Problem Set 1

CSE 452 / CSE M552

April 14, 2017

Submit short, typeset answers to the following questions. Please work on this individually.

Problem 1: Haiku MapReduce

Suppose we have 3 mappers and 3 reducers, and we use them to do word count on the following haiku:

```
how much can i fit
into a haiku format
now i am out of
```

Each mapper gets one line of the haiku, and the partition function is super-simple: the value of the first letter of the word in ASCII, modulo the number of reducers. For example, all words that begin with "a" (ASCII 97) and "d" (ASCII 100) will be sent to the same reducer.

List the key-value pairs that each mapper produces, the key-value pairs that each reducer processes, and the final output of each reducer.

Problem 2: A Counter of Questionable Safety

Someone has given you the following Golang implementation of a concurrent counter:

```
type PossiblySafeCounter struct {
  mu sync.Mutex
  sum int
}

func (c *PossiblySafeCounter) inc() {
  c.mu.Lock();
  defer c.mu.Unlock();

  go func() {
    c.sum++
  }()
}

func (c *PossiblySafeCounter) read() int {
  c.mu.Lock();
  defer c.mu.Unlock();

  return c.sum
}
```

You want to increment it from multiple concurrent threads. Will you have data races? Why or why not?

Problem 3: At Most Once

As we explained in class, to implement at most once RPC, we need a unique message ID, such that the client agrees to never reuse a message ID (for a different message) and the server agrees to always remember if the request corresponding to the message ID has been reused. Explain what would happen if we removed (a) the client constraint (a message ID could be reused) or (b) the server constraint (remembering past message ID's).

Problem 4: Primary, Backup, and Judge (PB&J)

You have an implementation of the Primary/Backup system discussed in class, but you have lost its source code. You can't remember, but you're worried that there might be a bug in the primary: it will sometimes send an acknowledgment to the view server *before* transferring its state to the backup. Write a (pseudocode) client and specify a sequence of node failures and message drops in order to find out whether the primary is in fact buggy.