Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Prof. T. Roscoe, Prof. R. Wattenhofer

Computer Systems

— Solution to Assignment 10 —

1 Approximate Agreement

Basic

1.1 Modified Synchronous Algorithm

a) Yes. As there are at most $f < \frac{n}{3}$ byzantine nodes, in every iteration the median will lie in the interval of the correct nodes' values.

b) No. For instance, consider a network which consists of 11 nodes, 3 of which are byzantine. Suppose 4 correct nodes (group A) have the input 0, and 4 correct nodes (group B) have the input 1. In each iteration, the byzantine nodes send 0 to the nodes in group A, and 1 to the nodes in group B. Hence, the nodes in group A obtain the median 0, and the nodes in group B obtain the median 1. Since the correct nodes never change their values after any iteration, ε-Agreement fails, assuming $\varepsilon < 1$.

Basic

1.2 From Approximate Agreement to Byzantine Agreement

a) Yes. If all correct nodes have the same input bit b, correct-range validity ensures that all correct nodes obtain value $x = b$.

b) No. If the correct nodes have distinct input bits, then they can obtain any ε-close values in $[0, 1]$. It is possible that a correct node obtains $x = 0.5 - \varepsilon/3$, and therefore its final output is 0, while another correct node obtains $x = 0.5 + \varepsilon/3$ and therefore its final output is 1.

c) We set $\varepsilon = 1/2 \cdot 10^{-2024}$. The nodes join the approximate agreement algorithm with their input bits as initial values. When a node obtains a value x, it queries the shared coin. Once $f + 1$ nodes (hence at least one correct node v) query the coin, the random value r is decided and all nodes learn it eventually. When a node has obtained both the random value r and a value x via approximate agreement, if outputs 0 if $x < r$ and 1 otherwise.

The outputs x obtained via approximate agreement are ε-close, and agreement only fails if r is between the lowest and the highest values x obtained by correct nodes via approximate agreement. Then, if the first correct node that queries the coin has obtained value x, all correct nodes obtain outputs in the interval $[x - \varepsilon, x + \varepsilon]$. This means that only values $r \in [x - \varepsilon, x + \varepsilon]$ may lead to disagreement. Such a value r is obtained with probability at most $2 \cdot \varepsilon = 10^{-2024}$.

Advanced

1.3 Unbounded Input Space: Quick Fix

a) Nodes could simply define the number of iterations $I = \lceil \log_2((\max X - \min X)/\varepsilon) \rceil$.

b) Similarly to *correct-input validity*, one cannot distinguish between a correct node and a byzantine node that follows the algorithm correctly, but with an input of its own choice.

c) The algorithm proceeds as follows: every node sends its value to all nodes. Node v computes its estimation $\texttt{max\_range}_v$ as the difference between the highest value received (which is at least $\max X$, since all correct values were received), and the lowest value received (which is at most $\min X$, also because all correct values were received).

Note: removing the lowest f and the highest f values might discard some correct values and make the algorithm stop too early.

d) Nodes first run the algorithm from Task **c)**. Each node obtains an estimation $\texttt{max\_range}_v$. It computes $I_v = \lceil \log_2(\texttt{max\_range}_v/\varepsilon) \rceil$ as a sufficient number of iterations.

Then, the for loop goes up to I_v instead of the hardcoded number of iterations I. When node v computes its last value x_{I_v}, it sends a halting message $(\texttt{halt}, x_{I_v})$ to everyone. If node v receives a halting message $(\texttt{halt}, x_{I_u})$, it pretends it got x_{I_u} from u in each of its following iterations.

Algorithm 1 Synchronous Approximate Agreement: Unbounded Input Space

```
1: Code for node $v$ with input $x$.
2: Send $v$ to all nodes.
3: Add every received value to $X$.
4: $\texttt{max\_range}_v = \max X - \min X$.
5: $I_v = \lceil \log_2(\texttt{max\_range}_v/\varepsilon) \rceil$.
6: $x_0 = x$.
7: for $i$ in $1 \ldots I_v$ do
8:     Send $x_{i-1}$ to all nodes.
9:     Add every received value to $R_i$.
10:    If node $u$ sent $(\texttt{halt}, x_{I_u})$ (now or in some previous iteration), add $x_{I_u}$ to $R_i$.
11:    $T_i$ = the multiset obtained by removing the lowest $f$ and the highest $f$ values in $R_i$.
12:    $x_i = (\min T_i + \max T_i)/2$.
13: end for
14: Send $(\texttt{halt}, x_{I_u})$ to all the nodes.
15: Output $x_{I_u}$.
```

Let I_u denote the lowest correct estimation on the number of iterations, obtained by some correct node u. Since every correct node obtains $\texttt{max\_range}_v \geq \max X - \min X$, correct values obtained in iteration I_u already satisfy ε-Agreement (and correct-range validity). Then, once node u sends its $\texttt{halt}$ message, we only need to ensure that all the following iterations maintain the range of correct values obtained in iteration I_u (and don't need to guarantee anything about convergence). This can be proven with the help of Lemma 21.6.

e) With the mechanism above, not really. The values $\texttt{max\_range}_v$ are essentially chosen by the byzantine nodes.

2 Consistency and Logical Clocks

2.1 Different Consistencies

a) This is a direct corollary of Lemma 22.10, Lemma 22.12 and Lemma 22.13: Assume for the sake of contradiction that sequential consistency implies linearizability. Since according to Lemma 22.12 linearizability implies quiescent consistency this would mean that sequential consistency implies quiescent consistency which is a contradiction to Lemma 22.13. An analogous derivation can be done to disprove that quiescent consistency implies linearizability.

b) We disprove the statement by a contradicting example: Assume that we have three nodes u, v and w that each execute one operation on the same object with initial value 2. u and v execute *inc* (increment the object by 1) and w executes *double* (multiply the object by 2) with $inc^v_* < inc^u_\dagger < double^w_* < inc^v_\dagger$. Since there is no quiescent point and all nodes only have one operation, quiescent and sequential consistency are both fine with a sequential execution that doubles first and then increments: $((2 \cdot 2) + 1) + 1 = 6$. Linearizablility on the other hand requires $inc^u < double^w$ to be in this order in S.

2.2 Measure of Concurrency from Vector Clocks

Here the Python code:

```
# some random possible state of the system after one message was sent from node
    ↪ 1 to node 0
log0 = [[1, 5]] # list of logged vector clocks of node 0
log1 = [] # list of logged vector clocks of node 1
c0 = [20, 5] # current state of vector clock at node 0
c1 = [0, 10] # current state of vector clock at node 1

muS = c0[0] + c1[1] + 1
muC = (1 + c0[0]) * (1 + c1[1])

# number of consistent snapshots before receiving the message:
mu1 = (log0[0][0]) * (1 + c1[1])

# number of consistent snapshots after receiving the message:
mu2 = (1 + c0[0] - log0[0][0]) * (1 + c1[1] - log0[0][1])

# total number of consistent snapshots:
mu = mu1 + mu2

measureOfConcurrency = (mu - muS) / (muC - muS)
print(measureOfConcurrency)
```

And here the more general code:

```
# some random possible state of the system
log0 = [[1, 13], [14, 26]] # list of logged vector clocks of node 0
log1 = [[13, 14]] # list of logged vector clocks of node 1
c0 = [30, 26] # current state of vector clock at node 0
c1 = [13, 26] # current state of vector clock at node 1



muS = c0[0] + c1[1] + 1
muC = (1 + c0[0]) * (1 + c1[1])
```

```
# terminating is equivalent to receiving a message in the next operation for
    ↪ this calculation
terminalClock = [c0[0] + 1, c1[1] + 1]
log0.append(terminalClock)
log1.append(terminalClock)

# start of uninterrupted execution:
start0 = 0
start1 = 0

# end of uniterrupted execution:
nextReceive1 = log1[0][1]

mu = 0
idx1 = 0
for l0 in log0:
  mu += (l0[0] - start0) * (nextReceive1 - start1)
  # go through every message received by 1 before the next message received by 0
  for idx in range(len(log1) - 1):
    if start0 < log1[idx][0] < l0[0]:
      lastReceive1 = nextReceive1
      nextReceive1 = log1[idx + 1][1]
      mu += (l0[0] - log1[idx][0]) * (nextReceive1 - lastReceive1)
  # go to next message received by 0
  start0 = l0[0]
  start1 = l0[1]

measureOfConcurrency = (mu - muS) / (muC - muS)
print(measureOfConcurrency)
```