Problem Set 4

Due before midnight on Monday, October 29, 2018.

1 Consensus Problem

In this and following assignments, we will be developing a simulator for a distributed consensus algorithm. Consensus is at the heart of maintaining consistency in distributed databases as well as in cryptocurrencies and blockchain algorithms.

We consider the *consensus problem* in a simple setting. The players are trying to reach agreement on a course of action. Each player has a current preference called her *choice*, which is the current value stored in her *choice register*. We assume a simple binary choice, so the choice value is either 0 or 1. The players communicate with each other, and from time to time a player may change her choice. The goal is for the players to arrive at a stage where all players are making the same choice. In this case, we say the players have *reached consensus*, and we call the common choice the *consensus value*. We also require that the consensus value be *stable*, meaning that once consensus has been reached, nobody can subsequently ever change her choice.

We assume the agents communicate using the *random-pair* communication model. In this model, a *communication round* consists of a randomly chosen player (called the *sender*) sending a message to another randomly chosen player (called the *receiver*). Sender and receiver must be distinct. For the algorithms considered here, the sender's message is always her current choice value. The receiver, depending on the message received, may change her current choice and her internal state.

A population of players *solves* the consensus problem if following is true:

1. For all possible initial choices of the players, if the players start in their designated initial states, the computation eventually reaches consensus with probability 1.

2. Once consensus has been reached, no player can subsequently ever change her choice.

Note that if all players start with the same choice, then that choice is the consensus value, and no player ever changes her choice.

There are many possible algorithms for reaching consensus. Here are a couple very simple ones that we will be exploring.

1.1 Fickle

Whenever a *fickle* player receives a message, she changes her choice, if necessary, to agree with the sender's choice. That is, she sets her choice register to the value contained in the message.

It is easy to see that there is some sequence of message transmissions that causes the system to reach consensus. Once consensus has been reached, no player will change her choice since every subsequent message contains the consensus value.

It is also easy to believe that it might take a very large number of random communication rounds to reach consensus.

1.2 Follow the Crowd

A *follow-the-crowd* player has a one-bit state register in which she saves the last message received. She changes her choice only when she gets *two* messages in a row that both disagree with her current choice. Thus, she waits until she gets a sense of the crowd before deciding to follow. We assume that each player starts with her state register set equal to her choice.

In greater detail, when a follow-the-crowd player receives a message m, she compares it with her current state. If it differs, she replaces the current state with m. If it is the same, she replaces the current choice with m.

It is believable that this might converge to a consensus value faster than *fickle* since it is less likely for a player holding the majority choice to change to the minority value.

2 Assignment Goals

1. To learn how to organize a simulation of a large system.
2. To learn about a simple model of asynchronous distributed computing.
3. To learn how to generate uniformly distributed random numbers from a finite interval.
4. To experience a computationally-intensive application where efficiency matters.

3 Problem

In this assignment, you will implement a simulation of a large number of agents attempting to reach consensus using the *fickle* algorithm under the random-pair communication model. The *follow-the-crowd* algorithm will be used in a later assignment.

You are required to implement two classes and a main program.

- `class Agent` models an agent running the *fickle* algorithm. The public interface must support these functions:
  - `Agent(int ch)` constructs an agent with choice `ch`.
  - `void update(int m)` performs the update to the agent as specified by algorithm *fickle* upon receipt of the message `m`.
  - `int choice() const` returns the agent's current choice.
- `class Simulator` simulates a collection of n agents trying to reach consensus using the random communication model described in section 1. Its public interface consists of the following:
  - `Simulator( int numAgents, int numOne, unsigned int seed )` constructs a simulator for `numAgents` agents. The first `numOne` of these have initial choice 1; the remainder have initial choice 0. `seed` is used to initialize the random number generator `random()`.
  - `int run( int& rounds )` runs the simulation for as many rounds as it takes to reach consensus. The number of communication rounds used is stored in the output parameter `rounds`. The consensus value is returned.

To carry out the simulation requires the ability to select a random pair of distinct agents j and k to serve as sender and receiver in a communication round. Further details are given in section 4.

To know when consensus is reached requires the simulator to keep track of the number of agents having each of the two possible choice values. Since the only way an agent might change its choice is because of `update()`, you should just update the counts of agents having a given choice after each communication round. Do *not* poll every agent after every round.

- `main.cpp` implements a command

  `> consensus numAgents numOne [seed]`

  that takes two required argumets, `numAgents` and `numOne`, and one optional argument, `seed`. These three arguments should be converted to numbers and passed to the `Simulator` constructor. If `seed` is omitted, the result of `time(0)` should be used instead.

  The `run()` function in `main.cpp` should instatiate a `Simulator` with the given parameters and then run it. When `Simulator::run()` returns, you should print a single line to `cout` consisting of five whitespace-separated numbers: The number of agents, the number of agents initially choosing one, the actual seed used, the number of communication rounds required to reach consensus, and the final consensus value.

You should test your code using various combinations of parameters. Increasing the population size `numPlayers` will cause a big increase in run time as will having `numOne` be close to `numPlayers/2`. You may terminate your experiments once the run time grows to more than a few seconds. This may come rather quickly with *fickle*, but that is for you to find out. As usual, you should submit test input files and the corresponding outputs produce by your program.

4 Program Notes

I will furnish some test cases on the Zoo in `/c/cs427/code/ps4/`, and you should test it with some parameter combinations of your own. However, you might only be able to duplicate my output if you run your code on the Zoo and your program uses the random number generator in the same way, namely, at each round, first select the sender and then select the receiver.

To select a random sender from among n agents, you can use the function `RandomUniform(n)`, which returns a uniformly-distributed random integer in the range $[0 \ldots n-1]$.

```
int RandomUniform( int n ) {
  long int usefulMax = RAND_MAX - (RAND_MAX+1)%n;
  long int r;
  do { r = random(); }
  while ( r > usefulMax );
  return r % n;
}
```

The purpose of this code is to make all numbers in the given range equally likely.[1]

To make sure you use the same number of calls on the random number generator as I do when choosing the sender and receiver, you should first chose the sender j from among the n agents. Now there are only $n-1$ eligible receivers, so you should choose a a number k in the range $[0 \ldots n-2]$ and adjust k to avoid j by incrementing k if $k \geq j$.

The submission guidelines are the same as in previous assignments. Submit all files needed to compile your project along with a `Makefile`. Include a `notes.txt` file, a file of sample inputs and a file of the corresponding outputs.

5 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1 (see below).

#	Pts.	Item
1.	4	All relevant standards from previous problem sets are followed regarding submission, identification of authorship on all files, and so forth. A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++17`. Running `make` successfully compiles and links the project and results in an executable file `consensus`. Each function definition is preceded by a comment that describes clearly what it does.
2.	2	Required sample input and output files are submitted.
3.	4	The program shows good style. All functions are clean and concise. Inline initializations, inline functions, and `const` are used where appropriate. Variable names are appropriate to the context. Programs are consistently indented according to the course indenting style. Each class has a separate `.hpp` file and, if needed, a separate `.cpp` file.
4.	2	Everything is private in all classes except for the specified public interface and any needed special functions (constructors, destructor, move and copy constructors and assignments).
5.	8	All of the functionality in section 3 is correctly implemented.
	20	Total points.

Figure 1: Grading rubric.

[1] Note that it is not sufficient to just take `random()%n` since if n does not divide `RAND_MAX + 1`, some numbers will have a greater probability of being chosen than others.

For example, if `random()` were to produce numbers in the range $[0 \ldots 9]$ and we wanted numbers in the range $[0 \ldots 3]$, then reducing each of the numbers in the range $[0 \ldots, 9]$ mod 4 gives the sequence $0, 1, 2, 3, 0, 1, 2, 3, 0, 1$. We see that 0 and 1 each occur 3 times, whereas 2 and 3 each occur only twice. Thus, 0 and 1 are each generated with probability 0.3 and 2 and 3 are each generated with probability only 0.2. To be uniformly distributed, all probabilities should be 0.25. In this example, where we pretend `RAND_MAX==9` and $n = 4$, my code computes `usefulMax` to be `9 - 10%4 = 7`. Then whenever `random()` returns 8 or 9, the program loops and tries again.