CMSC 327 Distributed Systems

# Project 3: Clock Synchronization
# Due October 27, 2010

This project asks you to implement and evaluate a clock synchronization algorithm. Your program will only have access to a local clock that drifts quite a bit. Luckily, you will have access to a clock server with a much more accurate clock. We are going to use Java's Remote Method Invocation (RMI) capability to access the clock server. Java RMI let's you invoke methods on an object living in a different virtual machine, possibly running on the same computer or across the Internet.

## Learning Objectives

◦ Explore OO-Distributed Architectures ◦ Use Java RMI ◦ Implement a Clock Synchronization Algorithm

## Deliverables

Submit a zip file with the following directory structure:

`cmsc327_proj3_LASTNAME_FIRSTNAME/`

| | |
|---|---|
| `src/ClockServerRunner.java` | creates a ClockServer object and register it with the NameServer |
| `src/LocalClockEvaluator.java` | evaluates the provided local clock |
| `src/ImprovedLocalClock.java` | your clock synchronization program |
| `README` | simple text file explaining how to run your code |
| `results.pdf` | your evaluation |

## Resources

Various Java RMI tutorials can be found online[1][2].

## Design

Explain (in prose and pictures, not code) the design of your synchronization algorithm. Why did you decide to choose this algorithm? Did you modify the algorithm at all? What parameters can you change to tune the performance of the algorithm? In what situations is your algorithm applicable?

## Evaluation

Collect data to measure how well your clock synchronization algorithm works. Your first task is to evaluate the accuracy of the local clock class provided. Using the ground truth data from the server, characterize it's error. You should consider reporting the statistics about this error over a long period of time (e.g. minutes). The mean and standard deviation of the error is of interest, but also its long-term behavior. Is the error bounded? How fast does it grow? After you have implemented your clock synchronization algorithm, compare its accuracy to that of the original local clock class. Finally, reflect on the trade-off between the accuracy of your program and the frequency of server queries.

[1] http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html
[2] http://download.oracle.com/javase/tutorial/rmi/index.html

## Implementation

You are given four Java classes (`Clock.class`, `ClockMessage.class`, `ClockServer.class`, and `LocalClock.class`). `ClockServer` provides an accurate estimate of the time. It's exact implementation is unimportant for this assignment. The only important aspect of `ClockServer` is that it implements the Clock interface. The Clock interface has only three methods: `getTime()`, `getTaggedTime()`, and `getGroundTruth()`. The first two should be used for synchronization purposes, and the third should **only** be used for evaluating your program. `getTime()` and `getTaggedTime()` return the time in milliseconds since the epoch. The second method not only returns the time, but also the times when the request was received and the reply was sent.

```
public class ClockServer extends UnicastRemoteObject implements Clock{
}

public interface Clock extends Remote{

    /**
     * @ return this clock's estimate of the time in milliseconds since January 1, 1970
     */
    public long getTime() throws RemoteException;

    /**
     *
     * @ return a pair of timed Values, when the message was received and when it was sent
     *   (time in milliseconds since January 1, 1970)
     */
    public ClockMessage getTaggedTime() throws RemoteException;

    /**
     * @return the "ground truth" time in milliseconds since January 1, 1970
     */
    public long getGroundTruth() throws RemoteException;
}

public class ClockMessage implements Serializable{

    public long in;   // when the request was received
    public long time; // the time at the server
    public long out;  // when the reply was sent

    public ClockMessage(long in, long time, long out)
    {
        this.in = in;
        this.time = time;
        this.out = out;
    }
}
```

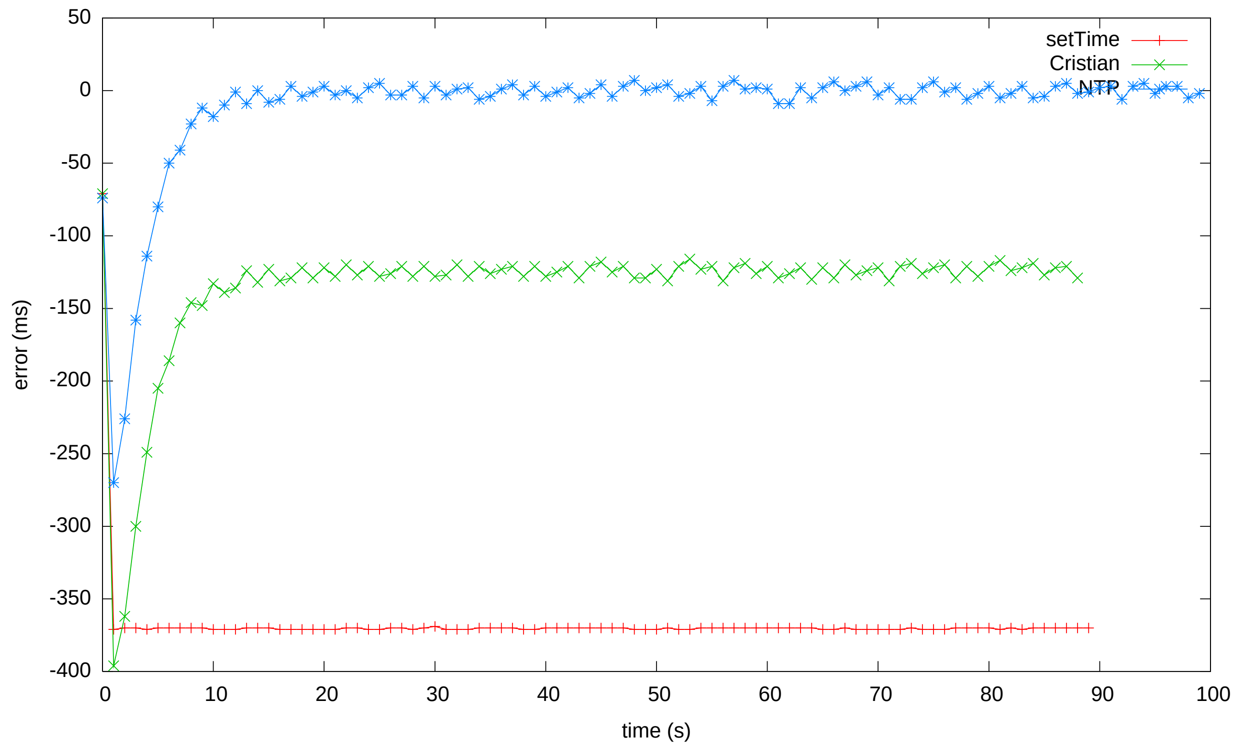
You should create a `ClockServerRunner` class that when run creates a `ClockServer` object and registers it with a NameServer so it can be accessed by your clients.

The `LocalClock` serves as the inaccurate clock and will act as the parent class of your more accurate clock.

```
public class LocalClock extends TimerTask {
    /**
     * current time in microseconds since the epoch
     */
    private long time;

    /**
     * how much the clock ticks per interrupt
     */
    private long tick = 100;

    /**
     * Create a new local clock
     */
    public LocalClock()

    /**
     * access the clock's time
     * @ return this clock's estimate of the time in milliseconds since January 1, 1970
     */
    public synchronized long getTime()

    /**
     * set the clock's time
     * @param timeInMS: the time in milliseconds since January 1, 1970
     */
    public synchronized void setTime(long timeInMS)

    /**
     * @param t: the tick size used to increment the time
     */
    public synchronized void setTick(long t)

    /**
     * @return: the tick size used to increment the time
     */
    public synchronized long getTick()

    /**
     * Return a string represetation of the clock including the local time in
     * microseconds and the tick value
     */
    public String toString()
}
```

The second class you should create, `LocalClockEvaluator`, should evaluate the accuracy of the basic local clock provided. This should simply measure the error between the time reported by the local clock and the ground truth value provided by the server. Finally `ImprovedLocalClock` should implement your clock synchronization algorithm. Time information should only be accessed through the provided classes. Do not use Java's `Date` or `Calendar` classes or the `currentTimeMillis()` method.

## Example



A comparison of three algorithms for clock synchronization. The first simply sets the time to the time reported by the server, the second uses Cristian's algorithm for estimating the communication latency, and the third uses NTP's technique for measuring the time offset.