15-395 Lab 7
Sockets – Network Database

Objective

This lab is designed to give you exposure to network programming using Berkley sockets. It is also designed to reinforce your understanding of shared libraries. But, most importantly, I think it is a good exercise in software design. It asks you to reuse code, design an interface, design a protocol, and produce good software.

Times of Interest

Assigned: Tuesday, November 12th, 2007
Due: Tuesday, November 19th, 2007

Part 1: The Goal

Last assignment you built a database library and a simple client program to demonstrate that it works. For this assignment, you are going to do three things:

- Write a database server which will use this library to maintain the database. Basically, this server should expose the functionality of your library via the network, so applications can use it to maintain a database via the network.

- Write a library for creating network database clients. This should be a shared library. It should project your original database library onto the client in a transparent way. In other words, this library should function as if it were your original database library, with one important exception: It shouldn't maintain the database, it should just communicate with the server that does.

- Write a network database application that uses your newly create library to maintain a remote database

- Ensure that your database server can maintain at least a few concurrent connections, including connections form different hosts, and test this functionality (I will).

Part 3: Considerations

You'll probably find the slides from the sockets lecture helpful. Ditto for the slides about Threads and concurrency control (although you might choose to use select() instead of threading). These slides are available on the Web.

Among books, the Network Programming, Volume 1: Network APIs: Sockets and XTI is my favorite – and it has been recently revised (by a new co-author). There are also copious tutorials on the Web.

If you'd like a quick look at networking beyond the API, you might want to look at some of my old OS lecture notes, back when we covered some networking in OS: http://gigo.sp.cs.cmu.edu/~cs412/fall00/ln/. Lectures 28-30 (mostly 29 and 30) might be of interest.

There is plenty of good reading on the Web, too. Ask your favorite search engine.

Part 2: API Considerations

You'll probably want your network API to mirror the local API as much as is possible – don't change the interface, except as necessary. Remember, the two libraries are ding the same things. If we were coding in Java, we'd use the same Interface. In C++, the same abstract class.

One thing that you'll probably need to change is the way you name the database file. In addition to a local file name, you'll need to include the hostname or IP address, and port number. You might consider replacing the filename argument with a richer string – something that is patterned after a URL, for example: "128.2.210.27:5500/myfile.db"

If the library interfaces are exactly the same, you might consider placing them into two different libraries and letting the user select between a local and a remote database by linking the right one. This is a cool idea – but not one without peril.

It is conceivable that the same application would want to use both local and network databases – and this could be a nightmare. If you'd like to do it to explore linking – great idea. But, please do understand the significant limitation.

Part 3: Communication Consideration

You'll need to decide if you are going to use SOCK_DGRAM/UDP or SOCK_STREAM/TCP. Either way, I'd like your communication to be *reliable*.

In favor of a SOCK_STREAM, it is reliable by default and it can readily accommodate large messages. Also in its favor, it uses familiar read() and write() for communications. But, it does carry with it overhead – it takes time to establish and tear-down sessions. And, it might take some effort to parse the communication, since it is a stream.

In favor of SOCK_DGRAM, it is fast and cheap, and it is message-oriented, which might make the parsing a moot issue. But, its got some disadvantages. Since it is unreliable, you'll need to add your own ACK-and-resend protocol above it. And, because UDP packets aren't to be fragmented, they can't be larger than 576 bytes (well, at least without some knowledge of what's downstream). So, you might well need to string several messages together to accomplish your goals – we wouldn't want to limit our users to small objects.

I also recommend taking some time, in advance, to think about how your client and server will communication – specifically, what messages they wills send/receive, how they will be formatted, and what rules will be needed to govern them. In other words, I'm suggesting that you

think of this communication as an "application level protocol", not something ad hoc, and take the time to design it. I really think this'll help you in the long run.

And, it shouldn't take too long – you've already got the API. You know what needs to be sent and what should be expected in return. You know what the messages look like. Just decide how to "marshal them" into organized messages.

Part 4: Concurrency

Once you have the database server working for one client at a time, please make it concurrent. In other words, use either select() or threads so that it can handle multiple requests at the same time. This makes good sense – since the same server could be managing many databases located on many disks that can operate in parallel.

But, be careful here. Think carefully about your data structures. Will anything be damaged if two users try to manipulate it at the same time? One traversing a linked list while another has pointers out of place for a remove?

Should these problems present themselves, you don't need to solve them with tremendous efficiency – but you do need to solve them. Consider using a mutex (or similar) to lock down this type of data structure. The code that manipulates a shared resource that is fragile in this way is known as a *critical section*, and solving the problem might be as simple as guarding the critical section with a mutex. Wrapping your whole server in a mutex prevents any concurrency – and, consequently, is a bad idea.

Part 5: Common Sense

Your project must include a *Makefile*. It should be well-written, including comments, descriptive identifiers, &c

But, you might want to construct some tools or add some instrumentation. For example, you might want to add logging to your libraries. Or, you might want to write a tool to walk the database and list records. Or, you might want to write a simple proxy that you can stick on the network in between the database and the client that does the logging. In this case, the proxy would just have to sockets and act as a go-between for the client and server.

Last, but not least, you'll want a ready source of data with which to feed your database. Try finding some on the Web and mining and formatting it using shell scripts. You want to test with thousands of records – or, well, even way more.

Part 6: The Most Important Part

I'm here to help. Please let me know how I can be of service.