School of Computer Science, McGill University

COMP-512 Distributed Systems

Assignment 1; Due on 20-October 10am

Please be aware that students have to work individually on this assignment.

Exercise 1: Minis (20 Points)

1. (5 Points) Software clocks

   Assume an asynchronous system and a process p requests the time from a time server S. The measured round trip time is 24 ms, and p receives $t = 10:54:23:674 (hr:min:sec:ms)$ from S.

   (a) *According to the lecture notes on time, to what time does p set its local clock?*

   (b) *Assume now that ps current time, just before resetting it, shows* $10:54:24:005$. *What is the problem if p performs this reset? How can this be resolved?*

2. (6 Points) In class we discussed three different ways to marshal data into messages: XML (Json), Java's serialization, and Google Protocol Buffer

   For each of the techniques, give an advantage it has over the others.

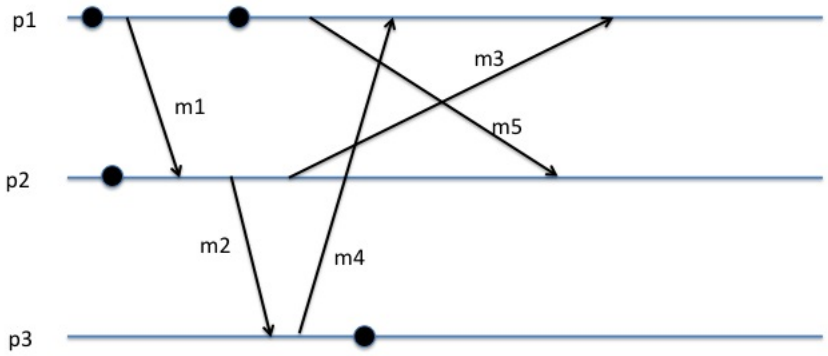
3. (9 Points) Given the below events (dot = local event) in a system.

   Timestamp each event (local, send, receive) with

   (a) (4 points) Lamport timestamps

   (b) (5 points) Vector timestamps

   You can assume that at the beginning all clocks are 0 resp. (0,0,0).



Exercise 2: Eventual Consistency (35 points)

Assume a set of N processes each having a copy of a specific data item X. Each process can access the local copy, reading and writing it. The processes are only loosely connected to each other. That is, connections are continuously established and released (e.g., mobile network). Whenever two processes build a connection, they exchange the state of their copies of X. Furthermore, whenever there is a stable connection between two processes for a long time, these two processes exchange the state of their copies every time interval Y. *You have to develop an algorithm that guarantees eventual consistency.* Eventual consistency means that once there are no updates for a sufficiently long time and all processes eventually exchange their information, all processes will have the same state for X.

To illustrate the behavior, let's have a look at two examples.

1. Assume that processes $p1$ and $p2$ update X concurrently. Eventually all should have the value written by $p1$ (process with lower identifier). The update of $p2$ should not be reflected in any copy.
2. Assume that process $p1$ writes X and sends the update to $p2$. $p2$ installs the update. Now $p2$ also updates X. Eventually all processes should have the final value written by $p2$. That is, if a process $p3$ first receives $p1$'s update and then $p2$'s update, it installs them in this order. If it receives first $p2$'s update, it installs it. If it later receives $p1$'s update, it discards it.

Your solution should use a vector clock to resolve the issue. Consider the relevant events in the system (e.g., receiving an update request from the application, getting connected with another peer which initiates the exchange, receiving the state of X from another process, time interval Y has elapsed, ... You can assume that at start time all processes have a copy of X with the same initial value.

Note: WinFS (developed by Microsoft) allowed for replication of data and had to tackle the exact above problem. They did not exactly use vector clocks but something conceptually very similar.

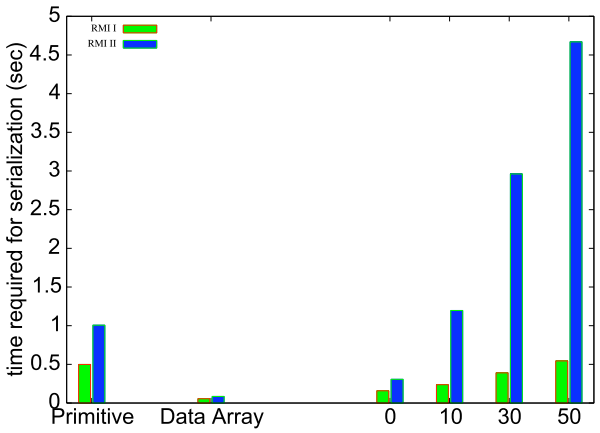
Exercise 3: Performance Evaluation (45 Points)

Given two object-oriented programming languages, each with its own implementation of RMI. In the following figures, the two RMI implementations are denoted as RMI I and RMI II. Object and data serialization for message marshalling is language dependent. Both RMI I and RMI II support communication over TCP sockets and over HTTP (which itself is implemented as a layer on top of TCP). The HTTP version of RMI I creates a connection for each remote method invocation. In all other cases, connections are reused. That is, one time a socket connection is created between the client and the server and this connection is used for all calls of the experiment. Three different method invocation tests are conducted. The first test calls a simple method with a single primitive parameter (integer) 50,000 times and no return value. This represents probably the most common remote method invocation type. The actual data transferred is very small. The second test makes 10 calls to a method with a single parameter which is an array of half a million characters. Again, the method has no return value. This evaluates bulk transfer. The third test makes a few hundred invocations to a method that has as parameter as well as return value a complex object. The object itself consists of a list of embedded objects (on average 175 embedded objects). Each embedded object consists of a number of doubles. In the test this number is varied from 0 to 50. Both the invocation as well as generating the response requires to serialize the complex object.

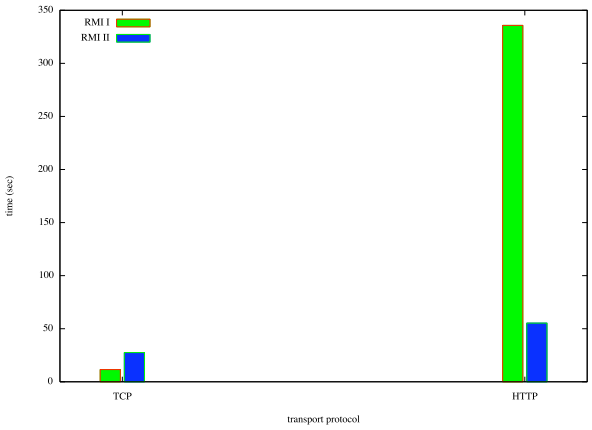
Below figures show the following. Figure 1(a) shows the time needed purely for serialization in the different tests for RMI I (green) and RMI II (blue). 0, 10, 30, 50 refer to the number of doubles in each embedded object for the third test. Figure 1(b) show the execution times for RMI I and RMI II for both TCP sockets and HTTP for the test with the primitive data type. Figure 1(c) shows the corresponding information for the invocation with a char array. Figure 1(d) shows the execution times for RMI I with TCP (green) and HTTP (pink), for RMI II with TCP (blue) and HTTP (brown) for 0, 10, 30 and 50 doubles in the embedded objects.

1. *For each of the figures describe all the observations that you can make. For performance differences, you have to decide whether they are significant or "noise".*

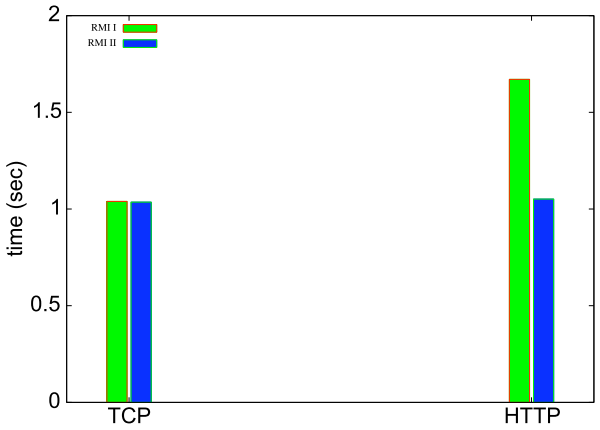
2. *Attempt to find explanations for the behavior and any type of performance difference.*



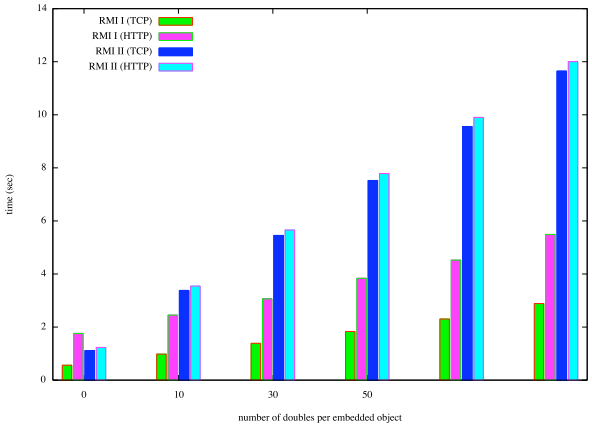
(a) Serialization Times



(b) Method call with Primitive Data Type



(c) Method call with Char Array



(b) Method call with Complex Object

Figure 1: RMI Analysis