15-498 Project #2
Simulating a Distributed System

Times of Interest

- Friday, February 15, 20008: Assignment distributed
- 11:59PM on Friday, March 7th, 2008: Assignment submission deadline

Educational Objectives and Overview

This lab is designed to expose you to an important tool for distributed systems research and development, thread-based simulations of distributed systems, as well as to reinforce your understanding of mutual exclusion in a distributed environment .

You are asked to build a framework for simulating a distributed system. Once this framework is constructed, you should use it to evaluate the performance of synchronization primitives based on each of the "majority vote with tie breaking" and "path compression" techniques. Specifically, you should implement the primitives, themselves, as well as appropriate workloads to evaluate their performance under various contention and failure models.

Additionally, you are asked to write a report describing the architecture of your simulator, your workloads, the observed behavior of each algorithm under each workload, and a comparison of the behavior of the two algorithms.

We hope that, in the end, you'll leave with a much better understanding of the behavior of each technique, as well as the skills to implement a full-visibility simulation of a distributed system.

Your Mission

Simulate multiple processes competing for one or more serially reusable memory objects, using each of majority voting *with tie breaking* and path compression to control access. Basically, assume that there is one instance of a memory object in your simulated distributed system and that a collection of threads are vying for access to it. For this project, don't worry about migrating or replicating the object(s) – we'll discuss techniques for doing this later. Furthermore, don't worry about doing anything with the simulated shared memory object – just acquire it, hold it, and release it.

Basically, just ensure that the processes can get mutually exclusive access to each of the simulated memory objects using each of the algorithms. Evaluate each algorithm under various levels of contention and failure, and write a report. This report should describe in a meaningful way your results for each technique, as well as provide a meaningful comparison of the two.

You should consider such things as the number of messages required per access to the critical section, as well as the effects of various levels of contention and failure.

You are required to simulate a system with a single memory object. But, it would be fun and interesting to simulate a system with more than one. And, given a properly designed simulator, this won't likely be a much bigger undertaking.

The Simulator's Architecture

We are not prescribing any particular approach to the design and implementation of your solution. But, we do want to offer you some ideas that might, or might not, be of use to you.

You are being asked to simulate many different things – two different synchronization techniques and many different workloads. If you take the approach of writing a new program, completely from scratch, for each possibility, you are likely to write well more code than you need. As a result, you probably will not have time to test as many workloads as you would otherwise like, and you run the risk of annoying your grader with piles and piles of code. And, let's not forget that you need leave yourself in a crunch for the report.

To avoid the "code diarrhea – no time left" syndrome, we'd like to suggest that you consider your solution to be composed of at least three different components, and design it with these, and reusability, in mind. These aren't listed in any particular order, and should probably be designed concurrently to ensure interoperability.

1) The concurrency control primitives: request() and release() primitives for each of the two algorithms.

2) A workload generator: Determines when the simulated tasks should request the critical section and for how long each should hold it. This probably should also be able to generate hazards such as partitionings and lost messages.

3) The simulated tasks: The logic that simulates a process requesting access to the critical section and releasing it, as directed by the workload. You might also want a simulated task to fail, as requested by the workload.

4) The simulation framework, itself: The framework for moving and queuing of messages from one simulated task to another, and simulating communication failures as directed by the workload. This might also implement the primitives used to instrument the simulation and collect counts, such as the number of messages of each type, or in aggregate.

Simulating Tasks

Your project should simulate tasks in a distributed system. There are two ways of doing this. One technique involves simulating each task with an independent thread. Another technique involves making the entire simulation event driven.

If you decide to simulate each task with an independent thread, there are a few considerations. First, your simulator will almost certainly be non-deterministic. This can make it less useful than a simulator that produces exactly the same results each time. In order to mitigate this, you'll want to implement a detail logs – this will, at the least, ensure that you are able to determine the details of any particular run. Second, in order to avoid unpleasant interaction between communication and the thread logic, you might want to use two threads: one to handle communication, such as voting, which needs to happen even if the thread's logic is blocked waiting for the critical section – and another thread to simulate the thread's actual logic.

If you decide to implement an event-based simulation, the simulation becomes an exercise in queuing. The idea is to use a global queue to keep track of time. As events are requesteds, they are enqueued. Every event goes into the global queue – requests, releases, messages, &c. The simulator's time is always the advanced to the time of the next event on the queue in order that it can be executed without busy-waiting until that time. You might still choose to use threads to simulate program logic. But, if you do, you need to keep them synchronized so that they only act as directed by the global queue.

Simulating A Network Using Shared Memory

This project asks you to simulate network communication using the process's memory, which is shared among the threads. Many approaches can satisfy this requirement. Below we describe one approach, including simplified pseudo-code. You are not required to base your solution on this pseudo-code. We just provide it as a starting point for those who are looking for some direction.

The approach below models a network by giving each simulated host a queue for incoming messages. When one host sends a message to another host, it accomplishes this by adding its message to the receiver's queue. Since each queue is shared between the receiver and the potential senders, access to it forms a critical section. The approach below uses condition variables for concurrency control.

For clarity of the pseudo-code, a simple FIFO queue is used. But, you might want to implement a priority queue, in order that messages can be sorted and processed by time. This will become useful, for example, when a message should be sent at time X, but received much later, at time X+Y.

To organize the collection of queues, each queue is built from a simple queue and the synchronization primitives required for its access. An array of these complex queues is created with one entry for each simulated host. The simulated host's receive queue can be found by indexing into this array using a serially assigned thread id.

Global (Per process) Structures and Storage

```
typedef struct
```

```
{
  mutex mx;
  condvar cv;

  message head;
  message tail;
} msgqueue;

msgqueue msgs[NUMBER_OF_THREADS];
```

Pseudo-code to Receive Messages

```
mutex_lock(msg[tid].mx);

if (!msg[tid].head)
{
  cond_timedwait (msgs[tid].cv, msgs[x].mx, timeout);
  ---------------- OR -----------------
  cond_wait (msgs[tid].cv, msgs[x].mx);
}
incoming_msg = msgs[tid].head;
msgs[tid].head = msgs[tid].tail = NULL;

mutex_release(msgs[tid].mx);

while (incoming_msg)
{
   // take action for incoming_msg (first message in queue)
   // dequeue incoming_msg (first message in queue)
}
```

Pseudo-code to Send Messages

```
// Given a new message, new_msg

mutex_lock(msg[tid].mx);

new_msg.next = msgs[tid].tail;
msgs[tid].tail = new_msg;
if (!msgs[tid].head)
  msgs[tid].head = msgs[tid].tail;

cond_signal (msgs[tid].cv);

mutex_release(msgs[tid].mx);
```

The Workloads

Although your workloads can, technically, be hand-crafted – it is a much better idea to generate them automatically using some sort of statistical distribution as a guide, for example, an exponential or normal distribution. Using these as the "basics", you can then add failures or other features. Doing this will enable you to test many, many possibilities with far less work.

There are four basic things to consider in developing your workloads:

1) When should each processor request access to the critical section?
2) How long should each access to the critical section last?
3) When should failures occur?
4) How long should each failure last?

You are welcome to answer these statically by creating long lists of activities by hand. But, for reasons of your own sanity, we don't recommend this. Instead, we recommend that you use statistical models to describe these things. For example, you might want to assume that the frequency of each processor's request is governed by a uniform distribution – basically sprinkling an equal number of processors along the distribution from "never use critical section" to "sometimes use critical section" to "constantly use critical section", and everything thing along the way. Or, instead, you might want these to follow a linear distribution or an exponential distribution. The same goes for failure. You probably want to pick a model for the occurrence of and duration of failures, and then generate failure events accordingly.

You might even want to try different models. You might want to assume, for example, that all accesses to the critical section take exactly the same amount of time, and compare it to some other model.

In general, uniform random distributions are the simplest to construct. It takes a little more work to construct normal distributions and exponential distributions – but formulas for each are well documented on the Web and in other places. Drop by if you need help finding such documentation.

Although the most straight-forward to implement, uniform distributions rarely occur in the real world – nothing is perfect. Instead theoretically uniform things often follow a normal distribution – just like we often discuss of the grades in a class. The distribution is weighted at the center, with increasingly more distant occurrences becoming less likely. Exponential workloads tend to be a good model for many real-world systems that are tied to real users, for example, the duration of processes, accesses to Web sites, the sizes of objects on the Web, &c. Exponential distributions involver very few short or small things, but a few very large things (or exactly the opposite).

Although it is interesting to compare results with distributions, I won't promise you that it will necessarily be compellingly interesting. You shouldn't test things just for the sake of testing them – instead test them because you think they are meaningful. In the case of this project, in particular, the results will likely be interesting, but not compelling.

Tweaking Workloads

In addition to workloads based on pure distributions, you might want to tweak some workloads to test certain edge cases, for example the "tie situation" in the majority voting scheme. This is especially important if you suspect that certain cases might have interesting results – and they don't happen to come up in your distribution-generated experiments.

You might also want to try to determine how often these interesting cases occur within for different statistical distributions at different levels of contention.

The Report

The goal of the report is to tell us two things:

1) what you have done
2) what you have learned

By reading your report, we want to gain an understanding of how you constructed your simulator(s). This should include a discussion of how you simulated the tasks and the network. It also should include a discussion about how your simulators can be configured for different workloads. Figures, &c should be used, as helpful.

We would also like to understand your workloads. Why did you select the particular distribution of events. How did you decide when each process would request entry into the critical section and how like it would stay – this might boil down to the description of a random number generator with a particular distribution. Similarly, we would like to understand why you chose to have hosts fail as they did.

You should present some meaningful summary of your observations. This might include tables, charts, or graphs. If you'd like, you can include the unprocessed raw data as an appendix – but, for our purposes, we'd like something easily digested to speed along grading.

We'd also like a brief analysis of your results. The goal of this part of the report is simply to convince us that you understand the meaning of the raw data discussed above.

If you know of any weakness in your simulators, workloads, &c, we'd like you to briefly discuss those. The same is true if there are things that you think might have been useful, but weren't practical within the timeframe of the project.

Grading

To do everything we ask is a great deal of work. We know that. But, this is only required for an "A" – which is a grade that means "distinction". A "B" is a huge step less work, and so is a "C". We feel that even a party animal with a girl/boyfriend in a different state should be able to pull off a "C".

To get an "A", you must do it all and do it well. In other words, we want both techniques implemented and tested in the meaningful cases, a well-designed and implemented simulation framework, and an excellent report.

To get a "B", you must implement both techniques, and test them in the common case and with some failure. Your simulation should be well-designed and well-implemented. Your report should also be excellent.

To get a "C", you must implement either technique and test it in the common case and with at least some failure. Your simulation should be readable and understandable. Your report should be excellent.

To get a "D", you must implement either technique, test it in the common case, and write an excellent report. Your simulation can show the signs of a hard-fought battle.

To get an grade in the 50s, you must submit a project that shows the signs of a hard-fought battle and can simulate the common case, but perhaps with significant problems (crashes, freezes, results slightly off), &c. Your report should be excellent and describe what you learned – and what went wrong.

Grades below 50 are given to reward effort without results that rise to the level of achievement.

Note: Projects with less than excellent reports will be graded down by at least ten points. Think of it this way: You are building the simulation to report the architecture and simulations results. Do the latter well.

One Plan of Attack

1) Think about your workload generator. What form will its output take? Will there be one generator for all events? Or one per process? Or one per process and one for the network? Or something else? Will things be trace driven? Or will there be direct calls to generator functions?

2) Think about the tests that you want to run – revisit step 1 and double check.

3) Think about your thread main body. How will it interact with the workload generator or workload script? What primitives will it use? If it blocks waiting for events, can it still handle unrelated communication, such as adjustments to its edge(s) in the global graph? What primitives will it use?

4) Think about the design of the network infrastructure. How will the threads dispatch messages? How will disparate dispatch and arrival times be specified. What API will it publish? If there is one workload generator, it should be an API for the simulation as a whole. If the generator is per thread, it will be used within the thread library. If the network failures are generated globally, there needs to be a way of achieving this, the same if they are done on a per-thread basis. Does the network involve its own thread? Or is everything "pushed and pulled" by the simulated processes?

5) Think about the simulation from a more global perspective. Is coordinated time needed? Is there a work loop that is generating (counting) this? How is it communicated to the threads.

6) Think about the synchronization techniques. Can you implement them with a simple API given your design? If so, sketch out the code. If not, revise your design and try again.

7) Design the whole simulation, sketch out each component.

8) Make sure you specify the interface between each component super-clearly.

9) Design the implementation of each component

10) Review everything. Is it still feasible? A good design?

11) Double check for mutual exclusion problems within your simulator. Are all of your critical sections covered? Check everywhere that two threads interact.

12) Code and test

13) Repeat 12 until complete.

14) Now, test with a mission. Try the workloads that you eventually want to report. Make sure everything makes sense. If it does, record it. If not, explain it or declare it broke'. If broke', throw away bogus data, rerun the experiments and reverify results.

15) Draft your thoughts. Find the wholes

16) Collect more data

17) Repeat 16-17, as necessary

18) Write the report. Create graphs or charts as necessary. Only include raw data in the appendix, and then only if it is useful to do so.

19) Polish everything.

20) Take a deep breath. Relax. Loop back. Be proud.