Data Structures and Functional Programming	**Problem Set 6**
CS 3110, Spring 2016	
Version: 2	**Last Modified: May 5, 2016**

Objectives

- Write asynchronous programs in an event-driven style.
- Learn about the 2/3 (Two-thirds) Consensus Protocol
- Develop a small distributed system.
- Design a small but sophisticated piece of software.

Changes and Errata

- (May 2) There should be no event called Error.
- (May 3) In figure 1, there should not be a horizontal line in the second round between replica 1 and 2.
- (May 4) In step (d) of the algorithm description, majority is $(f+1)$, not $(2f+1)$. Note majority is different from unanimity.
- (May 4) There should be one more property that a consensus protocol should satisfy, which is **liveness**, now described in Part Two. The pseudo-code in Part Two and the instruction in Part Three (Exercise 2) are changed to reflect the strategy for achieving liveness.
- (May 4) The third column in the `localConfig.txt` file now specifies the round of crash, not just whether it crashes. If the number is $k > 0$, the replica will crash in round $k-1$.
- (May 4) More examples are given in Part Six on some patterns that may be useful for this assignment
- (May 4) Remote Controller is released in the release code and the instruction to use it is included in Part Four.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: Lecture 22, 23, 24
- CS 3110's Async Documentation

- Jane Street's Async documentation
- The CS 3110 style guide
- Real World OCaml, Chapter 18

Instructions

Compile Errors

All code you submit must compile. **Programs that do not compile will be heavily penalized.**

Naming

We will be using an automatic grading script, so it is **crucial** that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions are **treated as compile errors** and you will have to submit a patch.

Code Style

Finally, please pay attention to style. Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct may still lose points. Take the extra time to think out the problems and find the most elegant solutions before coding them up. Good programming style is important for all assignments throughout the semester.

Late Assignments

Note that **we do not accept late submissions** this time, since it is due on the last day of class, and need to be graded as soon as possible.

What to Submit

You should submit the following files to CMS.

- `replica.ml` containing your implementation of the replica.
- `README.txt` describing your implementation strategy and data structure used.

Part One: Project overview

In this problem set you will be implementing the 2/3 (Two-thirds) Consensus Protocol. Specifically, you will be responsible to implement the logic of each individual replica in the network. While networking is always involved in real world applications of consensus, in this assignment you will focus on the logic to reach consensus, and do not need to worry overly about networking.

Part Two: 2/3 Consensus Protocol

Consensus Protocols are used to achieve agreement across multiple machines over the network. In data centers, data tables are replicated on multiple machines, called **replicas** to ensure persistence in the face of machine failure. When operations on data are conducted, replicas have to agree on which operation to execute.

Consensus protocols are methods by which replicas agree on some to be determined value, which could be queries, data contents, or something else. Replicas each propose values and they will try to agree on one value using an algorithm, which is the protocol. A wide variety of consensus protocols exist. In this assignment, we will implement the 2/3 Consensus Protocol, also called the two-thirds consensus protocol.

It is simple to reach consensus if all replicas have guaranteed connection to all other servers, as shown in lecture. It is more difficult to achieve consensus in the face of failure. The 2/3 Consensus Protocol is a fault tolerant protocol.

The **validity property** of consensus requires that the selected value must belong to the set of input values. Secondly, the **agreement condition** requires that all replicas need to agree on exactly one value when the protocol terminates.

For the data center to work correctly, a consensus protocol should run within a finite amount of time. This condition is called the **termination condition**. We have seen in lecture that the termination condition is not always guaranteed to hold. It is also essential that all replicas that have not crashed should make forward progress, and that replicas should not be stuck in deadlock waiting for each others' vote and unable to generate new votes. This condition is called the **liveness** condition. Note that liveness is different from termination, because replicas can keep making forward progress (thus alive) without being about to come to a consensus (thus nonterminating).

In the 2/3 Consensus Protocol that tolerates f failures, exactly $3f+1$ replicas are created. Each replica decides on a value using a simple voting procedure in rounds. The replica can make a definite decision when $2f+1$ replicas agree on a value. This means that each replica can still make decision if no more than f of other replicas have crashed. The 2/3 Consensus Protocol guarantees that if it terminates, all replicas will agree on the same value that is proposed by some replica (logical property and agreement is guaranteed). The algorithm for each replica to reach a decision is fairly simple (living up to its name) and is as follows (also see lecture 22):

1. Suppose there are a total of $3f+1$ replicas, for each replica
2. Start with $r=0$ and $v=$ initial vote
3. while decision is not made
   (a) Broadcast (r, v) to all reachable replicas
   (b) **Collect and store** all votes from replicas, even if they are for a later round
   (c) If $2f+1$ votes are collected for round r, use the procedure in (d), (e), (f) to decide the vote for round $r+1$
   (d) If there is a unanimous agreement on value v' among the $2f+1$ votes, make a final decision on value v'
   (e) Otherwise, if there is a majority value, i.e. a value with no less than $f+1$ votes, update v to the majority vote
   (f) If there is no majority, the replica is allowed to select any value for v that is among the voted values
   (g) increment r by 1

Example Figure 1 describes how 4 machines decide on the value of a data entry.

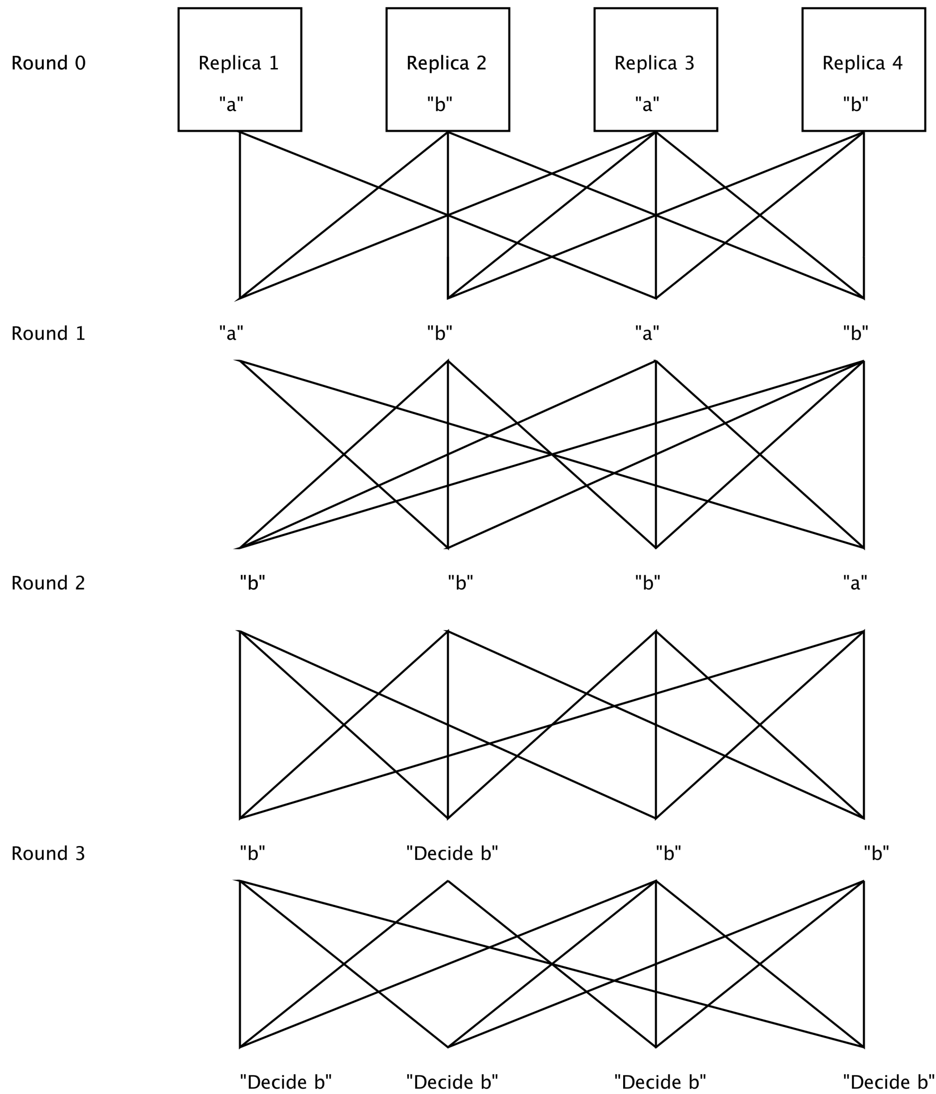

Figure 1: Execution of 2/3 Consensus Protocol with 4 replicas

Part Three: Implementation Task

Background

The code to set up network connection between replicas is already written for you. Your protocol will assume that the connections are all established at the start of the protocol. However, there may be machine crashes in the duration of the protocol, and your code need to work correctly facing up to f failures.

Exercise 1: Understand Existing Code

`Replica.mli` specifies the modules each replica needs to communicate with other replicas "over the network". The modules `Protocol` and `Config` are inputs to each replica, and the module `Replica` executes the voting logic.

Module `Protocol` specifies the data channel and data type used in the communication between replicas. The connections are represented as reader/writer pairs, and the utility functions `Protocol.send` and `Protocol.receive` are used to work with the readers and writers.

The messages transmitted across the network are defined in `Protocol` as a variant. You can conceptualize the variants as events happening during the protocol. The Connection event happens when a Connection is established. The Vote event happens when you receive a vote from other replica with a round number and a value. Remember that your vote in round $r+1$ should only be affected by votes in the round r. Lastly, the Decide event happens when some replica has already made a decision on the values.

Notice that `Protocol.receive` returns value with type `[‘Ok of ’a msg |‘Eof] Deferred.t`. The variants `‘Ok` and `‘Eof` are indications of whether the reading is successful, where `‘Ok` means something is successfully read, and `‘Eof` means that nothing can be read from the reader. Note that `‘Eof` only happens when there is something wrong with the channel, not when you are simply waiting for responses from the channel.

Since the returned value is a deferred value, you need to work with the `Async` module closely for this assignment. You will need to use `Async` functions `bind (>>=)`, `upon`, `return` extensively. Note that the `Async` module also provides functions that work with lists.

Module `Config` contains the metadata about each replica. The module contains information about the machine id, the total number of replicas, etc. You will find these information useful in implementing the voting logic. You are given a *logger* in `Config`, and you can use the logger and the `Protocol.to_string` function to output formatted messages on screen.

You are given a local simulation controller in `localController.ml`, which simulates network connection using a single process. You are also given a network controller in `remoteController.ml`, which connect to each other via TCP. See Part 4 for detailed instruction on setting up and cleaning up the replicas and controllers.

Exercise 2: Implement 2/3 Consensus Protocol

You should implement the logic of each replica to **vote**, **collect** and **decide** in `replica.ml`.

In `replica.ml` you need to implement the functor `Replica.Make`, and specifically the function `S.run`. `Replica.Make` takes one Protocol module that is described in Exercise 2, which handles sending and receiving messages to and from other replicas. `Replica.Make` also takes a `Config` module, and you should use the information provided in your implementation.

`S.run` takes a list of connections with other replicas, and an Ivar to be filled in. The replica should conduct the voting logic described in Part Two until it makes a decision. When the replica makes a decision, it should fill in the Ivar with that decision.

The decision filled inside the Ivar will be logged into the file `consensus.out`. You can examine the file and see if it satisfies the logical and agreement property.

Make sure you do not double count votes in each round. The vote for each round from each

machine should only be counted once, even when in your implementation they can be received multiple times.

Make sure you ensure the **liveness** of the protocol. This means that replicas that have not crashed should be able to make forward progress, instead of getting stuck in mutual deadlock. One potential reason of deadlock is that votes from a later round are ignored by a replica that is still running the previous round. In this case, when the replica gets to the later round, it cannot receive enough votes since some votes about this round have been ignored by it.

We do not specify how you actually collect and count votes in each round, and therefore you have the freedom to choose the implementation and underlying data structure to implement this consensus protocol. However, you are required to fill in the Ivar once the replica has made a decision. Our tests will look at the file `consensus.out`.

Exercise 3: Written Part

In README.txt, describe your strategy to implement the voting process for the replica. Make sure you outline the logic and mention any data structures you use.

Part Four: Compilation and running the protocol

To save the effort to compile, run and clean up your replica, we provided a `make` file in the release code called `Makefile`. This is meant to make compilation and running easier for you. You can still use the cs3110 tools to compile and run your code individually. To compile your code, use the following commands.

- `make`: to compile all files
- `make replica`: to compile `replica.ml`
- `make local`: to compile `localController.ml` and dependencies.
- `make clean`: to discard all compiled files.

If you are running the local controller, you need to use the file `localConfig.txt` in the following way:

- In the first line you should fill in number f that is the **number of faults** tolerated.
- For the next $3f+1$ lines, each line contains three integers.
- The first integer is the initial value this replica has.
- The second one is the network delay (in seconds) this replica is supposed to have
- The third one is whether this replica is supposed to crash. If the third field has value $k > 0$, the replica will "crash" in round $k-1$. It will not crash if the field is 0 or negative. You can use the delay and crash to simulate real network situation and test fault tolerance of your code.

- After setting the `localConfig.txt` file, run `make runlocal` to start the simulation.

Since the simulation is local, the program will exit itself when all $3f+1$ Ivars are filled. You can view the decisions in `consensus.out`.

To clean up server log files and the output, run

```
make cleanlogs
```

To help with setting up multiple servers in the remote setting, we provided files `addresses.txt` and `config.txt` to use with the remote controller. `addresses.txt` contains the addresses of the servers, and you can modify the number of servers in this file (make sure the number is $3f+1$ for some integer f). `config.txt` contains the initial values, the delay time, and the round after which the server crashes. The number of lines in `config.txt` should match that of `addresses.txt`. You can change the values to test for different initial values. To get the servers running, use the command

```
make runremote
```

You may encounter an error saying that the file `run.sh` is not executable. In that case, run the command

```
chmod +x run.sh
```

to make the script executable before you run it. To terminate the server and free up socket usage, run

```
make killservers
```

To clean up server log files and the output, run

```
make cleanlogs
```

If you want to compile the files yourself, for each file you should do

```
cs3110 compile -thread -p async,str <filename>.ml
```

The remote controller takes two arguments if run individually:

```
cs3110 run remoteController.ml <index> <initial_value>
```

If you want to compile the files yourself, for each file you should do

```
cs3110 compile -thread -p async,str <filename>.ml
```

The local controller takes a config file described above as `localConfig.txt`:

```
cs3110 run localController.ml localConfig.txt
```

Part Five: Optional Karma Problems

Karma questions are for you to have fun and further your understanding in the topic. It will not affect the grade of this assignment.

In lecture we have learned that 2/3 consensus protocol can be forced to not terminate using a denial of service attack. Write another module of module type `Replica.S` that, when used along with other $3f$ normal replicas in the 2/3 protocol, prevents this protocol from termination.

Note that the prevention of termination need to be done while maintaining **liveness**, i.e. each server should still make progress (collect votes, vote for the next round), and should not be in deadlock or crash.

Part Six: Hints and Tips

1. Since there is no way to know whether other replicas have decided, your `run` function needs to keep responding even after it has made a decision. Note that local controller does exit when every replica has made a decision.

2. The 2/3 consensus protocol does not necessarily halt and decide, since after each round it is possible for them to end up in exactly the same state as the previous round. However, due to delays in network connections, the protocol cannot stay in "lock steps" forever and will eventually terminate. We have injected random delays in the controllers to simulate real world connection, and your protocol should eventually halt and decide if it is correct.

3. The Jane Street's Async module documentation can be difficult to navigate, and therefore we have our own CS3110 version of documentation, as listed in Recommended Reading. The CS3110 documentation should be sufficient for the purpose of this assignment.

4. When working with the async library, make sure to keep your code readable. A preferred style to use the bind `(>>=)` operator is to put "`>>= fun x ->`" at the end of the line, and the body of the callback in the next line. For example:
   ```
   e1 >>= fun x ->
   e2 >>= fun y ->
   e3 >>= ...
   ```

5. Learn to use the async programming paradigm. An async program should be organized using callbacks and the bind, upon, return functions. Prevent spinning and polling (i.e. looping infinitely to wait for a value). Here we give two examples of async programming patterns that may be useful.

   The first example is an echo-er that prints the messages it receives from a single source, and get back to listening. You can relate this to the replica doing something with the message read from one reader, and then go back to listening.

```
(* echo function, listen back to the same source *)
let rec echo (source : unit -> string Deferred.t) =
  source () >>= fun message ->  (* wait for message *)
  print_endline message;        (* do something interesting with it *)
  echo source                   (* go back to listen again *)
```

The second example is a multi-echoer: it listens to multiple sources, echo every message it receives. However, the multi-echoer also stop echoing once it has echoed 100 times. This requires a mutual state. You can relate this to vote-counting in replica.

```
(* echo 100 times and then print DONE *)
let echo_all (source_list : (unit -> string Deferred.t) list) =
  let count = ref 0 in       (* initialize a mutual counter *)
  let rec echo_and_count source =
    source () >>= fun msg ->
        count := !count + 1; (* update the counter *)
        if !count = 100 then print_endline "We are done." else ();
        if !count >= 100 then return () else
          (print_endline msg; echo_and_count source)
  in
  let list_to_wait = List.map echo_and_count source_list in
  (* wait for all the echo-ers to finish *)
  Deferred.List.all list_to_wait
```

6. Note that **we do not accept late submissions** this time, since it is due on the last day of class, and need to be graded as soon as possible.