COMP6451 T1 2022
Assignment 1
Total Marks: 30
Due: 23:59 March 13, 2022

Submissions: Submit your solutions as a pdf or text file via the course moodle page. Your submission must be your individual work - UNSW rules concerning this will apply (see the Course Outline). Turnitin will be used to perform similarity checks. In general, these are short answer questions — aim to keep your answers brief but precise. Answer all parts, and show your working.

Question 1 (2 marks): Consider the issuance policy for a cryptocurrency that has been designed to be arbitrarily divisible. That is, unlike Bitcoin, in which the smallest unit of value is the Satoshi, this cryptocurrency represents the value of an account as a rational number a/b stored using a pair of integers (a, b).

Like Bitcoin, the system creates a sequence of blocks, and with each block created, the miner of the block is rewarded with newly created coins. The value of the block reward is initially a number N, but this number is periodically changed, by dividing by a rational number $r > 0$. More precisely, for a given block reward value x, exactly B blocks are created with reward value x per block, and the next set of B blocks is created with reward value x/r. Since the currency is arbitrarily divisible, it is always possible to represent this block reward as a value in the currency, even if it becomes very small. (If this currency takes off like Bitcoin, small amounts of this currency could actually be worth large amounts of dollars!).

What is the total amount of currency that is ever issued over all time in this currency system? Is it finite or infinite? If the answer depends on the values N and/or r, explain how it depends, and give the conditions for each case. If the answer is that the amount is finite, give a simple expression for your answer (not an infinite summation.) (Hint: nothing beyond what has been presented in lectures is needed to answer this question.)

Question 2 (5 marks): Design a data storage service that can be used to solve the following problem for a networked client device that has a very small memory:

`Data to be stored:` a very long but unchanging list L of 64 bit natural numbers, sorted from smallest to largest. The list cannot be stored permanently on the client device since the amount of memory required is too large, and the available memory is required for other computations that need to be performed by the client.

`Client side storage between queries:` as small an amount of data as possible

`Client Queries:` Given natural numbers $a \leq b$, fetch all the entries x in the list L with $a \leq x \leq b$

The input is provided just once, but multiple queries (with different values a, b) may be issued by the client. The client makes use of the data returned from a query for a short period of time, but then erases this data in order to free memory for other purposes.

The service is not trusted by the client. For example, the client is worried that the data store may be hacked and the data corrupted, so that it returns incorrect answers to queries. The client therefore requires a proof that the answer returned is correct. In addition to a list of numbers R in the query interval $[a, b]$, the server may provide some amount of additional "proof information" in its response. The client should be able to use this proof information to verify that in fact the server response R is equal to the correct answer $[x \in L \mid a \leq x \leq b]$ with very high probability. A small amount of information may be stored by the client at system setup and between queries in order to assist with this verification, but this should be kept at a minimum in order to free up as much memory on the client as possible.

A further requirement is that, in order to reduce network load, your solution should minimize the size of these proofs. That is, for a given query, the size of the proof information returned with the answer should be as small as possible. You may assume that the client is prepared to expend computation effort on verifying that the proof is correct, but again, the amount of computation should be minimized.

You do not need to provide code - just sketch the main ideas for a solution and analyse its resource requirements (message length, storage, computation time).

You answer should do the following:

1. Describe the data structure stored on the server, and how it is computed. Give an expression for the size of this data structure as a function of the length of L.

2. Describe the data that is stored at the client, and state its size.

3. Describe the "proof information" returned for a query, and state its size as a function of the length of the list of numbers R in the query response.

4. Describe the computation that the client performs to verify the proof. What is the running time of this computation, again as function of the length of R? Explain how this computation provides a guarantee to the client that the answer provided by the server is correct.

5. Suppose that it is known in advance that client wishes to process the list from left to right in blocks of unknown length. That is, the sequence of queries that the client will issue has the form $[a_1, b_1], [b_1, b_2], \ldots, [a_n, b_n]$, where a_1 is the minimal element of L, but the numbers $b_1 \leq b_2 \leq \ldots \leq b_n$ are not known in advance. Is your solution the most efficient possible solution for this special case? Explain your answer.

Question 3 (Total 13 marks): This question is intended to give you a feel for why designing protocols for distributed systems in which some of the nodes may be faulty is non-trivial.

Suppose that we are working on network that is *reliable* and *synchronous* and *fully connected*. Write n for the number of nodes on the network. Reliability here means that every message that is sent is delivered within a known time bound. For simplicity, we suppose that this time bound is less than 1 unit of time. Thus, a message that is sent at time t is guaranteed to be delivered before time $t + 1$. Synchrony means that the clocks of nodes on the network always indicate the same time. That is, at time t, all the clocks of nodes on the network say that it is time t. Fully connected means that any node is able to send a message directly to any other node; it does not need to rely on any other nodes to forward the message to ensure that it will be delivered.

These assumptions mean that we can design protocols that operate in a sequence of *rounds*. We work with natural number times, so that the system starts at time 0. Round $k \geq 1$ happens between time $k - 1$ and time k. The structure of a round k is as follows. At time $k - 1$ each node works out from its local state (whatever data it maintains locally) what messages it will send in that round to the nodes on the network, and it sends these messages. All the messages sent at time $k - 1$, are delivered during the same round, that is, just before time k. Just before the end of the round, each node looks at what messages it has received during the round and updates its local state to record some information about these messages. This state update is complete by time k, when the next round begins.

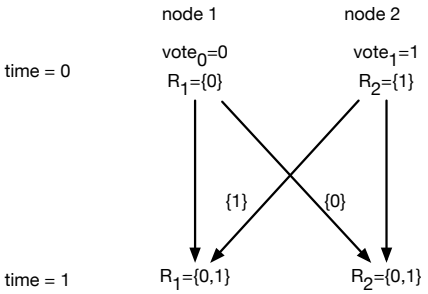
Some of the nodes may be faulty. Consider the following specification of a consensus protocol for agreeing on a binary value 0 or 1, which requires that the non-faulty nodes form a consensus. Each node i starts at time 0 with a value $vote_i \in \{0, 1\}$. Each node i is able to make a decision on a value $x \in \{0, 1\}$ by performing an action `decide`(x).

- **Unique-Decision:** A node may make at most one decision.

- **Agreement:** If nodes i and j are both non-faulty, and make decisions v_i and v_j, respectively, then $v_i = v_j$.
- **Validity:** If node i decides value v, then for some node j we have $vote_j = v$.
- **Termination** All non-faulty nodes eventually make a decision.

Consider the following simple protocol. In addition to $\texttt{vote}_i$, each node $i = 1 \ldots n$ stores a variable $R_i \subseteq \{0, 1\}$ in its local state, as well as the clock value $\texttt{time}_i$. The protocol operates by having the nodes tell each other about votes that they know about, and R_i is, intuitively, the set of votes that node i has heard about. At time 0, we have $R_i = \{\texttt{vote}_i\}$. At time t, each node i sends its current value of R_i to all other nodes. If node j receives messages $m_1, \ldots m_k$ during the round (each a subset of $\{0, 1\}$) it updates its local state by the assignment $R_j := R_j \cup m_1 \cup \ldots \cup m_k$.

We may illustrate such a protocol using a *message sequence chart*, in which we indicate the values of the variables and the message sent at each time. An example with a single round and two nodes is the following. We would continue down the page for more rounds. The `vote` values are indicated only at time 0 since they do not change. We also omit the local time variables $\texttt{time}_i$, since always $\texttt{time}_i = \texttt{time}$.



Question 3 (part 1) (2 marks): Let's first consider the situation in which no node is faulty, and assume that this is known to all nodes in the system. Show that each node is able to compute a decision at time 1, in such a way as to satisfy all the properties of the specification above. (Write some pseudo-code for making the decision, and explain briefly why each property holds.) Note that you need to show that your solution is correct for all choices of votes. With n nodes, there are 2^n such choices.

Suppose now that some of the nodes can be faulty, in such a way that non-faulty nodes still follow the protocol exactly, but faulty nodes may crash at any time. Once a node has crashed it stays crashed. At the time it crashes, the node sends an arbitrary subset of the messages that it was supposed to send at that

time. All the messages that it manages to send are delivered, but (obviously), none of the messages that it did not manage to send are delivered. If a node did not crash, it updates its local state according to the same rule as above, where $m_1, \ldots, m_k$ are the messages that were delivered to it in the round. (In the previous case, we would have $k = n-1$, but now we can have $n-f \leq k \leq n-1$, where f is an upper bound on the number of faulty nodes.

Question 3 (part 2) (2 marks): Give an example to show that if there is even one faulty (crashing) node, applying your decision rule from part 1 at time 1 does not satisfy the specification. (Present the example using a message sequence chart, in which you indicate the faulty node with an X at the time it crashes. Messages that the crashing node fails to send can be simply omitted from the diagram.) State precisely which part of the specification is not satisfied, and why.

Question 3 (part 3) (2 marks): Let the number of nodes be an arbitrary number n, and suppose that there is at most one faulty node that fails by crashing. Show that if we repeat the message transmission rule for two rounds, then at time 2 the non-faulty nodes can make a decision so as to satisfy the specification. (It is not known in advance whether there are 0 or 1 faulty nodes, nor is it known at what time the faulty node fails, or what messages it manages to send. Your explanation should be general enough to cover all the possible cases.)

Question 3 (part 4) (2 marks): For an arbitrary $f < n-1$, draw the structure of a message sequence chart that shows that if there are at most f faulty nodes, which fail by crashing, then it is *not* possible to make a decision, so as to always satisfy the specification, before time $f+1$. Explain your diagram.

Question 3 (part 5) (3 marks): (Harder) Suppose that there are at most $f < n$ faulty nodes, which fail by crashing. Show that repeating the protocol for $f+1$ rounds enables a decision to be made at time $f+1$, so as to satisfy the specification.

Question 3 (part 6) (2 marks): Now consider the same protocol for a different model of the ways in which a faulty node may behave. Suppose that a faulty node does not crash, but it may send inconsistent and incorrect messages. For example, if it has $R_i = \{0\}$, then a faulty i may send some nodes the message $\{1\}$ and some nodes the message $\{0, 1\}$. Show by example that if there are 3 nodes, one of which is faulty ($f = 1$), then deciding at time $f + 1 = 2$ does not satisfy the specification. Again, indicate what part of the specification fails. Is there any number of iterations of the protocol that allows a decision to be made so as to satisfy the specification?

Question 4 (5 marks): A national security agency has discovered that it can crack the cryptography being used by the enemy if it can solve the following

puzzle for particular 256 bit values N_1, N_2:

> Find two 32 bit numbers x_1, x_2 such that $x_1 < x_2$ and
> $h(x_1) = N_1$ and $h(x_1 + x_2) = N_2$.

where h is SHA-256. Solving the puzzle requires a very large and costly amount of computation, and to offload the cost, the agency comes up with a clever scheme. They pretend to be a philanthropist who is offering a 1 bitcoin donation to the first person who solves the puzzle. They hope that this will encourage many people around the world to use their desktop machines to search for a solution to the puzzle and submit the solution to the blockchain, where the agency will be able to read it. Describe precisely how they could do this with a Bitcoin transaction that will pay 1 bitcoin to anyone who solves the puzzle. Your answer should do the following should do the following:

- Give the locking script for this transaction.
- Give the unlocking script for the transaction.
- Show the sequence of stack values for a successful unlocking computation. Use abstract values such as x_1, $h(x_1)$ rather than actual numbers.
- Mary gets lucky, and solves the puzzle. What should Mary do to make sure that she collects the reward?

Hint: The complete set of Bitcoin Script opcodes is at

`https://en.bitcoin.it/wiki/Script`

You might find it helpful to use the Bitcoin Script simulator at

`https://siminchen.github.io/bitcoinIDE/build/editor.html`

to develop your solution. In your answers, use the opcode words rather than the corresponding numbers. Also follow the convention of omitting the value-pushing opcodes (that is, write just x instead of OP_DATA 1 x).

Question 5 (5 marks): The simple secret sharing scheme discussed in the Key Management lecture, of distributing sections of the secret key (shares) to different locations, has the property that compromise of one share decreases the cost of a brute force attack, because the attacker now knows some of the bits of the key.

Consider the following alternative. Let k be the secret key, generated uniformly at random, represented as a number mod n for some appropriate n. Generate p random numbers $x_1, .., x_p \in [0, n-1]$ uniformly at random and let $x_{p+1} = -(x_1 + ... + x_p)\ mod\ n$

Define the $p+1$ shares to be

$$k + x_1\ mod\ n,\quad k + x_2\ mod\ n,\ ....\ ,\quad k + x_{p+1}\ mod\ n$$

Prove the following:

1. Recombining the $p+1$ shares allows k to be reconstructed (explain how). If you need any constraints on any of the numbers for the proof to work, say what they are.

2. An attacker who learns any p of the $p+1$ shares still gets no information about k. That is, any key k' is consistent with the p shares, with every possible key k' having the same probability, from the attacker's point of view, of being the secret k.

– END –