15-498 Project #3: Distributed Database

Times of Interest

- Wednesday, March 5, 2008: Assignment distributed
- 11:59PM on Wednesday, April 9, 2008: Assignment submission deadline

Overview

This project is very easy to describe: You should design and implement a real, live distributed database by applying the tools and techniques that we have discussed in class, as well as the techniques for recovery that we are currently discussing. You should also write a report describing your solution.

Educational Objectives

The education objectives are also straightforward:

- Gain expereince with Java RMI, ONC RPC, CORBA, XML-RPC, or SOAP/RPC
- Reinforce understanding and gain practical experience with replica management
- Reinforce understanding and gain practical experience with failure management and recovery
- Additional experience and understanding of the design and implementation of distributed software
- Improved technical authoring skills
- A sense of accomplishment that can only be achieved by looking back at a job well done.

Software Environment

Your solution should be buildable and runnable in the standard r Linux/Andrew environment.

Database Functional and Structural Requirements

The requirements are very straightforward:

- Your database's API and semantics should follow one of two models:
    - Option #1:
        - It should be modeled on gdbm/ndbm. The semantics of your implementation should match those of gdbm/ndbm as closely as is reasonable given the

        distributed nature of your database. This approach lends itself to a distributed hash
    - Option #2:
        - It should be modeled after SQL – but it need only implement a small subset. For example, it should include rudimentary implementations of CREATE TABLE, ALTER TABLE, DROP TABLE, SELECT, INSERT, DELETE, UPDATE, FETCH, JOIN. and possibly COMMIT, ROLLBACK, and SET TRANSACTION. It should also include other functionality as interests you, or as is prudent given your design. This approach is more real-world.

          Please note, you should only take this route if you are already familiar with SQL, or if you want to learn. We won’t teach it in DS, but there are tons of resources on the Web. Please also note that SQL is very large and offers many opportunities. You’ll need to be very picky in what you chose to do – you can’t do it all

          The database should be capable of growing to a size many times larger than can be accommodated on single host. (What's many? Any reasonable interpretation will suffice).

- The database should be more available in light of failure than a similar database residing on a single host. Additionally, the database should be able to tolerate the failure of at least one replica without reduced accessibility.
- A mechanism should be provided for recovery after a failed replica is physically repaired.
- The performance of the system should be comparable to that of a similar database operating on a single host as measured by the latency of the operations and the maximum throughput of the system. Comparable does not necessarily suggest identical or better. But it does imply that you should minimize the impact of network latency, &c.

Interface (API)

As noted above, you can pick between an gdbm/ndbm-style interface and an SQL-style interface. In either case, you will need to adapt the interface to a distributed environment.

If you are taking the gdbm/ndbm route, you should be as loyal to the original as is possible – and should only exclude features that are incompatible with a distributed database. If you are taking the SQL route, you will need to be very, very picky – you cannot possible do it all. Keep in mind the goals of the project and do enough to satisfy those. Beyond that – do enough to be truly proud.

The Report

The report is absolutely critical. It should explain the overall design of your database, what techniques it uses, and why. It should explain the major design decisions you made along the way, as well as why you made them. It should also describe your testing, and any shortcomings.

Since this project is open-ended, the designs, features, &c will vary significantly across the entire class. We will read your code, but a good report will help us in many ways. It will help us to understand the organization of your solution and your goals, so your code will be more readily understood. It will ensure that we don't miss anything important. And, it will highlight your thought process - something that isn't always discernable from code alone.

The Source Code

We will read your source code. Please organize it in a way that is manageable and follow good coding practices. Please remember that code should be readily understood by both machines and people!

Submission Procedure

Your project should be submitted using the same procedure as the prior two projects.

Suggested Plan Of Attack

One approach to this project might be to "grow it" as described below:

1. Read the ndbm or gdbm “man pages” or read SQL documentation on the Web.
2. Construct and manipulate a "toy" database. The goal of this is to ensure that you understand how the original, non-distributed versions work before you try to emulate them.
3. Design your solution keeping in mind that you'll probably want to grow it as described below.
4. Implement a very simple base for your solution. This solution might include only one repository, one client and one server - all living on the same host. You might want the client and server to be

logically separate, but connected via local function or object/method calls.

5. Separate your client and server so that they run on different hosts replacing function calls with remote invocations.
6. Expand your systems capability so that it implements a write-all/read-one quorum system and multiple servers.
7. Implement a more flexible quorum-based (or not) system, if desired
8. Implement client-side caching, if desired.
9. Implement failure recovery, perhaps using logging and/or checkpointing.
10. Write your report

Another approach might be to design it on paper and then implement it one piece at a time, without growing it from a non-distributed system. This is actually the way I would choose to approach the problem. The advantages are that the implementation is less likely to have vestiges of the centralized approach, the design is less restricted, and less time is wasted writing code that won't be used. The disadvantage is that the system can't be tested as a whole until the very end, so progress might be more tentative and harder to measure. Careful design and modular testing can, of course, mitigate the uncertainty – but it can’t eliminate it.