15-440 Distributed Systems
Homework 1

Due: October 9, In class.

October 15, 2012

1. In this scenario, three nodes, named `Schenley, Frick,` and `Carnegie` are working on a job for a user. The sequence of events is given:

   e1) Carnegie sends sync request to Frick
   e2) Frick receives sync request from Carnegie
   e3) Carnegie sends sync request to Schenley
   e4) Frick sends sync request to Schenley
   e5) Schenley receives sync request from Carnegie
   e6) Schenley sends sync acknowledgement to Carnegie
   e7) Carnegie receives job from user
   e8) Carnegie receives sync acknowledgement from Schenley
   e9) Frick sends sync acknowledgement to Carnegie
   e10) Carnegie sends work to Schenley
   e11) Schenley receives work from Carnegie and begins processing
   e12) Carnegie receieves sync acknowledgement from Frick
   e13) Schenley sends sync acknowledgement to Frick
   e14) Schenley sends work to Frick
   e15) Frick receives work from Schenley and begins processing
   e16) Schenley completes processing and waits for results from Frick
   e17) Frick completes processing and sends results to Schenley
   e18) Schenley receives results from Frick
   e19) Schenley combines results and sends them to Carnegie
   e20) Carnegie receives results
   e21) Carnegie sends close to Schenley
   e22) Carnegie sends close to Frick

   Using this sequence of events:

   (a) Write out the Lamport Clock representation of each timestep, using the notation L([event]) = [Lamport Timestep]. For example, L(e1) = 1.

   (b) Write out the vector time representation of each timestep.

   Solution: See next page

Event	Lamport Timestep	Vector Timestep [Carnegie Frick Schenley]
e1	1	[1 0 0]
e2	2	[1 1 0]
e3	2	[2 0 0]
e4	3	[1 2 0]
e5	3	[2 0 1]
e6	4	[2 0 2]
e7	3	[3 0 0]
e8	5	[4 0 2]
e9	4	[1 3 0]
e10	6	[5 0 2]
e11	7	[5 0 3]
e12	7	[6 3 2]
e13	8	[5 0 4]
e14	9	[5 0 5]
e15	10	[5 4 5]
e16	10	[5 0 6]
e17	11	[5 5 5]
e18	12	[5 5 7]
e19	13	[5 5 8]
e20	14	[7 5 8]
e21	15	[8 5 8]
e22	16	[9 5 8]

One way to check this is to simulate this in code, such as the following:

```
package main

import "fmt"

type Clock struct {
        id int
        lamport int
        vector []int
}

var g_event int

func max(a, b int) int {
        if a > b {
                return a
        }
        return b
}

func (c *Clock) send() *Clock {
        // Trigger event
        c.event()
        // Copy new clock data into message
        m := &Clock{0, c.lamport, make([]int, len(c.vector))}
        copy(m.vector, c.vector)
        return m
}
```

```
func (c *Clock) recv(m *Clock) {
        // Take max of lamport and vector clocks
        c.lamport = max(c.lamport, m.lamport)
        for i := 0; i < len(c.vector); i++ {
                c.vector[i] = max(c.vector[i], m.vector[i])
        }
        // Trigger event
        c.event()
}

func (c *Clock) event() {
        // Update lamport and vector clocks
        c.lamport++
        c.vector[c.id]++
        // Increment global event counter and print
        g_event++
        fmt.Printf("e%d: %d, %v \n", g_event, c.lamport, c.vector)
}

func main() {
        // Create clocks
        num_processes := 3
        clocks := make([]*Clock, num_processes)
        for i := 0; i < num_processes; i++ {
                clocks[i] = &Clock{i, 0, make([]int, num_processes)}
        }
        // Create aliases for clocks
        C := clocks[0]
        F := clocks[1]
        S := clocks[2]
        // Event list
        m1 := C.send()  //e1
        F.recv(m1)      //e2
        m3 := C.send()  //e3
        F.send()        //e4
        S.recv(m3)      //e3
        m6 := S.send()  //e6
        C.event()       //e7
        C.recv(m6)      //e8
        m9 := F.send()  //e9
        m10 := C.send() //e10
        S.recv(m10)     //e11
        C.recv(m9)      //e12
        S.send()        //e13
        m14 := S.send() //e14
        F.recv(m14)     //e15
        S.event()       //e16
        m17 := F.send() //e17
        S.recv(m17)     //e18
        m19 := S.send() //e19
        C.recv(m19)     //e20
        C.event()       //e21
        C.event()       //e22
}
```

2. In class, we examined the problem of *distributed mutual exclusion*: Guaranteeing that only a single process can be in a particular critical section at one time.

   The lecture notes discuss four algorithms: The use of a central lock server; Ricart & Agrawala's algorithm; Lamport's Algorithm; and a Token Ring.

   (a) Easy warm-up: What are the two requirements we discussed in class?

   > **Solution:** Safety and fairness.

   (b) In Lamport's algorithm, a node first puts its lock request in its own queue. It then sends a request to every other node and waits to hear replies from all of those nodes. Messages must be processed in-order. A node won't grant itself the lock until it has heard REPLIES from all other nodes containing a timestamp greater than the timestamp of its own request.

   What is the fairness provided by Lamport's algorithm compared to the fairness provided by the token-ring algorithm?

   > **Solution:** Lamport's algorithm provides fairness by using a queue to process requests from multiple nodes. Requests are granted strictly in the order they are made.
   >
   > The token ring provides a weaker form of fairness. It provides round-robin ordering on the nodes, but it does not guarantee that requests are granted in the strict order of arrival. To see this, consider the case when a node requests the mutex immediately after it has passed the token... there are many opportunities for its predecessor in the ring to request the mutex, and receive it, first.

   (c) The solutions presented in class did not consider the problem of a machine failing. Using Lamport's Distributed Mutual Exclusion algorithm, there are two obvious consequences of a machine failure: A machine dies holding a lock forever; and the inability to make forward progress because a machine fails to respond to messages.

   How would you solve the first problem, of a machine dying and holding a lock forever?

   > **Solution:** By providing a timed lease on the lock, the node will have a predetermined amount of time that it may hold the lock before renewing the lease. At the end of the lease, if the node has not requested a lease renewal then the lock is revoked and other nodes may now hold the lock.

   (d) Token Ring is also vulnerable to a node failure: Node N will try to send the token to node N+1. If its connection to N+1 fails, N+1 fails, the token will get "stuck" at node N. Improve this algorithm so that it is robust to connection failures and to the failure of a node that is *not* currently holding the token.

   Give a description of your algorithmic changes and why they work. Code is not needed. You may assume that the network is modestly-sized enough that you don't need an O(1) algorithm - using up to or less than O(N) memory and CPU is fine.

   > **Solution:** Each node must know the successor of every other node. If a node tries to pass the token to its successor and does not recieve an ACK after a certain amount of time, it will attempt to pass the token to it's successor's successor. It will continue doing so until it recieves an ACK from the node.
   >
   > This solution ensures that the system is robust as long as there is a single node left holding the token.
   >
   > The major drawback to this approach is that if node N is *wrong* about node $N+1$ not being alive, it will unfairly deprive node $N+1$ of the chance to grab the lock.

(e) If you were particularly concerned about the death of one machine preventing progress, which of the discussed solutions to distributed mutual exclusion would you use? Explain briefly why.

Solution: There are several correct answers to this question. You may consider the number of messages required to acquire a lock, how the fairness of the algorithm behaves, the complexity of implementing the system, and the algorithm's robustness to failure.
The solutions we had in mind were:

- A central coordinator is less likely to stop working if a single random node dies. Instead of being cripped by the death of any of N nodes, now only $\frac{1}{N}$ nodes can kill things.

  The central coordinator can be made more robust by running multiple coordinators and using a distributed election (or replication) scheme to determine which one is the head. This is the scheme used most often in reality.

- A token ring with the aforementioned changes, both leasing and the "skip-a-dead-neighbor" trick, is a good start to building a more robust system. The drawbacks of the token ring approach – many messages, and potentially long delays until you hold the lock – remain, however.

3. Dropbox is, at its heart, a distributed filesystem. See this page in the dropbox docs: `https://www.dropbox.com/help/36/en`.

(a) Allowing such conflicts to happen is a deliberate design decision in Dropbox. Briefly explain how this differs from AFS, and how AFS handles concurrent modifications to a file. (Briefly: 3-4 sentences)

Solution: Dropbox handles write conflicts by creating new files for each client with a different name. While AFS client writes through at file close and the server immediately informs other clients that have cached the same file. Thus the application needs to cooperate and perform necessary synchronization.

(b) Why do you think Dropbox chose its design over AFS's design? What advantage does it confer to Dropbox?

Solution: Dropbox was primarily designed to help each user share and synchronize files among different computers and devices, so it is less likely to have two clients modifying the same file concurrently. Advantage of Dropbox's design includes: it is simple to design and implement; and it ensures no version is overwritten.

(c) In what way is caching in dropbox more similar to AFS than it is to NFS?

Solution: Dropbox and AFS both cache the whole files while NFS only caches blocks of the files.

4. Some questions about RPC.

(a) RPC is designed to reduce programmer effort in writing distributed systems by making remote function calls as easy to use as local function calls. Identify *three* different ways in which this abstraction does not hold true.

Solution: There are four basic drawbacks for RPC over local calls: Connectivity loss (a local function call typically can't fail in the same ways); Latency; Increased Total Overhead; No hardware or local caching advantages.

(b) Hit the web to answer this one: Briefly identify two advantages and one disadvantage of using Google's "protobuf" format instead of the JSON format you're using for marshaling in Project 1.

Solution: Two general advantages: JSON is more widely used; JSON is more easily human readable, and so doesn't require much, or any code or implementation context to userstand/verify/debug a JSON packet. JSON is very, very easy to use from javascript, making it an easy choice for communicating between browsers and servers.

Protobuf is much more compact than JSON, particularly for numbers and arrays. Protobufs are faster than JSON serialization. Protobufs include explicit versioning information to help adapt to future needs.

(c) Your code in Project 1 handles the problem of preventing duplicate RPCs by using what networking people call "Stop and Wait": It has one message outstanding at a time. If a message with a sequence number lower than the next-expected sequence number is received, the system will throw it away.

Imagine using your system to talk between CMU and the machine at UW from micro-quiz 1. Assume the RTT from here to UW is 70ms. How many RPC calls can you execute per second using a single client connection from here to UW?

Solution: RPC is independent of the underlying transport layer. "Stop and Wait" means that a new Data Packet cannot be sent until the ACK for the previous packet is received. Thus, sending the data packet out and waiting for its ACK require at least one RTT (round-trip time). With this at 70ms to UW, the rate of data sent is capped at 1 request per 70ms, or only 14 RPCs per second.

(d) Briefly sketch out how you would improve this design to handle having multiple RPCs in flight at a time.

Solution: Technically an open question. The common case is to add caching to Data both seen and sent, and release the incomming data to the user once the appropriate data packet has sucessfully been ack'd. Allowing EpochHandler() to still resend appropriate packets to recover unacknoledged and missing messages. So Client and Server will retain all advantages of LSP, but with the ability to send off new packets wihtout waiting on an Ack. This is a feature of TCP.