

Management of Distributed Replicas

Report on One Semester of Sabbatical Leave during Fall 2008

Stephen J. Hartley
Computer Science Department
Rowan University

Executive Summary

Although I didn't "rewrite completely" the Java program that Prof. Crichlow and I have been using for over five years in our research, as specified in my application, I accomplished all of my goals by making numerous, substantial modifications to the program. The program simulates and evaluates the management of database replicas in distributed computing systems and uses an algorithm that combines optimistic and pessimistic transaction processing. The simulator originally handled its queues of pending work inefficiently and was reworked during my sabbatical leave to use the recent Java library of data structures for concurrent programming, `java.util.concurrent`.

Technical Details

All of our simulations to date have worked with a single resource type. The software was extended to handle multiple resource types, with a single resource type being a special case. A paper about this extension was submitted to a conference and accepted: "Using COPAR to Facilitate Quick Distribution of Disaster Relief," with Joel Crichlow (presenting author) and Michael Hosein, *Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications*, Orlando, Florida, November 16–18, 2008.

The simulator handled its queues of pending work in an inefficient manner: it examined (polled) the queues at fixed intervals. I fixed the inefficiency by using Java's blocking queue data structure so that the queues are processed when they change and only when they change. I also fixed some "check then act" race conditions involving the queues; these had been causing the program to "crash" (abort with an error message) at unpredictable times.

While making these modifications, I discovered a major flaw and a minor flaw in the algorithm and its implementation in our simulator, both easily fixed.

The major flaw resulted in the same resource possibly being allocated multiple times instead of just once by the optimistic part of the algorithm. The thread executing the pessimistic (permanent allocation) part of the algorithm recalculates, as part of every database transaction commit, the number of resources the thread executing the optimistic (tentative or

temporary allocation) part of the algorithm is allowed to allocate. If the permanent allocation thread gets way behind the temporary allocation thread due to network delays or congestion, the permanent thread's calculation is based on stale information relative to the temporary thread and might allow the temporary thread to allocate resources it should not. This potentially leads to violations that must be later undone. This flaw was fixed by turning off the permanent thread's recalculation and having just the temporary thread keep track of how many resources it has available for allocation. As part of this fix, after receiving the first message from a server performing optimistic processing on a transaction, the parent of that transaction sends "compensate" messages to all other servers that later perform optimistic processing on the transaction, telling them to undo or reverse their optimistic processing of that transaction.

The minor flaw can also lead to extra violations and undone transactions but to a lesser degree. If the temporary thread puts transactions that it cannot immediately satisfy into a queue so that it can check them again later, the temporary thread might end up performing transactions in a different order than the permanent thread, possibly leading to extra violations that must be undone later. This happens because when the permanent thread encounters a transaction that it cannot satisfy, it discards the transaction and does not try it again later. This flaw was fixed by eliminating that queue and having the temporary thread discard transactions that it cannot immediately satisfy.

During these modifications, I noticed that the simulator was frequently copying objects, many times unnecessarily. The idea was for a thread to make a copy of an object before modifying the object and passing the modified object to another thread, keeping the threads from interfering with each other's processing.

In any concurrent (multithreaded) program, such as our simulator, it is highly desirable to have as many objects as possible be immutable (unmodifiable), another technique besides copying to keep threads from interfering with each other but much easier to verify. However, no objects were immutable in our simulator. As part of eliminating unnecessary object copying, I made all objects immutable except the queue objects and the singleton object containing the state (current resource counts) of the simulation, which are objects that the threads must share. Whenever an immutable object needs to be modified, a new copy is created with the new value instead of changing the original object. Both the new object and the original are immutable. (This modification was not quite completed by the end of the sabbatical period.)

Other Activities

I read the excellent book *Java Concurrency in Practice* by Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley, 2006).

I obtained two UPS (uninterruptible power supply) units for our lab machines. I downloaded and installed the software needed by the machines to know when power is lost so they can shutdown gracefully. Unfortunately, losing power is not a rare event in Robinson Hall.