ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Autumn Term 2018 Prof. T. Roscoe, Prof. R. Wattenhofer

Computer Systems

Assignment 3

Assigned on: **October 12, 2018**

We categorize questions into four different categories:

Quiz Short questions which we will solve rather interactively at the start of the exercise sessions.

Basic Improve the basic understanding of the lecture material.

Advanced Test your ability to work with the lecture content. This is the typical style of questions which appear in the exam.

Mastery Beyond the essentials, more interesting, but also more challenging. These questions are **optional**, and we do not expect you to solve such exercises during the exam.

1 Paxos

Quiz

1.1 An Asynchronous Riddle

A hangman summons his 100 prisoners, announcing that they may meet to plan a strategy, but will then be put in isolated cells, with no communication. He explains that he has set up a switch room that contains a single switch. Also, the switch is not connected to anything, but a prisoner entering the room may see whether the switch is on or off (because the switch is up or down). Every once in a while the hangman will let one arbitrary prisoner into the switch room. The prisoner may throw the switch (on to off, or vice versa), or leave the switch unchanged. Nobody but the prisoners will ever enter the switch room. The hangman promises to let any prisoner enter the room from time to time, arbitrarily often. That is, eventually, each prisoner has been in the room at least once, twice, a thousand times or any number you want. At any time, any prisoner may declare "We have all visited the switch room at least once". If the claim is correct, all prisoners will be released. If the claim is wrong, the hangman will execute his job (on all the prisoners). Which strategy would you choose...

a) ...if the hangman tells them, that the switch is off at the beginning?

b) ...if they don't know anything about the initial state of the switch?

Basic

1.2 Paxos

You decide to use Paxos for a system with 3 servers (acceptors), which we call N_1, N_2, N_3. There are two clients (proposers) A and B. The implementation of the acceptors is exactly as defined in the script, see Algorithm 7.13. We extended the code of the proposers, such that they now use explicit timeouts. The algorithm is described below, note in particular Lines 2-4 and 12-14.

Algorithm 1 Paxos proposer algorithm with timeouts

```
/* 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.
 *
 * 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 δ, TicketNumber t) {
```

Phase 1

1: Ask N, N' for ticket t

Phase 2

2: Wait for δ seconds

3: **if** within these δ seconds, either N or N' has not replied with `ok` **then**
4: **return** **suggestValue**(N, N', c, δ, $t+2$)
5: **else**
6: Pick (T_{store}, C) with largest T_{store}
7: **if** $T_{\text{store}} > 0$ **then**
8: $c = C$
9: **end if**
10: Send `propose`(t, c) to N, N'
11: **end if**

Phase 3

12: Wait for δ seconds

13: **if** within these δ seconds, either N or N' has not replied with `success` **then**
14: **return** **suggestValue**(N, N', c, δ, $t+2$)
15: **else**
16: Send `execute`(c) to every server
17: **return** c
18: **end if**

a) Assume that two users try to execute a command. One user calls **suggestValue**(N_1, N_2, a, 1, 1) on A at time T_0, and a second user calls **suggestValue**(N_2, N_3, b, 2, 2) on B at time $T_0 + 0.5sec$.

Draw a timeline containing all transmitted messages! We assume that processing times on nodes can be neglected (i.e. is zero), and that all messages arrive within less than $0.5sec$.

b) In **a)** we chose artificial initial ticket numbers and timeout values, and we saw that Paxos terminates rather quickly.

Let us look at another selection of these values: The two clients start with initial ticket numbers $t_A = t_0$ and $t_B = t_0 + 1$ for some value t_0, and both clients have the same timeout $\delta_A = \delta_B$. Assume that both clients start at T_0. What will happen?

Advanced

1.3 Improving Paxos

We are not happy with the runtime of the Paxos algorithm of Exercise 1.2. Hence, we study some approaches which might improve the runtime.

The point in time when clients start sending messages cannot be controlled, since this will be determined by the application that uses Paxos. It might help to use different initial ticket numbers. However, if a client with a very high ticket number crashes early, all other clients need to iterate through all ticket numbers. This problem can easily be fixed: Every time a client sends an `ask`(t) message with $t \leq T_{\max}$, the server can reply with an explicit `nack`($T_{\max}$) in Phase 1, instead of just ignoring the `ask`(t) message.

a) Assume you added the explicit `nack` message. Do different initial ticket numbers solve runtime issues of Paxos, or can you think of a scenario which is still slow?

b) Instead of changing the parameters, we add a waiting time between sending two consecutive `ask` messages. Sketch an idea of how you could improve the expected runtime in a scenario where multiple clients are trying to execute a command by manipulating this waiting time!

Extra challenge: Try not to slow down an individual client if it is alone!

2 Consensus

Quiz

2.1 Consensus with Edge Failures

In the lecture we only discussed node failures, but we always assumed that edges (links) never fail. Let us now study the opposite case: Assume that all nodes work correctly, but up to f edges may fail.

Analogously to node failures, edges may fail at any point during the execution. We say that a failed edge does not forward any message anymore, and remains failed until the algorithm terminates. Assume that an edge always simultaneously fails completely, i.e., no message can be exchanged over that edge anymore in either direction.

We assume that the network is initially fully connected, i.e., there is an edge between every pair of nodes. Our goal is to solve consensus in such a way, that *all* nodes know the decision.

a) What is the smallest f such that consensus might become impossible? (Which edges fail in the worst-case)

b) What is the largest f such that consensus might still be possible? (Which edges fail in the best-case)

c) Assume that you have a setup which guarantees you that the nodes always remain connected, but possibly many edges might fail. A very simple algorithm for consensus is the following: Every node learns the initial value of all nodes, and then decides locally. How much time might this algorithm require?

Assume that a message takes at most 1 time unit from one node to a direct neighbor.

Basic

2.2 Deterministic Random Consensus?!

Algorithm 8.15 from the lecture notes solves consensus in the asynchronous time model. It seems that this algorithm would be faster, if nodes picked a value deterministically instead of randomly in Line 23. However, a remark in the lecture notes claims that such a deterministic selection of a value will not work. We did it anyway! (See algorithm below, the only change is on Line 23).

Show that this algorithm does not solve consensus! Start by choosing initial values for all nodes and show that the algorithm below does not terminate.

Algorithm 2 Randomized Consensus (Ben-Or)

```
1:  v_i ∈ {0,1}                    ◁ input bit
2:  round = 1
3:  decided = false
4:  Broadcast myValue(v_i, round)
5:  while true do
        Propose
6:      Wait until a majority of myValue messages of current round arrived
7:      if all messages contain the same value v then
8:          Broadcast propose(v, round)
9:      else
10:         Broadcast propose(⊥, round)
11:     end if
12:     if decided then
13:         Broadcast myValue(v_i, round+1)
14:         Decide for v_i and terminate
15:     end if
        Adapt
16:     Wait until a majority of propose messages of current round arrived
17:     if all messages propose the same value v then
18:         v_i = v
19:         decided = true
20:     else if there is at least one proposal for v then
21:         v_i = v
22:     else
23:         Choose v_i = 1
24:     end if
25:     round = round + 1
26:     Broadcast myValue(v_i, round)
27: end while
```

Advanced

2.3 Consensus with Bandwidth Limitations

Consensus with no failures, a fully connected network and unlimited bandwidth is trivial: First, every node sends its value to all other nodes. Second, every node waits for all values, and then decides.

So far we only studied failures. However, in practice bandwidth limitations are often of great importance as well. To simplify the problem, we assume no node crashes and no edge crashes in this exercise. Additionally, you can assume that all nodes have unique ids from 1 to n.

We assume that all messages are transmitted reliably, and arrive exactly after one time unit. The bandwidth limitation is as follows: Assume that every node can only send *one* message (containing one value) to *one* neighbor per time unit. E.g., at time 0, u_1 can send a message to u_2, at time 1 a message to u_3, and so on. However, u_1 cannot send a message to both u_2 and u_3 at the same time! Also, a node cannot send multiple values in the same message.

a) Develop an algorithm that solves consensus in this scenario. Optimize your algorithm for runtime!

b) What is the runtime of your algorithm?

c) Assume that you not only need to solve consensus, but the more challenging task that every node must learn the input values of all nodes. Show that this problem requires at least $n - 1$ time units!