ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Autumn Term 2019 Prof. R. Wattenhofer

Distributed Systems
Exam

Saturday, 25th of January 2020, 09:00 - 10:30

Do not open or turn before the exam starts!
Read the following instructions!

The exam takes 90 minutes and there is a total of 90 points. The maximum number of points for each subtask is indicated in brackets. **Justify all your answers** unless the task explicitly states otherwise. Mark drawings precisely.

Answers which we cannot read are not awarded any points!

At the beginning, fill in your name and student number in the corresponding fields below. You should fill in your answers in the spaces provided on the exam. If you need more space, we will provide extra paper for this. Please label each extra sheet with your name and student number.

Family Name	**First Name**	**Student Number**

Task	**Achieved Points**	**Maximum Points**
1 - Synchronous Reliable Broadcast		28
2 - Quorum Systems		18
3 - Caching in a Directed Network		20
4 - Bitcoin		24
Total		**90**

1 Synchronous Reliable Broadcast (28 points)

In this task we investigate byzantine agreement based on synchronous reliable broadcast. All nodes communicate in synchronous rounds. First, the nodes broadcast their input values 0 or 1 reliably (Line 1 to Line 12 of Algorithm 1). Then, the nodes decide on the agreement value.

Algorithm 1 Synchronous Reliable Broadcast + Agreement (code for node u)

```
1: Broadcast own input bit msg(u)
2: for all received msg(v) do
3:     Broadcast echo(u, msg(v))
4: end for
5: for all echo(w, msg(v)) received from at least n − 2f nodes w do
6:     if not echoed msg(v) before then
7:         Broadcast echo(u, msg(v))
8:     end if
9: end for
10: for all echo(w, msg(v)) received from at least n − f nodes w do
11:     Accept msg(v)
12: end for
13: Decide on the majority of all accepted values
```

Note that all messages in the **for all** loop can be broadcast in parallel. Moreover, byzantine messages which contradict the protocol will be ignored by the nodes locally. Assume there is only one Byzantine node ($f = 1$) and $n > 3f$. Show that Algorithm 1 satisfies ...

a) [3] termination

b) [8] agreement

c) [3] all-same-validity

From now on we assume the general case where $f > 1$ and $n > 3f$:

d) [4] Does Algorithm 1 solve byzantine agreement for large f, e.g. $f > 5$?

e) [10] Does Algorithm 1 solve byzantine agreement for $f = 2$ and $n = 7$?

a) Algorithm 1 satisfies termination, because all nodes terminate with a decision after three communication rounds.

b) The analysis is similar to Algorithm 17.9 from the lecture, where the nodes can agree on any input value. First observe that all correct nodes will accept all correct input values: due to synchrony, all correct values will arrive at all correct nodes after the first communication round, and, therefore, all correct nodes will echo this value; all correct nodes will get $n - f$ echos and accept the values.

Assume now that the byzantine node broadcasts a value. Each correct node will echo the respective value in the next round. Observe that the byzantine node can only echo the same value as it sent to the respective node, otherwise the correct node does not have to consider this value. Assume now that a correct node has accepted a byzantine value, e.g. 0, then it has received at least $n - 2f$ echos for 0 from correct nodes in the previous two rounds. Therefore, all other correct nodes will have received $n - 2f$ echos for 0 from these correct nodes as well, and will have accepted the byzantine value too.

c) Assume that all correct nodes have the same input value, e.g. 1. All nodes will accept the value 1 from at least $n - f$ nodes. Since the values are binary, the byzantine party can only make sure that the correct nodes accept the opposite value 0 exactly once. Since $n - f > 2f$

d) In the general case, agreement is not satisfied. If it was, Algorithm 1 would solve byzantine agreement in three rounds. For $f > 5$ this is a contradiction to the lower bound of $f + 1$ rounds of Section 17.4 in the lecture.

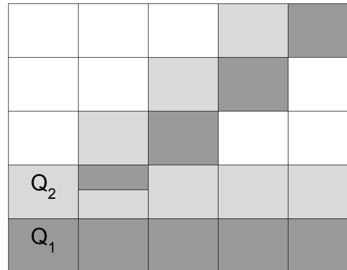
e) Also for $f = 2$ Algorithm 1 does not solve byzantine agreement. Observe first that all input values from the correct nodes will be accepted by all correct nodes (already proven in b)). A byzantine value does not have to be accepted by all nodes: assume that a byzantine b_1 node sends its value to exactly two correct nodes in the first round, we call these nodes c_1 and c_2. All correct nodes will receive the two echos from c_1 and c_2 for the value of b_1 in the second round. Note that b_1 has to echo the same value that it sent to the respective node in the second round and can therefore not change the number of echos. This can be done by the second byzantine node b_2. Since a correct node accepts a message if it hears at least 5 echoes, b_2 can send an echo to two other correct nodes (not c_1 or c_2). These two nodes, we call them c_3 and c_4, will hear three echos in the second round and send an echo themselves. In Line 10, c_3 and c_4 will receive 5 echos for the value from b_1. All other correct parties will receive less echos (one or four).

Assume now that the input value of c_3 and c_4 is 0, and the value of all other correct nodes is 1. The byzantine parties can use the strategy above to make sure that c_3 and c_4 accept two additional values 0 from the byzantine nodes, thus deciding on 0. The other correct nodes will only accept the values of the correct nodes thus deciding on 1.

SOLUTION

2 Quorum Systems (18 points)

Consider a quorum system consisting of $n = m^2$ server nodes arranged in a square. In this square, each quorum consists of a row and the diagonal starting in the leftmost node of the row. (The quorum in the top row does not have a diagonal anymore.) Two such quorums are shown in the figure:



a) [6] Under a uniform access strategy, what is the resilience of the quorum system? (Please be precise, if the resilience for example is $2\log n + 1$, write $2\log n + 1$ and not just $\mathcal{O}(\log n)$.)

We want to use our quorum system for a locking service. At any time, a client can ask an arbitrary quorum for a lock (as soon as possible). Each node in the quorum will then answer with the earliest 10 second interval the node has not sent to any client. The client waits for an answer from all nodes in its quorum, and chooses the latest 10 second interval in this set of answers. We assume that all nodes and clients are completely reliable and fast. We also assume that messages are asynchronous but fast.

b) [6] Show that this locking system is not correct, as it may happen that two clients get the lock for the very same time.

Let us adapt the locking system: When a quorum node u receives a request by a client c, the quorum node u first answers the client c (as before), and then exchanges its answer with the other nodes of the quorum the client c queried (all nodes of the quorum send a message to all nodes of the quorum). In particular, node u waits with answering a next client until it knows what every node of the quorum answered to client c.

c) [6] Is the locking system now correct?

a) Given that there are m quorums and each quorum overlaps with each other in one node, it is enough that $m/2$ nodes overlapping two different quorums fail for all the quorums to fail. Therefore, the resilience of the system is $\mathcal{R} = \lceil m/2 \rceil - 1$.

b) We can illustrate why this locking system is not correct with a simple example. If one quorum is locked and two clients access two other different quorums at approximately the same time, they will receive the same time notifications. More specifically, they will get from the free nodes the current time as the next available time and from the nodes overlapping with the locked quorum the time the lock is released. Then, they will both access the system at the time the lock is release, producing a collision, which shows that this locking system is not correct.

c) The new adapted locking system will not work either. To see why we need to consider the case where two clients c_1 and c_2 access the *same* quorum at the same time or almost the same time. In this case, when c_2 access a node u_1 that has been firstly accessed by client c_1, the node u_1 will wait for a notification from all other nodes containing the answer that they gave to c_1 before answering to c_2. However, a node u_2 that was firstly accessed by c_2 will wait for the answer of all other nodes to c_2 before answering c_1. In this way, u_1 will not answer c_2 until it receives the notification from u_2, but in turn u_2 will not send this notification because it will wait for u_1 to notify his answer to c_2, which produces a deadlock. This shows that this locking system is not correct.

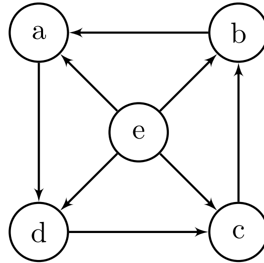
SOLUTION

3 Caching in a Directed Network (20 points)

Recall the selfish caching game on graphs with constant demand 1. The cost of node v is 1 if v caches the file, and $c_{v \leftarrow u}$ if v does not cache, where $c_{v \leftarrow u}$ is the shortest path length from v to a caching node u. If no accessible node caches the file, then node v incurs a cost $+\infty$.

In this exam question, we are interested in finding a Nash Equilibrium algorithm of the selfish caching game for *directed* graphs.

Consider the following example with the distance between two adjacent nodes of the outer square equal to $\frac{3}{4}$ and $c_{v \leftarrow e} = \frac{5}{4}$ for $v \in \{a, b, c, d\}$.



a) [5] What are all the pure Nash Equilibria in this example?

b) [7] Give one mixed Nash Equilibrium in this example.

*[continue **b)**]*

c) [4] Can you flip the direction of a single edge of the graph so that the network has only one pure Nash Equilibrium?

d) [4] Find a directed graph that does not have any pure Nash Equilibrium.

Solution

a) As the central node e has no neighbor, it has to cache the file. An outer node will cache the file if and only if its outer neighbor does not cache it, since any other node has a distance larger than 1. Thus, the two possible pure NE are when (a, c, e) or (b, d, e) are caching.

b) Node e still caches with probability 1, but all other nodes have a mixed strategy. Indeed, if an outer node has a deterministic strategy, every other node would follow their best response, and it would not be a mixed NE anymore. Let us consider an outer node caching with probability p. Since its predecessor should get the same payoff whether it caches or not, we know that p has to satisfy :

$$1 = p\frac{3}{4} + (1-p)\frac{5}{4}$$

which gives $p = \frac{1}{2}$. This is true for all nodes, and thus each outer node caches with probability $\frac{1}{2}$.

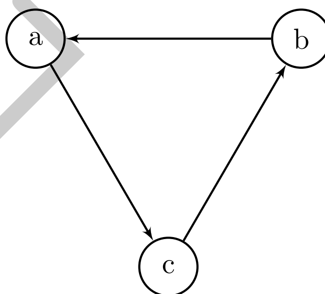
c)
- The social optimum is reached for the two pure NE. The cost is given by $3+2\times\frac{3}{4} = \frac{9}{2}$.
- The worst NE is the mixed one. The expected cost is given by $1+\sum_{v\in\{a,b,c,d\}} \mathbb{E}[cost(v)]$ which is equal to 5.

Thus, the Price of Anarchy is given by :

$$\text{PoA} = \frac{Cost(NE_)}{Cost(SO)} = \frac{10}{9}$$

d) We can flip the edge (a, d) and make the graph acyclic. Thus the decision of each node does not impact the strategy of any node it has access to, and knowing the strategy of those node it can access yields a unique best response. So (b, d, e) is the only NE of this graph.

e) We can take this example :



with a distance $\frac{3}{4}$ between two nodes.

It is impossible to have two or more nodes caching at the same time, since one of them would want to change its strategy as its parent caches. Also, it is impossible to have only one node caching, as one of the other nodes would be at distance $\frac{3}{2}$ and would better cache the file. Finally, it is impossible that no node is caching as any node would change its decision to avoid the $+\infty$ cost.

SOLUTION

4 Bitcoin (24 points)

Background: In Bitcoin, every node has its own clock. We assume that the clock skew of honest miners is at most 1 hour. Every block contains a timestamp specified by the miner who mined that block. A newly mined block is accepted only if its timestamp is strictly greater than the median timestamp of the previous 11 blocks, and less than $node_local_time + 2\ hours$ (timestamp bounding consensus rule).

The protocol tries to keep an expected block interval (time between two blocks) of 10 minutes. To achieve this, the mining difficulty parameter is adjusted every 2016 blocks ($\sim$ 2 weeks). Let's call this 2016 consecutive blocks a *period*. The difficulty adjustment is based on how long it took to mine the period. More precisely:

$$new_difficulty = old_difficulty \cdot \frac{expected_time}{elapsed_time}$$

where:

- $expected_time = 2016 \times 10 \times 60$ seconds
- $elapsed_time = max_timestamp(period) - min_timestamp(period)$

a) [4] If more miners enter the Bitcoin ecosystem during a period, does the difficulty parameter for the next period increase or decrease?

b) [4] If there was no upper bound on an accepted block's timestamp, how could an adversarial miner affect the difficulty parameter?

Assume an attacker manages to make timestamps in 4032 consecutive blocks look like this:

$$[\underbrace{t_0,\ t_0+1,\ t_0+2,\ ...,\ t_0+2014,\ \boldsymbol{t_1}}_{\text{2016 blocks } (period_1)},\ \underbrace{t_0+2016,\ t_0+2017,\ ...,\ t_0+4030,\ \boldsymbol{t_2}}_{\text{2016 blocks } (period_2)}]$$

where:

- t_0 = time snapshot at some arbitrary moment
- $t_0 + 1 = t_0$ plus 1 second
- $t_1 = t_0 + (2016 \times 10 \times 60)$ seconds
- $t_2 = t_0 + 2 \times (2016 \times 10 \times 60)$ seconds

c) [4] In this attack, what happens to the difficulty parameter at the end of $period_2$?

d) [3] Did the $max_timestamp(period_2)$ drift far from "real time"?

e) [3] Did the $min_timestamp(period_2)$ drift far from "real time"?

f) [3] Why wouldn't the attacker set t_1 to just be $t_0 + 2015$?

g) [3] Why is this kind of attack not a big cause for concern?

a) If more miners enter the ecosystem, blocks are generated faster than 1 block per 10 minutes. This will cause *elapsed_time* to go smaller than expected time, and thus difficulty will increase.

b) An adversarial miner could add a far-future-timestamp to one of his blocks, and this would cause *elapsed_time* to be much larger than *expected_time*, and that would lower the difficulty considerably.

c) At the end of t_2, we have $elapsed_time \approx 2 \times expected_time$. This causes difficulty to decrease approximately by a factor of 2.

Additional note on difficulty adjustment (not expected in an answer): Bitcoin has safeguards to ensure that difficulty cannot go up or down by more than a factor of 4. But a 50% reduction of difficulty is also quite bad. If this continues for a few more periods, Bitcoin mining will become unreasonably easy (the opposite of difficult), and that's not a good thing.

Additional note on the attack (not expected in an answer): To execute this attack, we need a period where the lowest time is as close to t_0 as allowed and the highest time is t_2. Without loss of generality, we can assume that at the start of the first period, the timestamp is t_0 and it is the "correct time." To get t_2 into this period, we need to have some block that has a timestamp that is $t_0 + 2 \times (2016 \times 10 \times 60)$. All nodes will reject this block because it's too far out in the future (the upper bound is Network time + 2 hours). So, breaching the upper bound on the timestamps is a not possible. Breaching the lower bound, on the other hand, can be done with 2 periods. Note that the timestamp of a block is valid if it is greater than the median of the last 11 timestamps. In the first period, it's possible to keep all but the last timestamp as close to t_0 as possible without violating the timestamp bounding consensus rule by incrementing each block's timestamp by just 1 second. The last timestamp of the period (t_1) is set to the "correct time" to ensure that $t_1 - t_0 \approx expected_time$ so that difficulty doesn't increase at the end of the first period. In the second period, we again start as close to t_0 as possible ($t_0 + 2016$) without violating the timestamp bounding consensus rule because the median of many timestamps close to t_0 and one t_1 is still closer to t_0. Now that we are in the actual time range above t_1, we can get to t_2 in this time period without going too far into the future. This gets us both $t_0 + \epsilon$ and t_2 in the same time period - this pulling the time warp attack.

d) No. $max_timestamp(period_2)$ is close to "real time". $max_timestamp(period_2)$ is $t2$, which is $t_0 + 2 \times (2016 \times 10 \times 60)$, is 2 periods after t_0. As difficulty of mining has not changed till the end of t_2, "real time" would have also advanced to around the same time.

e) Yes. $min_timestamp(period_2)$ has drifted far from "real time". The attacker did this by manipulating the timestamps of $period_1$ such that the median of the last 11 timestamps never increases too much. This way, $period_2$'s timestamps start closer to t_0 rather than t_1 (which it would, in the absence of this attack).

f) t_1 has to be $t_0 + (2016 \times 10 \times 60)$ to avoid difficulty going up at the end of $period_1$. The objective of the attack is to bring down the difficulty.

g) To force timestamps to all be a known low value requires that more than 50% miners are acting together for a substantial amount of time. If this were true, we have other larger problems like double-spending attacks to worry about. Additionally, this attack would be visible in public, and miners could be incentivized by timelocked transactions that reward them if they advanced the time to the right values.

SOLUTION

SOLUTION