GEORGE MASON UNIVERSITY

Department of Computer Science

CS 571 - Operating Systems, Fall 2013 Assignment #3
Due: Monday, December 2, 11:59 PM

FAULT-TOLERANT DISTRIBUTED COORDINATION

I. Introduction

The objective of this assignment is to combine and use various concepts we studied throughout the lectures, including **distributed coordination, communication** and **fault tolerance** in the design of a simple distributed system. The assignment has two main components: a.) implementing a distributed coordination (mutual exclusion) algorithm through sockets and RPC/RMI programming, and, b.) incorporating a basic fault tolerance capability to the system.

Note that there is a considerable design component in this project beyond implementation according to specifications. Below, you will find minimum requirements your system should satisfy at each dimension, along with some suggestions. As long as you satisfy these requirements, you can make your own design/implementation decisions. **The design and implementation of your project will be both evaluated**. Consequently, your project report should list and justify your design decisions.

II. Distributed Mutual Exclusion (DME)

We consider the reservation process of University Auditorium by various departments. When a client (an instructor) is willing to reserve the auditorium for a specific day, (s)he will contact his/her department. Each *department node*, in turn, will try to update a shared data structure/file Schedule when servicing the client request.

You will have to enforce distributed mutual exclusion, to make sure that no two department nodes are concurrently updating Schedule. To achieve DME, you will use the **Ring-Based DME Algorithm**. We discussed this algorithm during Lecture 7 (see Slides 7.15 - 7.18). The department nodes will form a **ring** and circulate a single token among themselves. A node willing to enter the critical section will need to wait until it receives the token. Once it has the token, it will enter the critical section and then pass it to its neighbor when it exits the critical section. A node which is not interested in entering the critical section will pass the token to the next node on the ring.

The algorithm satisfies the *safety* requirement, because there is only one token in the system and a node needs to have the token to enter the critical section. The *bounded-waiting* requirement is also satisfied, since the token is guaranteed to reach every node eventually – as long as each node exits the critical section in a finite amount of time. Feel free to contact the instructor if you need further details/clarifications about this algorithm. The use of other mutual exclusion algorithms (including Linear-Tree Algorithm, Central-Server Algorithm, Ricart-Agrawala Algorithm, or Raymond's Tree Algorithm) is not allowed.

In accordance with the DME Algorithm, the department nodes will form a ring and circulate a token among themselves to enforce the mutual exclusion. In any realistic distributed system, it cannot be assumed that a node has *full* information about the topology at start-up time. Each department node, at the time of its initialization/re-boot, will only have the connection information about the nodes that are one-hop away (immediate neighbors). The information about the immediate neighbor(s) can include the corresponding IP addresses and port numbers. This information can be read from a local file. This local topology file can also include information about whether the node initially possesses the token or not. A node *cannot have access* to the information about the nodes that are two-hop away and farther, at start-up time. Consequently, the topology file cannot have more than two records. However, if you prefer, you can disseminate additional information about the topology through explicit messaging once the ring is fully functional. This is a design issue which is particularly relevant for the fault tolerance aspect of the project.

Communication: You will use processes on one computer to represent clients and department nodes. You can assume that each client will connect to a well-known (pre-determined) department node, and that at most one client will connect to a given department node at a time. Feel free to use command-line parameters on the client programs to specify the network/port addresses of the department nodes they are connecting to.

To support failure detection, you can create new (child) processes or threads in accordance with your protocol. You will use RPC/RMI for client-department node communication. For inter-department-node communications, you can use either RPC/RMI or sockets, depending on your preference. You can use TCP or UDP as the transport layer protocol depending on your preference; but you should be able to justify your design decision. Do not use socket multicast/broadcast; all communication must be one-to-one.

You can use C or Java as the programming language. You can use any socket interface or remote procedure call / remote method invocation middleware as long as you abide by the rules above. However you are encouraged to use Berkeley Sockets and RPC (for C) as well as Java Sockets/RMI (for Java), since you are likely to find a large number of references for these options – and they are also available at VS&E lab machines.

Schedule will be a simple shared file, containing records regarding December 2013 reservations for University Auditorium. Assume that at most one activity can be scheduled for each day. The actual representation of reservations in the file is up to you. The clients should be able to send a Look-up request for a specific day, to inquire whether a specific day is available. Similarly, they may try to make a reservation for a specific day through Reserve request.

III. Fault Tolerance

The second project component involves the incorporation of a basic fault tolerance capability to the system. If successfully implemented and designed, your system should be able to detect and recover from crash faults that can affect the nodes, through appropriate communication and re-organization actions. In addition, when a (faulty) node becomes ready to re-join the group, it should be able to do so.

In order to tolerate and recover from faults at individual nodes you will have to design and implement an appropriate *failure detector* mechanism. The failure detection service should be also distributed, that is, the use of a centralized failure detector that would monitor all the nodes in the system is not allowed. The time-out mechanism will be one of the key

components in the design of failure detectors. Also, you can (and you are encouraged to) incorporate changes in the Ring-Based DME algorithm to support fault detection/recovery.

Once a fault is detected in a department node, the "system" should take appropriate actions to exclude the faulty node and resume regular operation (perhaps through re-organization or reconfiguration). The topology that results after a re-configuration should still be a ring. Similarly, when a faulty node recovers, it must announce its willingness to re-join the ring and it must be able to do so at a convenient time. For a recovering node, it should be possible to re-join the ring at an arbitrary point; the information about its new neighbor(s) can be provided, for example, as command-line parameters.

You can assume that no node is subject to arbitrary (byzantine) faults. Client processes cannot be used for fault detection or recovery purposes. The design of fault detection, re-organization and recovery protocol(s) is a major component of the assignment and should be justified/documented in detail in the final project report. Note that programming-language-dependent constructs (such as exceptions in Java, or signals in C) *may not be used* to detect occurrences of faults/crashes in other nodes. Like every realistic distributed system, your fault detection protocol must be based on adequate time-out mechanisms. Naturally, your specific *timer* implementation may use exceptions or signals *locally.*

The message complexity and overhead of your fault tolerance mechanism is also an important issue and it will be one of the critical evaluation factors in the grading process. For example, having each node send a "heartbeat" signal to every other node periodically would have a very high messaging overhead. *Your project report should include a thorough analysis of the run-time overhead and scalability of your fault detection, recovery and re-organization algorithms.*

Fault Model: The minimum requirement for the fault tolerance aspect is to be able to detect and recover from any *single* fault occurring in any node. Note that the faulty node may "crash" with or without the token, and you must address both cases. If a node crashes while holding the token, your fault recovery mechanism should be able to re-generate a *unique* token in the system.

You can assume that no further faults will occur until re-organization/reconfiguration is complete. However, when the system is fully functional again with remaining nodes, it must be possible to inject another fault to one node, and so on. You must accurately specify the fault model you are assuming in the project report (the conditions under which your system will be able to tolerate faults, such as the minimum time interval between two consecutive faults). You can assume that the client nodes never fail.

In case that you successfully design and implement your system to tolerate *multiple faults* occurring more or less *simultaneously*, we reserve the right to add up to 10% bonus points to your project grade (a challenging case might be the crash of the node holding the token and its neighbor).

IV. Run-time Behavior

The department nodes and clients should print appropriate messages to report important events. These can include Requesting Acknowledgment from X, Acknowledgment Sent to X, Awaiting Token, Token Sent/Received to/from X, Failure Detected at X, Entering Critical Section, Exiting Critical Section, Now Re-booting, Request Sent to Server, and so on. You do not have to write a graphical user interface, but the bottom line is that it should be possible to follow the order of events and important message traffic with little effort during the demo.

At the time of invocation, each server should accept command line parameters $d1$ and $d2$, that represent the critical section execution time and other processing delay, respectively. Specifically, after entering the critical section, the server should spend $d1$ seconds before leaving the critical section. Similarly, it should delay for $d2$ seconds before passing the token to the next node (even if it is not interested in the critical section). As a result, a server waiting for critical section entry will pass the token to its neighbor after $d1+d2$ seconds. Make sure that there is no need to re-compile the server program in order to change $d1$ and $d2$ values; changing the command line parameters should be sufficient. You can assume that $d1$ and $d2$ values will be the same for all the nodes, though $d1$ may not be always equal to $d2$. *Obviously, the time-out values used for fault detection at run-time should depend on $d1$ and $d2$ values.*

At run-time, it should be possible to follow the messages on the "consoles" of different servers and clients in different windows on the *same* screen/workspace. Having to switch to another screen (or workspace) for different servers is not acceptable. Recall that the grading of your project will be affected negatively if we are not able to follow the events clearly.

Your design should allow the emulation of "failure on-demand" to illustrate that your system can survive failures. In one extreme, you can naturally cause crash failures through "kill" command, Ctrl-C or a similar well-known mechanism. Alternatively, you can provide your own failure-injection mechanism (for example, the user may have to trigger a crash fault by pressing a special key, or a "fault-injection" process may send a related message to the involved node). Note that whenever a fault is injected, the affected node should *immediately* cease to communicate with all the peers and it should remain silent until a "re-boot" order is given by the user. It may not "announce" that it is crashing. "Re-boot" may be initiated by an explicit command or by re-running the process in case that it is killed by brute-force methods (such as Ctrl-C).

V. General Guidelines and Requirements

Groups: This project can be done either by single individuals or by a group of individuals (No group can have more than three members). You are *strongly* encouraged to work in a group in this project – you can learn more and deliver a better project through co-operation with your group members. All the members of a group will have the same project grade, so choose your partners wisely. **Each student** must send an e-mail to both *aydin@cs.gmu.edu* and *cliu6@gmu.edu* by November 19, Tuesday, giving information about his/her group members. If you fail to send an e-mail by that date, it will be assumed that you prefer to work individually. In case that you are unable to find, but willing to work with partners, send an e-mail to *aydin@cs.gmu.edu* regarding your programming language preference and times of availability (day/evening hours). We will try to match students according to their preferences, but be advised that the constraints may not be always fully compatible: it is always best to make your own group membership decisions.

Submission: Submissions will consist of soft and hard copy components.

The soft copy components will be submitted through *Blackboard*. You will need to submit all the softcopy components (listed below) as a single compressed *zip* or *tar* file. Submit your compressed file through the Blackboard by Monday, December 2, 11:59 PM. Before submitting your compressed file through the Blackboard, you should rename it to reflect the assignment number and the names of all the group members. For example, if Pam Jones and Mike Smith are submitting a tar file, their submission file should be named as: assignment3-p-jones-m-smith.tar. On Blackboard, select the Assignments link, then the Assignment 3 folder. In

addition to the assignment specification file, you will see a link to submit your compressed tar or zip file. *Please do not use Windows-based compression programs such as WinZip or WinRar* – use tools such as zip, gzip, tar or compress, available on Linux/Unix systems (also on zeus and and mason).

The soft copy components that need to be submitted in a single *tar* or *zip* file are as follows:

- a README file with:
  - the characteristics of the system on which you developed your system, that is, the compiler/operating system names and versions.
  - full operational information, including instructions on how to compile and run your programs, how to modify the configurations (e.g., IP/port numbers), how to inject faults, how to have a faulty node re-join the system, and so on.
  - a statement summarizing the known problems in your implementation (if any).
- The source codes of all your programs. Each source file should include in the first commented lines the names and G numbers of all the team members.

The hard copy component of the submission is the **project report**, which must be submitted to the instructor. You should bring the project report to your demo, in person; there is no need to submit the soft copy of the report through the Blackboard. The report should include:

1. A write-up summarizing the design and operation principles of your system
2. An accurate description of the **fault model** that you assume (The extent and number of faults that your system can tolerate: explain further if multiple faults can be injected simultaneously, the amount of time needed for recovery/re-configuration, etc.)
3. Details of protocols and algorithms you use to achieve distributed mutual exclusion, fault tolerance and re-configuration. In particular,
   (a) provide an in-depth explanation of how your system detects and recovers from:
      i. the crash of a department node within the critical section
      ii. the crash of a department node outside the critical section
      iii. the simultaneous crashes of multiple nodes, with or without token (in case that you address this case)
   (b) explain how it is re-configured after a fault (and, multiple faults have) been detected, and when the system resumes operation,
   (c) evaluate and *quantify* the scalability of your solutions (how would the run-time processing/messaging overhead and fault detection/recovery latency of your solutions change, as the number of the nodes on the system grows?)
   (d) explain how and when a recovering node can re-join the system.
4. Justification of major design decisions you made (e.g. the factors that led you to work with processes (or threads), the principle based on which you determined the time-out interval T, the motivations behind implementing failure detectors in the way you did, etc.)
5. An evaluation of your project, which may also include the comments on the complexity/efficiency of your design, the objectives you failed to reach (if any) and suggestions for further enhancements to your system
6. The pseudo-code of your programs

Demo: You will be required to present a demo of your running programs. During the demo, you will answer a few questions about your design, run your system and show its performance for at least 4-5 department nodes. You will be able to use your own laptop, or any machine accessible from the VS&E computers. The demos tentatively will take place on December 3 and December 5. Once the project groups have been finalized, you will receive an e-mail about the full list of available demo slots and you will be able to indicate your preferences in that list. **A project cannot be graded without a demo**.

Suggestions: It is crucial to manage your time wisely. If you plan to work in a group, form your group and organize a group meeting as soon as possible to determine the workload of each group member. It may be a good idea to develop your system in incremental fashion: first make sure that the system works correctly in the absence of faults and then move to the advanced requirements (fault tolerance and recovery).

Resources: You are strongly encouraged to refer to Network/Socket programming and RPC/RMI links available at the CS 571 Blackboard site (follow the Useful Links after choosing the Resources option on the left-menu). Linux/Unix *man* pages may be also useful. The CS 571 TA has a web page with sample socket and RPC/RMI programs that compile and run on GMU systems; this can page can be accessed at `http://mason.gmu.edu/~cliu6/cs571/cs571.html`. You can direct your programming and system questions to CS 571 TA (*cliu6@gmu.edu*). Make sure to check thoroughly the web tutorials and resources available through the Blackboard before contacting the TA. Conceptual questions about the project can be directed to the instructor (*aydin@cs.gmu.edu*).

Grading: The components of project grading will be as follows.

- Project Report/Design: 35 points
- Implementation: Distributed Mutual Exclusion 30 points
- Implementation: Fault Tolerance and Recovery 35 points
- Total: 100 points
- Bonus: 10 points

Late Penalty: Late penalty for assignment will be 15% per day. If you submit your project later than your scheduled demo time, you will need to make arrangements with the instructor for a new demo time. No late assignment will be accepted after December 6.