makes a difference whether the deposit is processed before the interest update or the other way around, which order is followed is not important from a consistency point of view. The important issue is that both copies should be exactly the same. In general, situations such as these require a **total-ordered multicast**, that is, a multicast operation by which all messages are delivered in the same order to each receiver. Lamport's logical clocks can be used to implement total-ordered multicasts in a completely distributed fashion.

Consider a group of processes multicasting messages to each other. Each message is always timestamped with the current (logical) time of its sender. When a message is multicast, it is conceptually also sent to the sender. In addition, we assume that messages from the same sender are received in the order they were sent, and that no messages are lost.

When a process receives a message, it is put into a local queue, ordered according to its timestamp. The receiver multicasts an acknowledgment to the other processes. Note that if we follow Lamport's algorithm for adjusting local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgment. The interesting aspect of this approach is that all processes will eventually have the same copy of the local queue (provided no messages are removed).

A process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process. At that point, the message is removed from the queue and handed over to the application; the associated acknowledgments can simply be removed. Because each process has the same copy of the queue, all messages are delivered in the same order everywhere. In other words, we have established total-ordered multicasting. We leave it as an exercise to the reader to figure out that it is not strictly necessary that each multicast message has been explicitly acknowledged. It is sufficient that a process *reacts* to an incoming message either by returning an acknowledgment or sending its own multicast message.

Total-ordered multicasting is an important vehicle for replicated services where the replicas are kept consistent by letting them execute the same operations in the same order everywhere. As the replicas essentially follow the same transitions in the same finite state machine, it is also known as **state machine replication** [Schneider, 1990].

Note 6.3 (Advanced: Using Lamport clocks to achieve mutual exclusion)
To further illustrate the usage of Lamport's clocks, let us see how we can use the previous algorithm for total-ordered multicasting to establish access to what is commonly known as a **critical section**: a section of code that can be executed by at most one process at a time. This algorithm is very similar to the one for multicasting, as essentially all processes need to agree on the order by which processes are allowed to enter their critical section.

Figure 6.11(a) shows the code that each process executes when requesting, releasing, or allowing access to the critical section (again, leaving out details). Each process maintains a request queue as well as a logical clock. To enter the critical section, a call to `requestToEnter` is made, which results in inserting an `ENTER` message with timestamp `(clock,procID)` into the local queue and sending that message to the other processes. The operation `cleanupQ` essentially sorts the queue. We return to it shortly.

```
1  class Process:
2    def __init__(self, chan):
3      self.queue      = []                    # The request queue
4      self.clock      = 0                     # The current logical clock
5
6    def requestToEnter(self):
7      self.clock = self.clock + 1                         # Increment clock value
8      self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
9      self.cleanupQ()                                     # Sort the queue
10     self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request
11
12   def allowToEnter(self, requester):
13     self.clock = self.clock + 1                         # Increment clock value
14     self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other
15
16   def release(self):
17     tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
18     self.queue = tmp                                    # and copy to new queue
19     self.clock = self.clock + 1                         # Increment clock value
20     self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release
21
22   def allowedToEnter(self):
23     commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
24     return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

Figure 6.11: (a) Using Lamport's logical clocks for mutual exclusion.

When a process P receives an `ENTER` message from process Q, it can simply allow Q to enter its critical section, even if P wants to do so as well. In that case, P's request will have a lower logical timestamp than the `ALLOW` message sent by P to Q, meaning that P's request will have been inserted into Q's queue *before* P's `ALLOW` message.

Finally, when a process leaves its critical section, it calls `release`. It cleans up its local queue by removing all received `ALLOW` messages, leaving only the `ENTER` requests from other processes. It then multicasts a `RELEASE` message.

In order to actually enter a critical section, a process will have to repeatedly call `allowedToEnter` and when returned `False`, will have to block on a next incoming message. The operation `allowedToEnter` does what is to be expected: it checks if the calling process's `ENTER` message is at the head of the queue, and sees if all other processes have sent a message as well. The latter is encoded through the set `commProcs`, which contains the `procID`s of all processes having sent a message by inspecting all messages in the local queue from the second position and onwards.

```
1   def receive(self):
2     msg = self.chan.recvFrom(self.otherProcs)[1]        # Pick up any message
3     self.clock = max(self.clock, msg[0])                # Adjust clock value...
4     self.clock = self.clock + 1                         # ...and increment
5     if msg[2] == ENTER:
6       self.queue.append(msg)                            # Append an ENTER request
7       self.allowToEnter(msg[1])                         # and unconditionally allow
8     elif msg[2] == ALLOW:
9       self.queue.append(msg)                            # Append an ALLOW
10    elif msg[2] == RELEASE:
11      del(self.queue[0])                                # Just remove first message
12    self.cleanupQ()                                     # And sort and cleanup
```

Figure 6.11: (b) Using Lamport's logical clocks for mutual exclusion: handling incoming requests.

What to do when a message is received is shown in Figure 6.11(b). First, the local clock is adjusted according to the rules for Lamport's logical clocks explained above. When receiving an ENTER or ALLOW message, that message is simply inserted into the queue. An entry request is always acknowledged, as we just explained. When a RELEASE message is received, the original ENTER request is removed. Note that this request is at the head of the queue. After that, the queue is cleaned up again.

At this point, note that if we would clean up the queue by only sorting it, we may get into trouble. Suppose that processes P and Q want to enter their respective critical sections at roughly the same time, but that P is allowed to go first based on logical-clock values. P may find Q's request in its queue, along with ENTER or ALLOW messages from other processes. If its own request is at the head of its queue, P will proceed and enter its critical section. However, Q will also send an ALLOW message to P as well, in addition to its original ENTER message. That ALLOW message may arrive *after* P had already entered its critical section, but *before* ENTER messages from other processes. When Q eventually enters, and leaves its critical section, Q's RELEASE message would result in removing Q's original ENTER message, but not the ALLOW message it had previously sent to P. By now, that message is at the head of P's queue, effectively blocking the entrance to the critical section of other processes in P's queue. Cleaning up the queue thus also involves removing old ALLOW messages.

Vector clocks

Lamport's logical clocks lead to a situation where all events in a distributed system are totally ordered with the property that if event a happened before event b, then a will also be positioned in that ordering before b, that is, $C(a) < C(b)$.

However, with Lamport clocks, nothing can be said about the relationship between two events a and b by merely comparing their time values $C(a)$ and $C(b)$, respectively. In other words, if $C(a) < C(b)$, then this does not