Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Autumn Term 2020 Prof. T. Roscoe, Prof. R. Wattenhofer

Computer Systems

— Solution to Assignment 7 —

1 Paxos

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.

1.2 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. Notice that A has a 1–second–timeout and B has a 2–second–timeout, and both clients increase their internal ticket counter t by 2 every time they ask for a ticket. The protocol shows the following:

- $T_0 + 0.0$: 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,a). 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,a).
- $T_0 + 2.5$: B sends a `propose`(2,b). 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,a). This is acknowledged by N_1 and N_2 with `success`.

- $T_0 + 4.0$: A sends a `execute`(a), since A now knows that a majority of the servers stores a. A returns and terminates.
- $T_0 + 4.5$: B sends a `ask`(4). N_3 sends back its latest accepted value `ok`(2,b). N_2 also sends back its latest accepted value `ok`(3,a).
- $T_0 + 6.5$: B sends a `propose`(4,a) (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.
- $T_0 + 8.5$: B sends a `execute`(a), since B knows that a majority of the servers store a now. B returns and terminates. Now both clients and all servers store the same command.

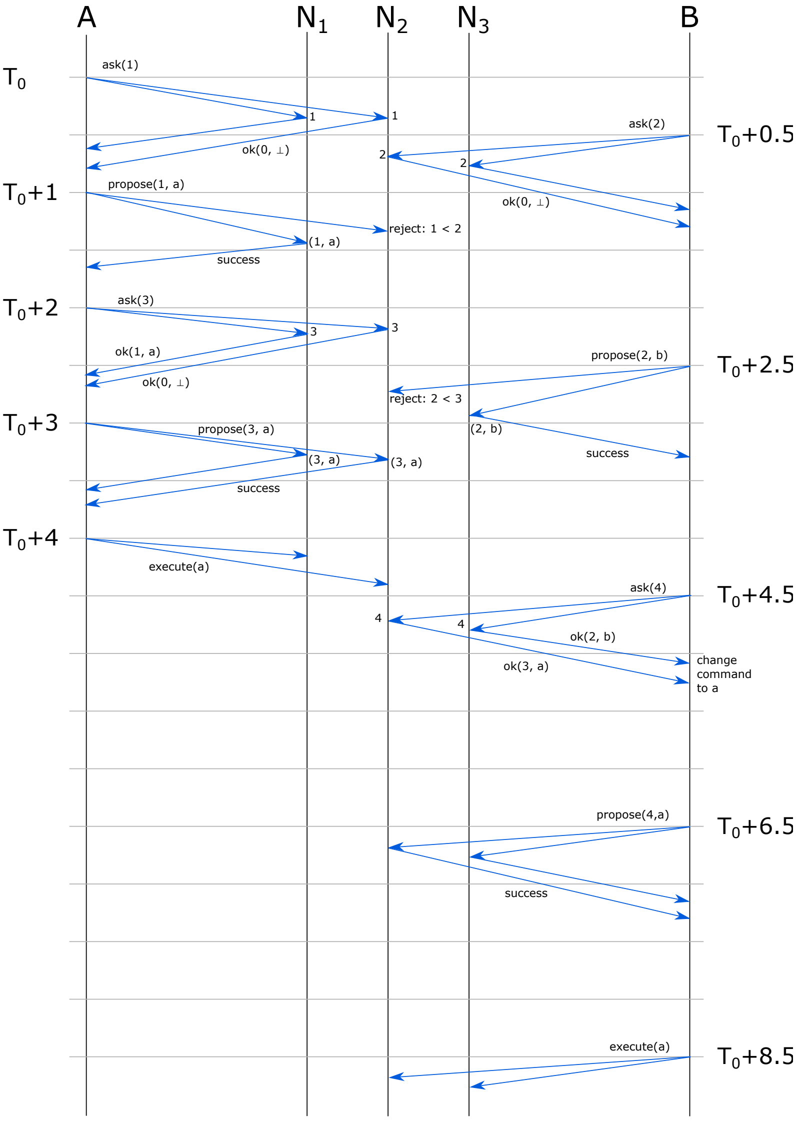

Figure 1: The timeline of the two clients running the given paxos-proposer-program with different timeout values. The values $T_{\mathtt{max}}$ and the tuples $(T_{\mathtt{store}}, C)$ per server are denoted next to each server's line whenever they change.

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 (approximately) the same initial ticket number.

In that case it can happen that two clients always invalidate each others tickets, and no client 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.

1.3 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 (not to be confused with the command that B is supposed to send in exercise 2a) 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 shown below. Changes are on Lines 1-4, 8 and 18. Note that Lines 2-4 are required for the start when $b = 0$. (Without those lines, $b = 2b$ would not increase the backoff time.)

Analysis

Assume that the first call of **suggestValue** 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 run of **suggestValue** 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.
 * $\delta$: 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.
 */
suggestValue(Node $N$, Node $N'$, command $c$, Timeout $\delta$, TicketNumber $t$, BackoffTime $b$) {
Wait for rand(0,$b$) seconds
if $b = 0$ then
    $b = b_{\text{min}}$        ◁ 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 $\delta$ seconds
if within these $\delta$ seconds, either $N$ or $N'$ has not replied with ok then
    return suggestValue($N$, $N'$, $c$, $\delta$, $t + 2$, min($2b$,$b_{\text{max}}$))
else
    Pick ($T_{\text{store}}$, $C$) with largest $T_{\text{store}}$
    if $T_{\text{store}} > 0$ then
        $c = C$
    end if
    Send propose($t$, $c$) to $N, N'$
end if
Phase 3 ..................................................................................
Wait for $\delta$ seconds
if within these $\delta$ seconds, either $N$ or $N'$ has not replied with success then
    return suggestValue($N$, $N'$, $c$, $\delta$, $t + 2$, min($2b$,$b_{\text{max}}$))
else
    Send execute($c$) to every server
    return $c$
end if
```

2 Consensus

2.1 Consensus with Edge Failures

a) $f = n - 1$. Remove all edges from a node – this node will not receive any messages and cannot decide.

Proof why $n - 2$ is too small to prevent consensus:

Note that if $f < n - 1$ the remaining network is always connected. To prove that, assume by contradiction, that you can split the network into two partitions by only removing $n-2$ many edges.

In that case there are two groups of nodes K and L which do not have a remaining edge between them: one containing k many nodes, and one containing l many nodes. Note that $k \geq 1, l \geq 1$ and $k+l=n$.

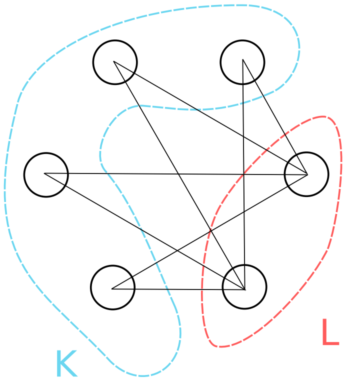

Figure 2: $n=6$ nodes partitioned into two groups K and L. Since they should be disconnected, all edges between the two sets have to fail. (Edges within the groups are not drawn to avoid cluttering up the drawing.)

Since the network is initially fully connected, there are initially $k \cdot l$ many edges between the two groups. As we assumed that the two groups are disconnected, all of these edges have failed, thus $k \cdot l \leq n-2$.

From $k+l=n$ follows $k=n-l$. Plugging that into the inequality we know that

$$k \cdot l = (n-l) \cdot l = n \cdot l - l^2 \leq n-2$$

or

$$-l^2 + nl - n + 2 \leq 0$$

Since n is fixed, we need to see if there is an l which satisfies this inequality, subject to $l \in [1, n-1]$. The first derivative of the inequality $(-2l+n)$ shows that there is only one extreme point $(l = n/2)$, and it is a local maximum (compare, e.g., the second derivative). Since the function is continuous, it follows that it is minimal on the border cases for l, i.e., $l=1$ or $l=n-1$. Plugging in these values for l we see:

$$-1 + n - n + 2 \leq 0$$

$$-(n-1)^2 + n \cdot (n-1) - n + 2 = -n^2 + 2n - 1 + n^2 - n - n + 2 = 1 \leq 0$$

Which both do not hold. Hence the inequality is unsatisfiable, and the split into two partitions is not possible[1]. □

We showed that with only $f = n-2$ many failures, the network always stays connected. Thus, all nodes can – by routing messages via other nodes – always learn all initial values, and then decide.

b) $f = \frac{n(n-1)}{2} - (n-1) = \frac{n^2-3n+2}{2}$. Remove all edges $(\frac{n(n-1)}{2})$ except a path which connects all nodes $(n-1)$.

[1] What did we do in this proof? We showed that there is no cut of size $n-2$ in the fully connected graph with n nodes.

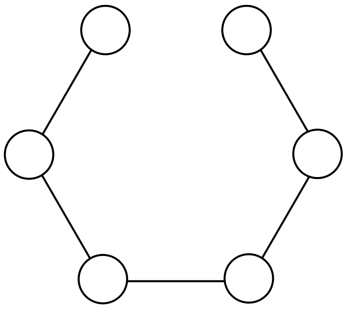

Figure 3: Smallest remaining connected network with $n = 6$.

c) Up to $n - 1$ time units.

Observe that the time to terminate is equal to the longest shortest path between two nodes in the network. This path contains at most n nodes, since there are only n nodes in the network, and a shortest path can never contain a loop. In **b)** we showed that such a scenario can indeed happen, where we have two nodes in distance $n-1$. Hence, the algorithm requires up to $n - 1$ time units.

2.2 Deterministic Random Conensus?!

We partition the nodes into two groups A and B. All nodes in A start with initial value 0, all nodes in B with 1. Let A contain $\lceil n/2 \rceil + 1$ many nodes, and all other nodes are in B.

We show that – only with a bad scheduling, we do not even need any crash failures – it is possible that the system remains in the identical distribution as before; and since there is no randomness anymore, we can therefore deduce that the system will not reach consensus.

1) In the Propose phase only one node $u \in A$ receives messages only from A. All other nodes receive messages from both A and B. Hence, u proposes 0, and all other nodes propose $\perp$.

2) In the Adapt phase all nodes from A receive the proposal of u, and all nodes from B do not receive the proposal of u. Thus, all nodes in A set $v = 0$, and all nodes in B set $v = 1$. This is exactly the same configuration as initially assumed!

Since our goal is to solve consensus in the worst-case scenario (allowing for a worst-case scheduling), this scheduling might happen in every round and the algorithm does not terminate.

2.3 Consensus with Bandwidth Limitations

a) Note that we assumed that no nodes or edges crash. Hence, we can use a designated leader, that just decides for its own value, and the only question is how fast the leader can distribute its value to all nodes.

For readability we call one time unit a *round.*

It is obvious that the leader can distribute its value one by one to every other node, requiring $n - 1$ rounds. But can we do better?

Yes: By having a carefully selected forwarding mechanism. Let u_1 be the leader. In the first round, u_1 sends its value to u_2. In the second round u_1 sends the value to u_3, and u_2 to u_4. This pattern is easily generalized: Every node which receives the value in round i will send it to nodes in all rounds $i + 1, i + 2, \ldots$. Note that with this approach the number of nodes which know the value doubles in every round.

Since the number of nodes which know the value of the leader doubles in every round, it follows that the algorithm requires only $\log(n)$ many rounds.

b) It requires $\log(n)$ time units.

c) The task is that n nodes learn $n-1$ values. Since we assumed that each message can only contain one value, it follows that every node needs to receive at least $n-1$ many messages. Thus, there are in total at least $n \cdot (n-1)$ many received messages. Of course the number of received messages is equal to the number of sent messages (assuming no message loss). Observe that in any round every node can send at most one message, thus, the number of sent messages in a round is at most n. Therefore, any algorithm which requires $n \cdot (n-1)$ many sent messages requires at least $n-1$ many rounds.

Note: Notice that a node may *receive* more than one message per round. It is necessary to argue with the number of messages *sent*.