CS 539 - Distributed Algorithms

Wenqi He (wenqihe2)

P1

Algorithm

Assuming (i) lockstep synchrony (ii) a spanning tree has been built:

```
Initially: for each node, no link is labeled and no child has been notified
Round 1: root sends Continue(0) to itself
Round r: for each node i,
  if received MarkEdge(n) from j in round r - 1:
    assign label n to edge (i,j)
  if received Continue(n) in round r - 1:
    n' := n
    while not all incident edges are labeled:
      increment n' and assign n' to any remaining edge (i,j)
      send MarkEdge(n') to process j
    if all children have been notified:
      if process i is not root:
        send Continue(n') to parent
      terminate
    else:
      send Continue(n') to any remaining child
      mark the child as notified
```

Correctness

First of all, in each round, there is only one "active" node which receives a *Continue* message.

It can be shown by induction that every node will eventually receive a *Continue* message, and therefore every edge will be assigned a label at least once. Furthermore, when node i receives a *Continue*, if any edge (i, j) has already been labeled by node j, the *MarkEdge* message sent by j either was handled by i in some previous round or will be handled by i in the current round before *Continue* – that is, before any new assignments happen. Thus, no edge will be assigned more than once.

It can also be shown by induction that when a *Continue*(n) message is sent, exactly n edges have been assigned unique labels from 1 to n. This is obviously true initially when $n = 0$. Now suppose the statement holds in round r, and a *Continue*(n) is sent. In round $r + 1$, another node receives it and marks another $m \geq 0$ edges. None of the m edges has been assigned a label before, and they will now be labeled $n + 1, n + 2, \cdots, n + m$. By the end of this round, when a *Continue*$(n + m)$ message is sent, exactly $n + m$ edges have been assigned unique labels from 1 to $n + m$. Thus the statement still holds. Eventually, the node that labels the last edge must send a *Continue*$(|E|)$, which means that by now all $|E|$ edges have been assigned unique labels from 1 to $|E|$.

Efficiency

Exactly $|E|$ *MarkEdge* messages will be sent, one along each edge of the graph. In addition, exactly $2(|V|-1)$ *Continue* messages will be sent, one in each direction along each edge of the spanning tree. The message complexity is therefore $O(|E| + |V|)$.

The round complexity is $O(|V|)$, as exactly one *Continue* message is sent in each round.

P2

Algorithm

It is easier to consider a more general algorithm that applies to $n \geq 2$ processes, assuming a fully connected graph. The intuition is that the lower bound of the max error $(n-1)U/n$ can indeed be achieved if all nodes attempt to set their clocks to an estimated average. The factor $1/n$ comes from taking an average, and the $n-1$ factor comes from communicating with all other $n-1$ nodes. The algorithm is as follows:

```
Each node i:
  send clock reading HC_i(t_i) to all other nodes
  upon receiving HC_j(t_j) from node j:
    save current clock reading HC_i(t'_j) along with the message
  upon receiving messages from all other n-1 nodes:
    set AC_i(t) := HC_i(t) + (1/n) * sum_{j≠i} ( HC_j(t_j) + (1/2)(d+D) - HC_i(t'_j) )
```

Correctness

Suppose the message from process j to process i is sent at "physical time" t_j and received at "physical time" t'_j, the actual delay for the message is $\xi_j := HC_j(t'_j) - HC_j(t_j)$. The correct average is:

$$\begin{aligned} AC_{correct}(t) &= \frac{1}{n}\sum_{j=1}^{n} HC_j(t) = \frac{1}{n}\sum_{j=1}^{n}\big(HC_i(t) + HC_j(t) - HC_i(t)\big) \\ &= HC_i(t) + \frac{1}{n}\sum_{j\neq i}\big(HC_j(t) - HC_i(t)\big) \\ &= HC_i(t) + \frac{1}{n}\sum_{j\neq i}\big(HC_j(t'_j) - HC_i(t'_j)\big) \\ &= HC_i(t) + \frac{1}{n}\sum_{j\neq i}\big(HC_j(t_j) + \xi_j - HC_i(t'_j)\big) \end{aligned}$$

Compare this with the adjusted clock of process i:

$$\begin{aligned} |AC_i(t) - AC_{correct}(t)| &= \left|\frac{1}{n}\sum_{j\neq i}\left(\frac{1}{2}(d+D) - \xi_j\right)\right| \\ &\leq \frac{1}{n}\sum_{j\neq i}\left|\frac{1}{2}(d+D) - \xi_j\right| \\ &= \frac{1}{n}\sum_{j\neq i}\frac{U}{2} = \frac{(n-1)U}{2n} \end{aligned}$$

By triangular inequality,

$$|AC_i(t) - AC_j(t)| \leq |AC_i(t) - AC_{correct}(t)| + |AC_{correct}(t) - AC_j(t)| \leq \frac{(n-1)U}{n}$$

When $n = 3$, the max error is $2U/3$.

Efficiency

Between every pair of processes exactly two messages are sent, so the message complexity is $O(|V|^2)$. The round complexity is $O(1)$, since every process sends its time to all neighbors in one step.

P3

We can represent process p's i-th event in round r as a tuple (p, r, i). An execution can be represented as a sequence of (p, r, i) tuples. This naturally define a total order: $e_1 \prec e_2$ if e_1 is scheduled before e_2. Imagine if we sort this sequence by round, and within each round by $\prec$. Formally, we can define another total order $\prec^*$:

Definition 1. $(p_1, r_1, i_1) \prec^* (p_2, r_2, i_2) := r_1 < r_2 \vee (r_1 = r_2 \wedge (p_1, r_1, i_1) \prec (p_2, r_2, i_2))$.

This creates a clear boundary for each round by ensuring that all round-r events are scheduled before all round-$(r+1)$ events. The sorted sequence essentially represents a globally synchronized execution. Now I'll prove that the two schedules defined by $\prec$ and $\prec^*$ respectively are equivalent:

First, with a local synchronizer, a process receives round-r messages only after sending round-r messages, and sends round-$(r+1)$ messages only after receiving all round-r messages. This guarantees that for any process, if two local events $e_1 \prec e_2$, then $r_1 \leq r_2$. One only needs to check the definition of $\prec^*$ to see that $e_1 \prec e_2 \Leftrightarrow e_1 \prec^* e_2$ in this case. That is, the local orderings of events are identical. Second, if e_1 is the sending of some message and e_2 is the receiving of it, then $r_1 = r_2$. It's also obvious from the definition of $\prec^*$ that for each pair of sending/receiving events, $e_1 \prec e_2 \Leftrightarrow e_1 \prec^* e_2$. In other words, if the original schedule is legal, the new schedule must be legal. Q.E.D.

It can be shown with a similar reasoning that if $\prec$ is an linear extension of the partial order $\rightarrow$ (happens-before), i.e. $a \rightarrow b \Rightarrow a \prec b$, then $\prec^*$ must also be a linear extension of $\rightarrow$, and vice versa. In other words, the two schedules are two different topological sorts of the same DAG defined by the happens-before relation. Since the two schedules respect the same happens-before relation, they represent equivalent executions.

P4

The first round determines the outcome. In round 1, the sender might:

1. **Send no messages.** If no 1-signature chain is produced in round 1, no chain will ever be produced, because it can never have a valid length. Everyone stays idle, and outputs $\perp$ by the end of round $f + 1$.

2. **Send the same value v to all nodes.** In round 2, everyone echoes v to each other. Since v is already seen by everyone, nothing happens from round 3 onwards. By the end of round $f + 1$, every honest party outputs v.

3. **Send the same value v to $m < n - 1$ nodes.** In round 2, those m nodes echo v. In round 3, the other $n - 1 - m$ nodes echo v. From round 4 onwards, nothing happens. By the end of round $f + 1$, every honest party outputs v.

4. **Send different values to any number of nodes.** In round 2, those who received values in round 1 echo those values. By the end of round 2, everyone receives all values but only accepts two. In round 3, those who learned (either one or two) new values in round 2 echo those values. Since everyone has already seen two values, nothing happens from round 4 onwards. By the end of round $f + 1$, every honest party outputs $\perp$.

What the sender does in round $r > 1$ doesn't matter. In each round, the sender could:

(a) **Send no messages.** All other nodes proceed as normal.

(b) **Send valid chains to any number of nodes.** All other nodes proceed as normal, because they receive these values from honest nodes anyways, and duplicate values are simply ignored.

(c) **Send invalid chains to any number of nodes.** This includes *sending a new value, double-signing a message, delaying the forwarding of a message for at least one round*, etc. In round $r+1$, the receivers will check that these messages are invalid and discard them.

All of (a) - (c) won't cause more messages to be sent from honest parties. Ignoring messages sent by the Byzantine sender, in scenario 1, no message is ever sent, so the complexity is $O(1)$. In all other scenarios, the message complexity is $O(n^2)$.

P5

(i)

Suppose solution exists for $n \leq mf$. First, let's define $P(v)$ to be the set of nodes whose inputs are v. Since $n \leq mf$, we can carefully assign input to each node such that $\forall i \in [1, m], |P(v_i)| \leq f$. Suppose there are no Byzantine parties and everyone outputs v_k. Since our assignment guarantees that $|P(v_k)| \leq f$, we can imagine an alternative world where all nodes in $P(v_k)$ are Byzantine but behave honestly. For others, the two worlds are indistinguishable, so they must still output v_k, but this violates the validity requirement. Q.E.D.

(ii)

When $n > mf$, the pigeonhole principle kicks in. The algorithm is as follows:

Algorithm

Run Dolev-Strong from every node simultaneously, except that:
1. At most two distinct values are forwarded for each sender.
2. At the end of round $f+1$, every node i computes a vector w_j of size n, where:
   if $j = i$, w_j is node i's input
   else if exactly one value v_k is observed from j, $w_j := v_k$
   otherwise $w_j := v_1$.

Finally, every node outputs $Majority(w_1, ..., w_n)$.

Correctness

Liveness follows from the liveness of Dolev-Strong. Safety also follows from the safety of Dolev-Strong: All nodes compute the same vector, and since $Majority$ is deterministic, they must arrive at the same output. Since $n > mf$, by pigeonhole principle, the output must have more than f votes, so at least one vote must come from an honest party. Thus our definition of validity is satisfied.

Efficiency

Since n executions of Dolev-Strong are run simultaneously, we have:

- round complexity: $O(f)$
- message complexity: $O(n^3)$
- bit complexity: $O(n^3 f|\sigma|)$

P6

Algorithm

```
Sender:
  proposes a value v.
  upon receiving n − f signed votes for v:
    forward the quorum of votes to everyone.
    output v.

Everyone else:
  upon receiving value v:
    if no value has been received,
      send back a signed vote for v.
  upon receiving a quorum of n − f votes for value v:
    output v.
```

Correctness

Validity

If the sender is honest, it will send the same v to everyone, then it's guaranteed to receive at least $n-f$ votes for v from non-faulty parties alone, so the second round will happen and every honest party will output v.

Safety

Safety is guaranteed by unforgeable signatures and quorum intersection: If two honest parties output y and y' respectively, they must have received at least $2(n-f)-n = n-2f \geq f+1$ votes from the same nodes, of which at least one must be honest. Because an honest party only votes once, $y = y'$.

Liveness

There is no liveness requirement for consistent broadcast. (This protocol doesn't make any liveness guarantees, so it's only consistent broadcast rather than reliable broadcast: A Byzantine sender could selectively send votes to only a few honest parties so they will output while others won't.)

Efficiency

Round complexity: 3 – one each for proposal, voting, and transfer of certificate.
Message complexity: $O(n)$, there is only one-to-all communication.
Bit complexity: $O(n^2|\sigma|)$, since a quorum of $n-f$ signed votes is sent to everyone.

P7 ($f < n/5$)

Algorithm

```
Every party i:
  send (vote_1, input_i) to everyone.
  upon receiving n − f (vote_1, ∗):
    if at least n − 2f votes are for the same v, send (vote_2, v) to everyone.
    else send (vote_2, ⊥) to everyone.
  upon receiving n − f (vote_2, ∗):
    if at least n − 2f votes are for the same non-⊥ value v, output (v, 1).
    else if at least f + 1 votes are for the same non-⊥ value v, output (v, 0).
    else output (⊥, 0).
```

Correctness

Validity

If every honest party has has input y, then in round 1, every honest party will get at least $n-2f$ $(vote_1, y)$ required to send $(vote_2, y)$ and therefore will send $(vote_2, y)$. Consequently, in round 2, every honest party will get at least $n-2f$ $(vote_2, y)$ required to output $(y, 1)$ and therefore will output $(y, 1)$.

Safety

In round 2, honest parties who don't vote for $\perp$ can only vote for a single non-$\perp$ value. If two honest parties send $(vote_2, y)$ and $(vote_2, y')$ respectively, where $y \neq \perp$ and $y' \neq \perp$, then by quorum intersection, at least $2(n-2f) - n = n - 4f$ parties – out of which at least $n - 5f \geq 1$ are honest – voted for both y and y' in round 1, therefore $y = y'$. An immediate corollary is that if any honest party votes for some non-$\perp$ value y in round 2, then y is the only non-$\perp$ value any honest party can output, because without support from honest parties, no other non-$\perp$ values can get more than f votes.

If an honest party outputs $(y, 1)$ in the end, it has received at least $n-2f$ votes for y in round 2, which means at least $n-3f$ honest parties voted for y. By quorum intersection, everyone is guaranteed to get at least $(n-f) + (n-3f) - n = n - 4f \geq f + 1$ votes for y from these nodes. Because everyone gets enough votes for y and there is no competition from other non-$\perp$ values, every honest party must output $(y, 0)$ if not $(y, 1)$.

Liveness

In round 1, all $|N| \geq n - f$ honest parties must send votes, so there will be at least $n - f$ votes available for every honest party to advance to round 2. Consequently, in round 2, all $|N| \geq n - f$ honest parties must send votes, so there will be at least $n - f$ votes available for every honest party to make a decision.

Efficiency

Round complexity: 2
Message complexity: $O(n^2)$, since everyone sends messages to everyone in constant rounds.

P7 ($f < n/3$)

Algorithm

```
Every party i:
  send a signed (vote_1, input_i) to everyone.
  upon receiving n − f (vote_1, ∗):
    if all n − f votes are for the same v,
      send (vote_2, v) to everyone, with the quorum of (vote_1, v) attached.
    else
      send (vote_2, ⊥) to everyone.
  upon receiving n − f verified (vote_2, ∗):
    if all n − f votes are for the same non-⊥ value v, output (v, 1).
    else if at least one vote is for a non-⊥ value v, output (v, 0).
    else output (⊥, 0).
```

Correctness

Validity

This protocol only satisfies a weaker form of validity: *If every party is honest and has input y, then every party outputs* $(y, 1)$. This is obvious: Everyone sends $(vote_1, y)$, so everyone will get $n - f$ $(vote_1, y)$ required

to send $(vote_2, y)$, so everyone will send $(vote_2, y)$. Consequently, everyone will get $n - f$ $(vote_2, y)$ required to output $(y, 1)$, so everyone will output $(y, 1)$.

Safety

First, I'll show that $(vote_2, *)$ can carry only one non-$\perp$ value, which means that the first round determines a unique candidate y so that the only possible outputs in round 2 are $(y, *)$ and $(\perp, 0)$. This follows from quorum intersection: If two parties – whether faulty or non-faulty – vote for $y \neq \perp$ and $y' \neq \perp$ respectively in round 2, then in round 1, $n-f$ parties voted for y and $n-f$ voted for y'. This condition is guaranteed by the attached quorum of votes with unforgeable signatures, which prevents Byzantine parties from making up arbitrary values. By quorum intersection, at least $2(n-f) - n = n - 2f \geq f + 1$ parties – out of which at least one must be honest – voted for both y and y', therefore $y = y'$. It remains to be proved that all honest parties can't output $(\perp, 0)$.

Let N and F be the sets of all non-faulty and faulty parties respectively. A node p receives votes from a non-faulty set $N_p \subseteq N$ and a faulty set $F_p \subseteq F$. For any two parties p and q, N_p and N_q must intersect, because $|N_p| + |N_q| - |N| = (n - f - |F_p|) + (n - f - |F_q|) - (n - |F|) = (n - 2f - |F_p|) + (|F| - |F_q|) \geq (n - 3f) + (|F| - |F_q|) \geq 1$. Suppose an honest party p outputs $(y, 1)$. For any other honest party q, at least one sender in N_q is also a member of N_p that sent $(vote_2, y)$ to p. Because it is honest, it must send $(vote_2, y)$ to q as well. Therefore, every honest party gets at least one vote for y and therefore must output $(y, *)$.

Liveness

In round 1, all $|N| \geq n - f$ honest parties must send votes, so there will be at least $n - f$ votes available for every honest party to advance to round 2. Consequently, in round 2, all $|N| \geq n - f$ honest parties must send votes, so there will be at least $n - f$ votes available for every honest party to make a decision.

Efficiency

Round complexity: 2
Message complexity: $O(n^2)$, since everyone sends messages to everyone.
Bit complexity: $O(n^3|\sigma|)$, since each $(vote_2, *)$ could potentially carry $n - f$ signed $(vote_1, *)$.

P8

Suppose solution exists for $n \leq 3t + 2c$, then it's possible to divide all parties into five groups, L, R, T, C_L, C_R where $|L| \leq t$, $|R| \leq t$, $|T| \leq t$, $|C_L| \leq c$ and $|C_R| \leq c$, such that all nodes in one of L, R or T could potentially be Byzantine and all nodes in one of C_L or C_R could potentially crash. Consider the following two symmetric scenarios:

I. All nodes in $L \cup C_L \cup T$ have input v, while all nodes in $R \cup C_R$ crash from the start. All nodes in $L \cup C_L$ must output v after finite steps due to validity.

II. All nodes in $R \cup C_R \cup T$ have input v', while all nodes in $L \cup C_L$ crash from the start. All nodes in $R \cup C_R$ must output v' after finite steps due to validity.

Now consider Scenario III where no nodes crash, but the communication between a node in $L \cup C_L$ and a node in $R \cup C_R$ takes longer than it takes for them to output in Scenario I/II (suppose GST is large enough for this to happen), then all nodes on one side appear dead to the other side. Additionally, all nodes in T are Byzantine and behave to $L \cup C_L$ as if (a) it has input v and (b) all nodes in $R \cup C_R$ are dead, and similarly, to $R \cup C_R$ as if (a) it has input v' and (b) all nodes in $L \cup C_L$ are dead. $L \cup C_L$ can't distinguish Scenario I and III, so they must output v. Similarly, $R \cup C_R$ can't distinguish Scenario II and III, so they must output v'. Since there are different outputs from non-faulty parties, safety is violated, thus the solution cannot be correct. Q.E.D.

P9

Algorithm

To implement a MRSW regular register, the $update(v, t)$ call for read operations is no longer needed, because its purpose is to ensure linearizability, which is not required by regular registers. Using the same $update(v, t)$ subroutine as defined in the ABD algorithm:

```
The writer keeps a local timestamp t
Each reader keeps a local register (v, t)

Writer:
  upon write(v) operation:
    t ← t + 1
    update(v, t)
    return

Reader:
  upon read operation:
    request local (v, t) from all processes and wait for n − f responses
    (v, t) ← (v′, t′) where (v′, t′) is the tuple with the highest timestamp
    return v
```

Correctness

A read that doesn't overlap with writes returns the value from the most recent write due to quorum intersection: the call to $update(v, t)$ during the most recent write ensures that $n - f$ processes see v. A reader that reads afterwards must first receive values from $n - f$ processes, among which at least one has seen v and therefore must send (v, t). Because v has the highest timestamp so far, the return value must be v.

For a read that overlaps with writes, at least $n - f$ processes have seen the previous/initial value (v', t') and therefore must have local timestamps $t \geq t'$ (monotonicity is ensured by $update(v, t)$). By quorum intersection, the reader must receive at least one value with timestamp $t \geq t'$, which means that the return value must have timestamp $t \geq t'$. In other words, the return value must be either the initial value $(t = t')$ or the value from one of the overlapping writes $(t > t')$.

Efficiency

Round complexity: 2 for write, 2 for read (as opposed to 4 with ABD).
Message complexity: $2n$ for write, $2n$ for read (as opposed to $4n$ with ABD).

P10

No. Consider the following sequence of operations (there could be other operations before or after):

```
[     write(v)     ]    (writer)
 [ read ]  [ read ]     (reader i)
```

Suppose **both reads by reader i overlap with the write to** $Reg[i][i]$, then obviously both reads of $Reg[i][i]$ overlap with the write to $Reg[i][i]$. Because each base register is only a regular register, it is possible that the first read of $Reg[i][i]$ returns the new value (v, t) while the second returns the old value $(v', t - 1)$ where $v' \neq v$. Additionally, suppose **no other reader reads** (v, t), then the timestamp of each $Reg[j][i]$ where $j \neq i$ is at most $t - 1$ during these two reads by reader i, so the first read must return v with timestamp t and the second v' with timestamp $t - 1$, which is a new-old inversion. The sequence is clearly not linearizable and therefore the register is not atomic. Q.E.D.

P11

(a)

The algorithm is not safe. Consider the following operations of two processes:

```
   *                          *         *
[ read(busy) ][ write((busy, turn)) ][ read(turn) ]
          [ read(busy) ][    write((busy, turn))    ][ read(turn) ]
               *                                  *      *
```

Suppose there are no other processes and `busy` is initially `false`, then both processes read `false` from `busy` and their respective process IDs from `turn`, so they both enter the critical section.

(b)

The algorithm is not deadlock-free. Consider the following operations of two processes:

```
   *              *                                *  *
[ read(busy) ][ write((busy, turn)) ][ read(turn) ][ write((busy, turn)) ] ...
      [ read(busy) ][ write((busy, turn)) ][ read(turn) ][ write((busy, turn)) ] ...
         *               *                                *  *
```

i.e. we could have the following infinite sequence:

$$read_i(busy), read_j(busy), write_i((busy, turn)), write_j((busy, turn)),$$
$$read_i(turn), write_i((busy, turn)), read_j(turn), write_j((busy, turn)),$$
$$read_i(turn), write_i((busy, turn)), read_j(turn), write_j((busy, turn)), ...$$

where `turn != me` always evaluates to `true`. In such an execution, both processes are stuck in entry forever.

(c)

The algorithm is not starvation-free because it is not even deadlock-free. This follows from the definitions.

P12

We only need to prove the following two lemmas. (1-process consensus is trivial.)

Lemma 1. *There is a wait-free consensus algorithm for 2 processes using* ***iplusplus***

Proof. Consider the following algorithm using an `iplusplus` object `I`:

```
Initially, Prefer ← [⊥, ⊥]
Each process i:
  Prefer[i] ← x_i
  if I++ == 0:
    output Prefer[i]
  else:
    output Prefer[1 − i]
```

Both processes decide x_k if process k increments `I` first.

□

Lemma 2. *There is no wait-free consensus algorithm for 3 processes using* ***iplusplus***

Proof. Suppose there is such an algorithm for 3 processes A, B, and C.

Consider the critical configuration S. Let O_A, O_B be the next operations of A and B, respectively. WLOG, suppose the next configuration is 0-valent if O_A is applied first and and 1-valent if O_B is applied first. O_A and O_B cannot be concurrent, otherwise both (O_A, O_B) and (O_B, O_A) would lead to the same configuration S' that must be both 1-valent and 0-valent. Therefore, the only possibility is that O_A and O_B both access the same object, and at least one of them updates the object. If they both access the same register `R`, then either:

1. **One reads from `R` and the other writes to `R`.** WLOG, suppose O_A is read and O_B is write. Let S' be the configuration after applying O_B and S'' the configuration after applying (O_A, O_B). (*) Suppose C runs solo starting from S', which is 1-valent, then it must decide 1. On the other hand, if it runs solo starting from S'', which is 0-valent, then it must decide 0. But S' and S'' are indistinguishable to C, so this is impossible.

2. **Both write to `R`.** Let S' be the configuration after applying O_B and S'' the configuration after applying (O_A, O_B). The rest of the argument is the same as (*).

If they both access the same `iplusplus` object `I`, then either:

1. **One reads `I` and the other increments `I`.** WLOG, suppose O_A is `read(I)` and O_B is `I++`. Let S' be the configuration after applying O_B and S'' the configuration after applying (O_A, O_B). The rest of the argument is the same as (*).

2. **Both increment `I`.** Let S' be the configuration after applying (O_B, O_A) and S'' the configuration after applying (O_A, O_B). The rest of the argument is the same as (*).

In all cases we arrive at contradictions, therefore such an algorithm does not exist. □