ordered, unique identifier to *m* and R-multicasts it.

Each process records the set of message it has R-delivered and the set of messages it has RTO-delivered. Thus it knows which messages have not yet been RTO-delivered.

From time to time it proposes its set of not-yet-RTO-delivered messages as those that should be delivered next. A sequence of runs of the consensus algorithm takes place, where the *k*'th proposals ($k = 1, 2, 3, ...$) of all the processes are collected and a unique decision set of messages is the result.

When a process receives the *k*'th consensus decision, it takes the intersection of the decision value and its set of not-yet-RTO-delivered messages and delivers them in the order of their identifiers, moving them to the record of messages it has RTO-delivered.

In this way, every process delivers messages in the order of the concatenation of the sequence of consensus results. Since the consensus results given to different correct processes are identical, we have a RTO multicast.

---

15.21 A basic multicast primitive guarantees that a correct process will eventually deliver the message, as long as the multicaster does not crash. Show the use of a reliable one-to-one *send* operation to implement B-multicast.

*15.21 Ans.*

A straightforward way to implement B-multicast is to use a reliable one-to-one send operation, as follows:

To B-multicast (g, m): for each process p∈g, send (p, m);

On receive (m) at p: B-deliver (m) at p.

15.22 Consider the algorithm given in Figure 15.17 for consensus in a synchronous system, which uses the following integrity definition: if all processes, whether correct or not, proposed the same value, then any correct process in the decided state would chose that value. Now consider an application in which correct processes may propose different results, e.g., by running different algorithms to decide which action to take in a control system's operation. Suggest an appropriate modification to the integrity definition and thus to the algorithm.

*15.22 Ans.*

A variant of the integrity condition is for any correct process to choose the *majority* of the proposed values (where the function *majority* is defined in the chapter to cover the case where there is no value occurring more often than all the others). Accordingly the algorithm in Figure 15.17 needs to be adapted to choose the *majority*, not the minimum, of the values obtained from $f$+1 rounds.

---

15.23 Show that byzantine agreement can be reached for three generals, with one of them faulty, if the generals digitally sign their messages.

*15.23 Ans.*

Any lieutenant can verify the signature on any message. No lieutenant can forge another signature. The correct lieutenants sign what they each received and send it to one another.

A correct lieutenant decides $x$ if it receives messages [$x$](signed commander) and either [[$x$](signed commander)](signed lieutenant) or a message that either has a spoiled lieutenant signature or a spoiled commander signature.

Otherwise, it decides on a default course of action (retreat, say).

A correct lieutenant either sees the proper commander's signature on two different courses of action (in which case both correct lieutenants decide 'retreat'); or, it sees one good signature direct from the commander and one improper commander signature (in which case it decides on whatever the commander signed to do); or it sees no good commander signature (in which case both correct lieutenants decide 'retreat').

In the middle case, either the commander sent an improperly signed statement to the other lieutenant, or the other lieutenant is faulty and is pretending that it received an improper signature. In the former case, both correct lieutenants will do whatever the (albeit faulty) commander told one of them to do in a signed message. In the latter case, the correct lieutenant does what the correct commander told it to do.