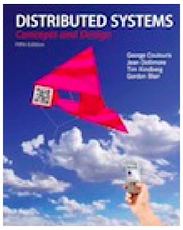

Distributed Systems: Concepts and Design

Chapter 15 Exercise Solutions

15.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel?

15.1 Ans.

An unreliable failure detector can be constructed from an unreliable channel – all that changes from use of a reliable channel is that dropped messages may increase the number of false suspicions of process failure.

A reliable failure detector requires a synchronous system. It cannot be built on an unreliable channel since a dropped message and a failed process cannot be distinguished – unless the unreliability of the channel can be masked while providing a guaranteed upper bound on message delivery times. A channel that dropped messages with some probability but, say, guaranteed that at least one message in a hundred was not dropped could, in principle, be used to create a reliable failure detector.

15.2 If all client processes are single-threaded, is mutual exclusion condition ME3, which specifies entry in happened-before order, relevant?

15.2 Ans.

ME3 is not relevant if the interface to requesting mutual exclusion is synchronous. For a single-threaded process could not send a message to another process while awaiting entry, and ME3 does not arise.

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 Give an example execution of the ring-based algorithm to show that processes are not necessarily granted entry to the critical section in happened-before order.

15.6 Ans.

Consider three processes in a ring, in the order P1, P2 and P3. Note that processes may send messages to one another independently of the token-passing protocol.

```
P1 ---------------> P2

|                   |

|                   |

P3 <----------------|
```

The token is initially with P2. P1 requests the token, then sends a message to P3, which also requests the token. The message passes the token at P2. Then P2 sends on the token. P3 gets it, but the token should have been granted to P1 first.

15.7 In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala's multicast-based mutual exclusion algorithm is inefficient for this case, and describe how to improve its performance. Does your adaptation satisfy liveness condition ME2?

15.7 Ans.

In Ricart and Agrawala's multicast-based mutual exclusion algorithm, a client issues a multicast request every time it requires entry. This is inefficient in the case described, of a client that repeatedly enters the critical section before another needs entry.

Instead, a client that finishes with a critical section and which has received no outstanding requests could mark the token as *JUST_RELEASED*, meaning that it has not conveyed any information to other processes that it has finished with the critical section. If the client attempts to enter the critical section and finds the token to be *JUST_RELEASED*, it can change the state to *HELD* and re-enter the critical section.

To meet liveness condition ME2, a *JUST_RELEASED* token should become *RELEASED* if a request for entry is received.

15.8 In the Bully algorithm, a recovering process starts an election and will become the new coordinator if it has a higher identifier than the current incumbent. Is this a necessary feature of the algorithm?

15.8 Ans.

First note that this is an undesirable feature if there is no advantage to using a higher-numbered process: the re-election is wasteful. However, the numbering of processes may reflect their relative advantage (for example, with higher-numbered processes executing at faster machines). In this case, the advantage may be worth the re-election costs. Re-election costs include the message rounds needed to implement the election; they also may include application-specific state transfer from the old coordinator to the new coordinator.

To avoid a re-election, a recovering process could merely send a *requestStatus* message to successive lower-numbered processes to discover whether another process is already elected, and elect itself only if it receives a negative response. Thereafter, the algorithm can operate as before: if the newly-recovered process discovers the coordinator to have failed, or if it receives an *election* message, it sends a *coordinator* message to the remaining processes.

15.9 Suggest how to adapt the Bully algorithm to deal with temporary network partitions (slow communication) and slow processes.

15.9 Ans.

With the operating assumptions stated in the question, we cannot guarantee to elect a unique process at any time. Instead, we may find it satisfactory to form subgroups of processes that agree on their coordinator, and allow several such subgroups to exist at one time. For example, if a network splits into two then we could form two subgroups, each of which elects the process with the highest identifier among its membership. However, if the partition should heal then the two groups should merge back into a single group with a single coordinator.

The algorithm known as the 'invitation algorithm' achieves this. It elects a single coordinator among each subgroup whose members can communicate, but periodically a coordinator polls other members of the entire set of processes in an attempt to merge with other groups. When another coordinator is found, a coordinator sends it an 'invitation' message to invite it to form a merged group. As in the Bully algorithm, when a process suspects the unreachability or failure of its coordinator it calls an election.

For details see Garcia-Molina [1982].

15.10 Devise a protocol for basic multicast over IP multicast.

15.10 Ans.

We can use the algorithm for reliable multicast, except that processes do not retain copies of the messages that they have delivered (and nor do they piggy back acknowledgements on the messages that they multicast). If the sender is correct, a receiver can fetch a missing message from the sender; but if the sender crashes then a message may be delivered to some members of the group but not others.

15.11 How, if at all, should the definitions of integrity, agreement and validity for reliable multicast change for the case of open groups?

15.11 Ans.

For open groups, the definitions of integrity and agreement do not change. The validity property is not appropriately defined for open groups, since senders are not necessarily members of the group.

For open groups we can define validity as follows: 'If a correct process multicasts message *m*, then some member of *group*(*m*) will eventually deliver *m*.'

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 We gave a solution to consensus from a solution to reliable and totally ordered multicast, which involved selecting the first value to be delivered. Explain from first principles why, in an asynchronous system, we could not instead derive a solution by using a reliable but not totally ordered multicast service and the 'majority' function. (Note that, if we could, then this would contradict the impossibility result of Fischer *et al.*!) Hint: consider slow/failed processes.

15.21 Ans.

If we used a reliable but not totally ordered multicast, the majority function can only be used meaningfully if it is applied to the same set of values. But, in an asynchronous system, we cannot know how long to wait for the set of all values – the source of a missing message might be slow or it might have crashed. Waiting for the first message delivered by a reliable totally ordered multicast finesses that problem.

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.