Distributed Systems Part II

Solution to Exercise Sheet 1

1 An Asynchronous Riddle

a) The crucial idea is to select one prisoner as a leader. The leader will turn the switch off, whenever he enters the room and the switch is on. All other prisoner will turn the switch on exactly once. So a prisoner who enters the room looks at the switch. If the switch is off and the prisoner has never turned it on before, he will turn the switch on. If the switch is already on or the prisoner already did turn the switch on during an earlier visit, he leaves the switch as it was. The leader counts how many times he turns the switch off. If the leader counted 99 times he can declare "We all visited the switch room at least once". Because he knows, that each of the other 99 prisoners has turned the switch on and he himself has been in the room as well.

b) If the initial position of the switch is unknown, the above protocol cannot be used, since the leader may miscount by one. However, this can easily be fixed. If each prisoner turns the switch on exactly twice, the leader can be sure that everyone visited the room after counting up to $2 \cdot 99 = 198$ turns.

2 Paxos Timeline

a) The timeline consists of five concurrent nodes, and the time progresses from top to bottom. In Figure 1 you can see how both clients propose their values at first, but only the value of client A gets accepted:

- $T_0 + 0.0$: A sends a `ask`(1). As N_1 and N_2 have never stored a value, they reply with `ok`(0, $\perp$).
- $T_0 + 0.5$: B sends a `ask`(2). As N_2 and N_3 have never stored a value, they reply with `ok`(0, $\perp$).
- $T_0 + 1.0$: A sends a `propose`(1,22). This is acknowledged by N_1 with `success` because its $T_{\max} = 1$. N_2 does not reply as its value $T_{\max} = 2$.
- $T_0 + 2.0$: A sends a `ask`(3). As N_2 has never stored a value it replies with `ok`(0, $\perp$). N_1 returns the latest stored value: `ok`(1,22).
- $T_0 + 2.5$: B sends a `propose`(2,33). This is acknowledged by N_3 with `success`. N_2 does not reply as its value $T_{max} = 3$.
- $T_0 + 3.0$: A sends a `propose`(3,22). This is acknowledged by N_1 and N_2 with `success`.
- $T_0 + 4.0$: A sends a `execute`(22), since A now knows that a majority of the servers stores 22. A returns and terminates.

- $T_0 + 4.5$: B sends a `ask`(4). N_3 sends back its latest accepted value `ok`(2,33). N_2 also sends back its latest accepted value `ok`(3,22).
- $T_0 + 6.5$: B sends a `propose`(4,22) (B took the newest value (with the highest ticket number)). Both clients N_2 and N_3 reply with an `success`. All servers have accepted the same value.
- $T_0 + 7.5$: B sends a `execute`(22), since B knows that a majority of the servers store 22 now. B returns and terminates. Now both clients and all servers store the same command.

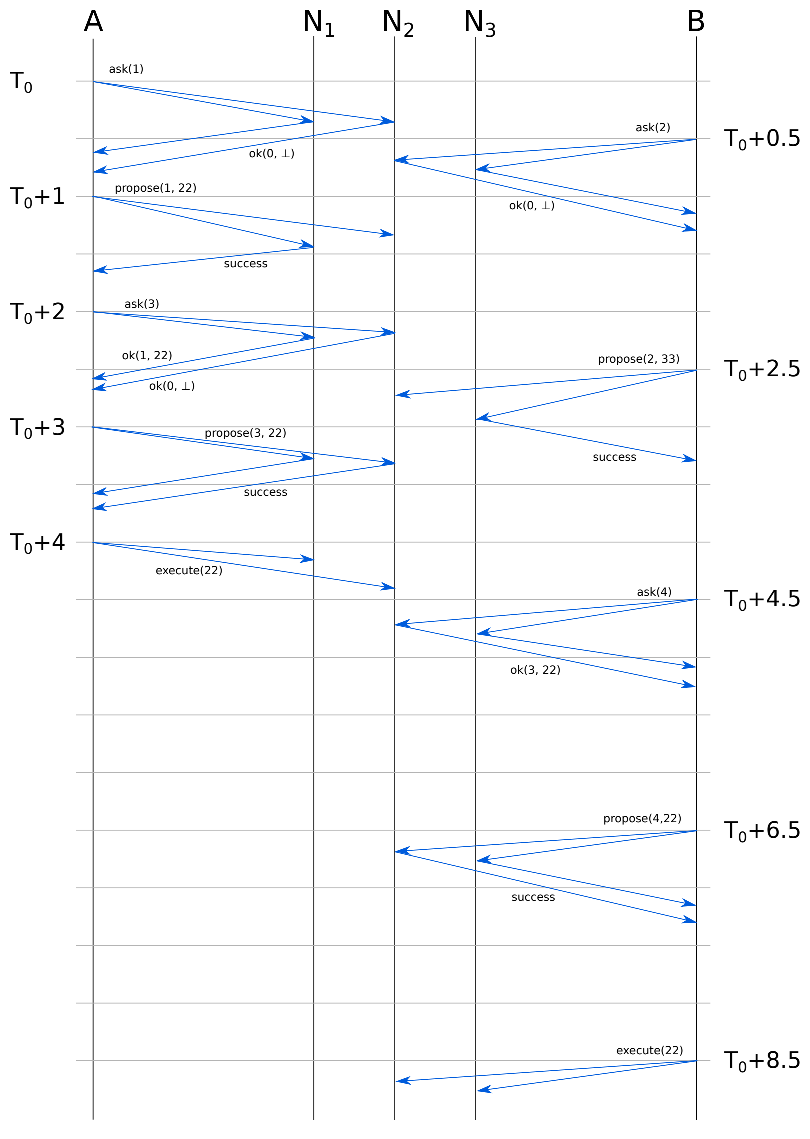

Figure 1: The timeline of the two clients running the given paxos-proposer-program with different timeout values

b) A possible worst-case scenario is when all clients start their attempt to execute a command (approximately) at the same time, use the same timeout and the same initial ticket number.

In that case it can happen that two clients always invalidate each others tickets, and no clients ever succeeds with finding a majority for its proposal messages.

Remark: Of course there can be a lucky schedule, where one client succeeds: For example, if all of its messages `ask`(t) are slow, and then all of its `propose`(t,c) are very fast and immediately get accepted. However, the probability that such an event occurs is rather small, and decreases with the number of servers involved.

3 Improving Paxos

a) Different initial ticket numbers might not be beneficial at all. Let H be the client with the highest initial ticket number. Assume that H asks for a ticket h, and then crashes. In that case, all other clients receive `nack`(h) and will try ticket $h+1$ in the next round. Hence the ticket numbers of all clients will immediately be very close to each other again.

Remark: Different initial ticket numbers can lead to problems even if no machine crashes: For example, it is likely that the client with the highest initial ticket number will always execute its command, and others will experience starvation. In such a system, all users which are using a client with a low initial ticket number will rarely see any progress, and therefore the system as a whole becomes rather useless.

b) We can use an *exponential backoff* approach, as it is used for example in 802.11 wireless networks.

We add a variable b to our code, and possibly a limit $b_{\max}$. Every time an attempt to execute fails, the client doubles the value of b, until $b = b_{\max}$. At the start of every new execution, the client waits for w seconds, where w is chosen uniformly at random from $[0, b]$. After w seconds, it sends the next `ask` message and continues as before.

The modified algorithm is showed below. Changes are on Lines 1-4, 8 and 18. Note that Lines 2-4 are required for that start, when $b = 0$. (Without those lines, $b = 2b$ would not increase the backoff time.)

Analysis

Assume that the first call of **execute** is with a backoff time $b = 0$. Hence, if there is only a single client trying to execute a command, it will be immediately executed, i.e., there is no disadvantage by applying the backoff approach.

Assume that multiple clients try to execute a command. Recall that the time required for a successful **execute** is 2δ. Hence, as soon as $b > 2\delta$, the probability that two clients interfere with each other diminishes rapidly.

Algorithm 1 Paxos proposer algorithm with timeouts and backoff

/* Execute a command on the Paxos servers.
 *
 * N, N': The Paxos servers to contact.
 * c: The command to exexcute.
 * δ: The timeout between multiple attempts.
 * t: The first ticket number to try.
 * b: The backoff time to wait.
 *
 * Returns: c', the command that was executed on the servers. Note that c' might be
 * another command than c, if another client already successfully executed a command.
 */

execute(`Node` N, `Node` N', `command` c, `Timeout` δ, `TicketNumber` t, `BackoffTime` b) {

Wait for **rand(**0,b**)** seconds

if $b = 0$ **then**
  $b = b_{\min}$ ◃ Set b to a value larger than 0, such that the doubling can start
end if

Phase 1 ....

Ask N, N' for ticket t

Phase 2 ....

Wait for δ seconds

if within these δ seconds, either N or N' has not replied with `ok` **then**
  return **execute**(N, N', c, δ, $t+2$, $\min(2b, b_{\max})$)
else
  Pick (T_{store}, C) with largest T_{store}
  if $T_{\text{store}} > 0$ **then**
    $c = C$
  end if
  Send `propose`(t, c) to N, N'
end if

Phase 3 ....

Wait for δ seconds

if within these δ seconds, either N or N' has not replied with `success` **then**
  return **execute**(N, N', c, δ, $t+2$, $\min(2b, b_{\max})$)
else
  Send `execute`(c) to every server
  return c
end if