CSCI.4430/6430 Programming Languages Fall 2017
Programming Assignment #2

*This assignment is to be done either **individually** or **in pairs**. **Do not show your code to any other group** and **do not look at any other group's code**. Do not put your code in a public directory or otherwise make it public. However, you may get help from the TAs or the instructor. You are encouraged to use the LMS Discussions page to post problems so that other students can also answer/see the answers.*

Distributed Consensus, Leader Election and Rebellion!

Modified Radial Growth Algorithm: the Basics

In this assignment, you'll be implementing a simulation of a distributed leader election algorithm called Radial Growth, with some added wrinkles. The basic idea of Radial Growth is that *N* actors are situated in a ring and attempt to gain consensus on who will be the 'leader' or 'coordinator' actor by propagating messages to the left and right. If a given actor receives its own message, then it has been elected the leader! The implementation of the algorithm is given here, on slide 13.

So what's different? This is a democracy, so instead of electing a leader-for-life, we want different actors to have a chance at being the leader. Each actor will have a 'tolerance' for how long it can deal with any other actor being the leader, and each actor can be initialized with a different tolerance; once this tolerance is exceeded, this actor will demand a new leader!

If half of the actors $\lfloor (N+1)/2 \rfloor$ have exceeded their tolerance for the current leader, it's time for a new election. Once the leader *L* has received messages from $\lfloor (N+1)/2 \rfloor$ actors stating they no longer want *L* as the leader, *L* will be deposed and send out a message saying so. Once a actor is deposed, it can no longer become leader; thus the number of elections before the simulation ends will be equal to the number of actors. We also freeze time once *L* is deposed (no more timestamps are sent until after a new leader is chosen). We re-run the Radial Growth election algorithm and determine a new leader and announce a new leader at *t+1*, at which point every actor resets its tolerance counter and the leader begins sending out new timestamps.

Implementation Considerations

So, **how do we elect our new leader?** Each actor will have a priority value associated with it. Run the Radial Growth algorithm and set a given actor as passive if its priority is lower than another actor running for election, or if it has already been leader. As a test case, you can try setting the priority of each actor to its ID, which will help verify the correctness of your implementation (in this case, the order of elected leaders will follow the IDs in descending order, N-1 -> ... -> 1 -> 0).

How do we know how long it's been since the last election? It's a tad autocratic, but the leader actor will be responsible for spreading the message that another timestep has occurred (we assume no message loss). Remember that the actors are situated in a ring, so each actor can only communicate with its immediate neighbors. **To avoid confusion, timestamp messages will only be passed to the left, e.g. in a clockwise direction.** So if the leader *L* is situated to the right of actor *a*, and *a* is to the right of *b* (*b -- a -- L*), then *a* will be responsible for telling *b* that a timestep has passed. The exception is when running Radial Growth; **when running RG, you're permitted to pass messages across the circle**.

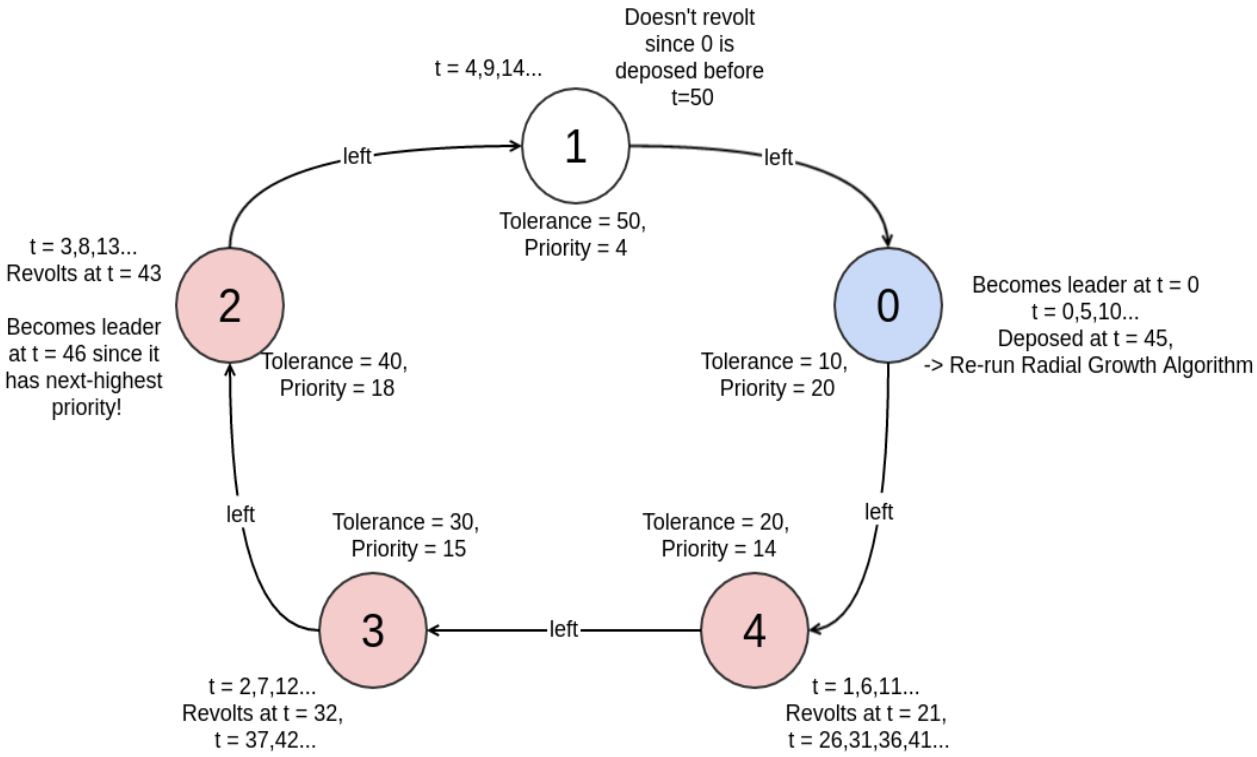
But we have a problem: **if *b* relies on *a* to find out what time *L* is stating, how does *b* know how much time it took for them to get the message?** We will assume that it takes the same amount of time for all messages to be passed between actors; so, for each actor along the chain, increment the timestamp by 1 so that each actor can interpolate how long it has been since the time first went out. Assume that it takes one timestep to send a message, and that receiving a message is instantaneous. Thus, if *L* says the time is *t=0*, and it passes a message to *a*, *a* will receive the message at *t=1*; and if *a* passes the message to *b*, *b* will receive the message at *t=2*. Thus, when a timestamp *T* reaches all the way around the ring, the leader will receive a message that states the time is $T + N$.

Keeping with the theme of autocracy, we assume that *L* wants to remain leader as long as possible. Thus, ***L* will only send one timestamp message out at any point, and wait until it receives its original message to send another timestamp.** This second timestamp should be consistent with the timestamp it received on its original message; so if *L* sends a timestamp *t=0*, which passes through four other actors, it should receive the timestamp *t=5* (do you see why?). At this point, *L* sends another timestamp, which will be received by the next actor at *t=6*, and thus forth.

This leaves one last potential problem: **how does every actor know when it's time for a new election?** After sending its 'revolt' message, a actor will continue to behave as normal until it receives notice of the new election, at which point it will decide whether or not to send out its own 'elect me' probes based on its priority and whether it has been leader in the past. **Consider the election season a land beyond time; the timestamp will freeze at *t* until a new leader is elected and assumes office at *t+1* (see the example under 'Requirements').** Additionally, ensure that whoever the next leader is, they receive the most recent time from the last leader actor, so we don't lose track of how long it's been!

Because *L* wants to remain leader as long as possible, it will not count its own tolerance against itself. Thus, while we still use $\lfloor (N+1)/2 \rfloor$ to determine when *L* is removed from office, *L* will never rebel against itself (it's good to be king!).

The following should help illustrate how the algoritm works; it demonstrates a full leadership cycle, when different actors revolt, and when the leader's successor begins its tenure.



Requirements

We expect your implementation to use a **config.tsv** file, where each line specifies the host, port, priority and tolerance for an actor. For the example above, the file should look like the following:

```
0 127.0.0.1 9000 20 10
1 127.0.0.1 9004 4 50
2 127.0.0.1 9003 18 40
3 127.0.0.1 9002 15 30
4 127.0.0.1 9001 14 20
```

Thus, we'll have five elections before we exit. The first actor has ID 0, priority 20 and tolerance 10, and will reside on host 127.0.0.1 listening on port 9000. The second actor has ID 1, priority 14 and tolerance 20, and will reside on host 127.0.0.1 listening on port 9001. And thus forth.

The Erlang version is slightly different; instead of including ports, include the name of the actor (since Erlang doesn't use port mapping but rather can identify a actor on a host machine using its name).

```
0 127.0.0.1 node1 20 10
1 127.0.0.1 node2 4 50
2 127.0.0.1 node3 18 40
3 127.0.0.1 node4 15 30
4 127.0.0.1 node5 14 20
```

The first node, `node1@127.0.0.1`, node1@127.0.0.1, will run an actor that has ID 0, priority 20 and tolerance 10. The second node, `node2@127.0.0.1`, will run an actor has ID 1, priority 14 and tolerance 20. And thus forth. **Make sure your config.tsv file is tab-separated, or the parser we provide will NOT work.**

While your implementation should theoretically work for any number of actors, we will be testing for configurations with between 2 and 8 actors.

Importantly, **it is assumed that all actors have a lefthand neighbor with the next-lowest ID and a righthand neighbor with the next-highest ID.** For example, ID 1 has a lefthand neighbor ID 0 and a righthand neighbor ID 2. For ID 0, the lefthand neighbor is the highest ID, N-1, and N-1's righthand neighbor is ID 0, thus forming the ring.

Your implementation will be expected to output the time at which a actor became leader and the time it stepped down for each election cycle, as well as the times at which different actors revolted. Using the configuration above, we should observe the following:

ID=0 became leader at t=0
ID=4 revolted at t=21
ID=3 revolted at t=32
ID=2 revolted at t=43
ID=0 was deposed at t=45
ID=2 became leader at t=46
ID=0 revolted at t=58
ID=4 revolted at t=69
ID=3 revolted at t=80
ID=2 was deposed at t=81
ID=3 became leader at t=82
ID=0 revolted at t=95
ID=4 revolted at t=105
ID=2 revolted at t=123
ID=3 was deposed at t=127
ID=4 became leader at t=128
ID=0 revolted at t=138
ID=3 revolted at t=159
ID=2 revolted at t=170
ID=4 was deposed at t=173
ID=1 became leader at t=174
ID=0 revolted at t=185
ID=4 revolted at t=196
ID=3 revolted at t=207
ID=1 was deposed at t=209
End of simulation

This should all be written to an 'output.txt' file in the same directory as your project.

Your submission must consist of two separate modules, one being a local concurrent implementation of the algorithm worth 80% of your grade, and the other a distributed solution worth 20%. The latter is realistically just a repackaging of the former, designed to use Erlang and SALSA's built-in actor management systems. That is, for the second module, instead of running all actors on a single node, each actor will be assigned to a different node. The simulation will be started from yet another node. It should be relatively easy to modify your code from the first project in order to fulfill the requirements of the second.

It is recommended that a 'supervisor' actor is used for managing the simulation. This way, you only have to pass your **config.tsv** to a single actor, which will be responsible for writing output, spawning your other actors and ending the simulation when enough election cycles have passed. Additionally, this actor can be responsible for informing all actors of a new election after the leader is deposed. There are ways of setting up actors and handling elections without a supervisor, and these are allowed, but they are significantly more complex and therefore not recommended for this assignment.

Notes for SALSA Programmers

Your concurrent program should be run in the following manner:

```
$ salsac concurrent/*
$ salsa concurrent.Run config.tsv
```

`salsac` and `salsa` are UNIX aliases or Windows batch scripts that run `java` and `javac` with the expected arguments: See .cshrc for UNIX, and salsac.bat salsa.bat for Windows. And your distributed program should be run in the following manner:

```
$ salsac distributed/*
$ salsa distributed.Run config.tsv <NameServer>
```

where NameServer is usually 127.0.0.1:3030, and the name server and theaters are expected to be running as explained below.

Time Saving Hints

1. For reference, please see the SALSA webpage, including its FAQ. Read the tutorial and a comprehensive example illustrating distributed programming in SALSA.
2. The module/behavior names in SALSA must match the directory/file hierarchical structure in the file system. e.g., the `InversionCount` behavior should be in a relative path `dna/InversionCount.salsa`, and should start with the line `module dna;`.
3. Messaging is asynchronous. `m1(...);m2(...);` does not imply `m1` occurs before `m2`.
4. Notice that in the code `m(...)@n(...);`, `n` is actored after m is executed, but not necessarily after messages sent *inside* `m` are executed. For example, if inside `m`, messages `m1` and `m2` are sent, in general, `n` could happen before `m1` and `m2`.
5. (Named) tokens **can only be used** as arguments to messages.

Running as a distributed system

To run your program as a distributed system, you must:

1. First, run the name server and the theaters:

   ```
   [host0:dir0]$ wwcns [port number 0]
   [host1:dir1]$ wwctheater [port number 1]
   [host2:dir2]$ wwctheater [port number 2]
   ...
   ```

   where `wwcns` and `wwctheater` are UNIX aliases or Windows batch scripts: See .cshrc for UNIX, and wwcns.bat wwctheater.bat for Windows.
2. Make sure that the theaters are run where the actor behavior code is available, that is, the `dna` directory should be visible in directories: `host1:dir1` and `host2:dir2`. Then, run the distributed program as mentioned above.

Please put the modified program in the file `DInversionCount.salsa`. Be sure to attempt to evenly distribute your workload across the nodes; a good way to do this is a random, uniform sampling of an array containing all nodes.

Notes for Erlang Programmers

To make your life a little easier, we've included a `parser.erl` for reading in **config.tsv**; an example usage is included in the comments of the file.

Your implementation for each solution will be run using `simulation:run('config.tsv').` **If your code doesn't observe this behavior, it will not be run and you will not receive credit for your solution.** Feel free to include any additional modules you wish to be used in your implementation, e.g. feel free to make both a `simulation.erl` and an `actor.erl` if this suits your approach.

Running as a distributed system

To make your code run on multiple machines (in a distributed network), you must use spawn/4 and related functions. Then you just need to make sure you launch Erlang with the same cookie set on all the machines. To run your program on multiple hosts, for example with a total of 5 different hosts for 4 actores and one supervisor, execute it as follows:

```
machine$ erl -noshell name workplace1@127.0.0.1 -setcookie election
machine$ erl -noshell name workplace2@127.0.0.1 -setcookie election
machine$ erl -noshell name workplace3@127.0.0.1 -setcookie election
machine$ erl -noshell name workplace4@127.0.0.1 -setcookie election
machine$ erl
erl> simulation:run('config.tsv').
```

Things to watch out for

- Make sure actors are cleaned up after the simulation is over, otherwise they may continue to run between sessions and lead to undefined behavior.
- Your output must match the examples given, exactly. Please be sure to include a newline after each `io:format` statement.

Documentation for Erlang

Please look at http://erlang.org/doc/ for Erlang examples, function documentation, and usage. Pay special attention to http://erlang.org/doc/man/io.html, and remember Google is your friend (but don't copy and paste code!).

Due date and submission guidelines

Due Date: Thursday, 10/30, 7:00PM

Grading: The assignment will be graded mostly on correctness, but code clarity / readability will also be a factor (comment, comment, comment!).

Submission Requirements: Please submit a ZIP file with your solutions in two separate directories, **concurrent** and **distributed**, plus a README file in the top-level directory. README files must be in plain text; markdown is acceptable. Your ZIP file should be named with your LMS user name(s) and chosen language as the filename, either userid1_erlang.zip (or userid1_salsa.zip) or userid1_userid2_erlang.zip (or userid1_userid2_salsa.zip). Only submit one assignment per pair via LMS. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific features / bugs in your solution.

Do not include unnecesary files. Test your archive after uploading it. Name your source files appropriately: `Run.salsa` for SALSA, and `simulation.erl` for Erlang.