CSE P 590 SG Assignment #2a

Due February 3, 2004

This programming assignment is due in class (before class begins) on Tuesday, February 3, 2004. Be aware that we may hand out another short written assignment to overlap, and plan accordingly. If you haven't done any network programming before, you'll want to start *immediately*. This is an individual assignment.

What to turn in (and how)

We are going to try something different in terms of submission this time around. We have set up the CSETP Online Test Facility for you to show us your solution running on your computer. To use this facility, you set up your server to connect to our authoritative server as a client, and then you use the server-initiation interface to make our special client synchronize with your server. Our special client will then verify that your server is keeping proper time, and send you an email to that effect.

What to email the TA Turn in the assignment via email to the TA (`acollins@cs.washington.edu`). Include a short textual description of how your code is laid out and how it works, and the answers to questions 1 and 2. Then attach your code file(s), and a copy of the email response from the test facility (to make sure we can connect it back to our copy of the test results).

Your grade will be based on your answers to the questions, your code (which we plan to read but not compile ourselves), and your successful completion of our online test.

CSETP—The CSE Time Protocol using TCP

Background

The assigned paper by Cristian [Cri89] discusses the limits of a simple clock synchronization primitive and then goes on to describe a basic architecture to distribute "authoritative" clocks across a wide-area network. This architecture is surprisingly similar to the now widely-used NTP [NTP03] protocol.

NTP adds one important wrinkle to the basic Cristian algorithm, which is the notion of primary and secondary time servers. Primary servers have direct access to a time source (like a GPS or WWV receiver), while secondary servers synchronize to primary servers to get their "authoritative" time, which they then serve out to other clients.

We in the department of Computer Science and Engineering, however, have never met a well-developed, carefully-implemented, and widely-popular distributed system that we wouldn't rather take apart and build instead our own hacked-up, get-it-done-quick, basic-functionality imitation. In that spirit, we propose CSETP, the CSE Time Protocol.

Protocol definition

CSETP extends NTP's primary/secondary notion to a general idea of tiers of servers. Every node in the system is both a client and a server. Each node connects as a client to one other node (the "parent node") in order to obtain the authoritative time, and then serves its own synchronized clock out to any number of other nodes (the "children").

Each node knows its own tier number, a non-negative integer which is always one greater than the tier number of its parent. There is exactly one tier-0 node in the system, running on tioga.cs.washington.edu.

Any node that connects directly to tioga is then a tier-1 node, and so on. The "correct" time in CSETP, then, is whatever tioga says it is; there is no notion of multiple roots tied to some base reality as in NTP. The server on tioga in fact follows a "grad-student" time that tracks real time only to within a ρ of 0.01 (grad students not being known for keeping good time).

The thing that makes the CSETP protocol unique is the way that it allows us to work around firewalls in establishing arbitrary client-server distributed systems. CSETP supports two styles of client-server interaction, "client-initiation," which looks remarkably like traditional TCP client-server interaction, and "server-initiation," where the node that is logically the client is the one that binds to a well-known port number and the node that is logically the server makes the first step in connecting.

Messages in CSETP are newline-separated ASCII text sent over a TCP connection that is kept open for the life of the client-server interaction. Either party may close the connection when it wishes to be rid of the other, although typically we expect the logical clients to be the parties to make this determination. There are three valid CSETP messages:

- The time-request message, which consists of a single integer (guaranteed by the sender to fit in 32 bits), followed by a newline. This message is sent by the logical client to request a clock reading, and the integer is a sequence number to be echoed back with the response.
- The time-response message, which consists of three whitespace-separated integers, the 32-bit sequence number, a 32-bit tier number, and a 64-bit timestamp representing milliseconds since January 1, 1970, in that order, all terminated by a newline. CSETP supports only millisecond-resolution clocks. Note that the tier number describes the *server's* tier; the client must add one to get its own tier number. Servers may not change the tier number they send during a session lifetime.
- The server-initiation message, which consists of a single string specifying the email address of the server owner, terminated, as always, with a newline. This string may not contain whitespace, and may be no longer than 256 bytes. See below for a description of server-initiation.

All messages are newline-terminated, and use whitespace to separate parts. Following the normal network convention, newlines are defined to be CRLF (`0x0d` followed by `0x0a`). Whitespace may be any combination of spaces (`0x20`) and tabs (`0x09`), although you may rely on the fact that my server sends only a single tab character if you so desire. Most high-level libraries should handle all of this conversion (particularly newline conversion) internally, so I don't expect any problems here. The Internet motto is "be generous in what you accept and conservative in what you send," and this is a good rule of thumb here.

Client-initiation using TCP

Client-initiation follows the normal model of TCP/IP client-server interaction. The server binds to a well-known port (the default is `0x5347`—decimal 21319, ASCII "SG") and waits for clients to connect. Clients open a connection using any local port number.

In client-initiation, the client is always the first to speak once the connection is open. It sends time-request messages, and the server responds one-for-one with time-response messages. The two directions of a TCP connection are independent, so there is no rule that one request must be answered before another is sent. However, requests are always answered in order, and, because this is TCP, there is no notion of abandoning a request which been lost.

Servers should only accept valid time-requests, and should respond promptly with valid time-responses. They should not accept any other messages, and should not speak unless spoken to. Clients should accept only valid time-responses, and should not respond to them at all. Clients may send valid time-requests at any time, but should not send any other message types. Behavior in the face of unexpected messages is undefined; they may be ignored or the connection may be closed.

Server-initiation using TCP

Server-initiation turns the connection-establishment around. Here the *client* binds to a well-known port number (the default is 21320), and waits for servers to connect. Servers use an out-of-band mechanism (network-speak for the user) to know what clients would "like" to synchronize with them.

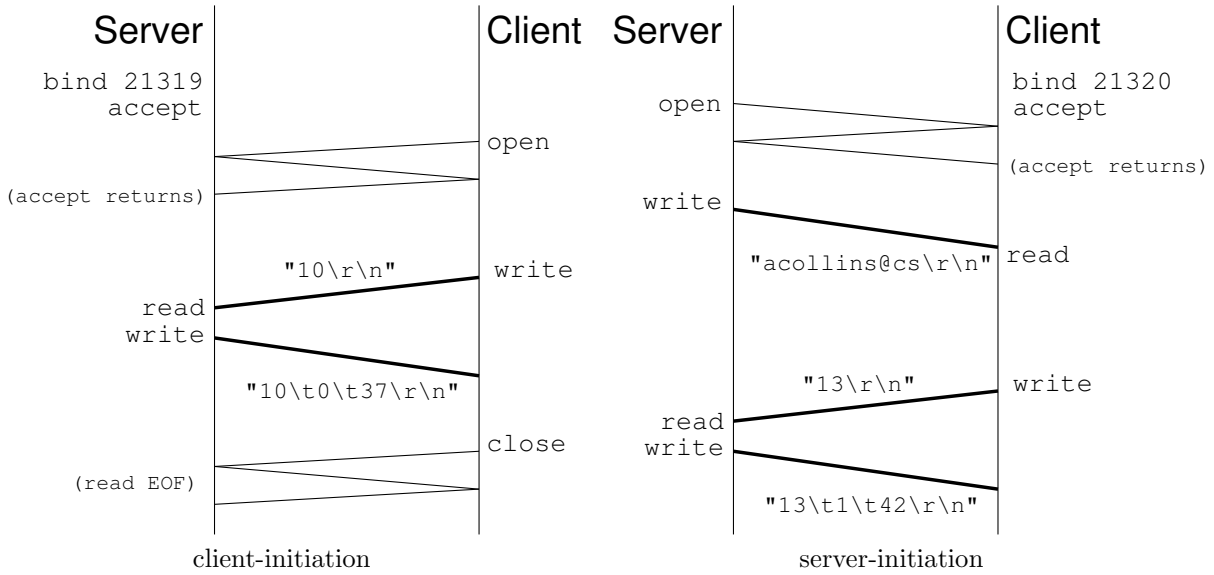

Figure 1: Initation timelines. Time flows down in these diagrams. the grayed-out messages are TCP protocol stuff you can ignore. The calls made by the client and server are shown at the sides, and the contents of the data messages are shown in C format next to the arrows. Note that you should use your own email address for server-initation, and expect much larger timestamp values. Closing the connection is the same for both schemes.

In server-initiation, the server speaks first once, sending a server-initiation message. The email address in the message may be used by the client to communicate with the server's owner after the connection is closed. When the client receives the server-initiation message it begins to use the connection in the same way that the client-initiation client would—at this point it doesn't matter who opened the connection. Therefore, after the first message, the server will only speak when spoken to.

As with client-initiation, servers should only accept valid time-requests, and should respond promptly with valid time-responses. They should not accept any other messages, and should not speak unless spoken to, *except that they must send one valid server-initiation immediately following connection establishment.* Clients should only accept valid server-initiations and time-responses, and should not respond to either. Server-initiations are only valid immediately following connection establishment. Clients may send valid time-requests at any time *after receiving a valid server-initiation.*

Your mission

Your assignment is to implement a CSETP server, and answer the following questions. Your server should take as an argument the name of the parent server to connect to. If you wish, it can also take an optional argument to specify a different port number to connect to. It should use the DNS lookup facility to resolve that name (you don't have to do this yourself—just use the library function on your system), and then periodically synchronize to the parent to maintain its local tier number and authoritative clock (one of the questions deals with how often we need to synchronize). Your server should support at least the following three interfaces

- A client-initiation client,
- A client-initiation server, and
- A server-initiation server

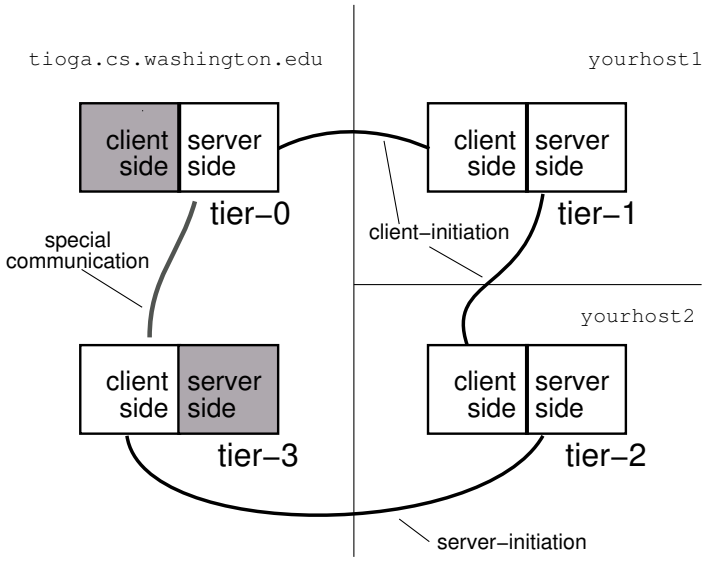

Figure 2: Representation of the test architecture for part 3. Your two servers can run on different machines, as shown, or on one machine. In the latter case, you will need to arrange for them not to conflict in port numbers.

Implementing the server-initiation client is optional. You will use the client-initiation client to synchronize with the master server on tioga, and the server-initiation server to allow the online test facility to synchronize with you. You will also use the client-initiation client and server to synchronize two copies of your own implementation to each other.

1. Measure the latency between your test machine and tioga. What is a reasonable absolute minimum RTT to assume? Note that the CSETP server on tioga adds a minimum 20ms one-way delay (each way) to each message. This is in addition to the minimum network delay. You can either measure the total delay by observing the CSETP messages, or you can use `ping` to measure the network delay and add the 20ms.

   Using this value, and knowing that the ρ for the master server is no worse than 0.01, what is the best precision we can hope to achieve in synchronizing the clock (meaning for a single experiment)?

2. Devise a maximum allowable RTT and a minimum resynchronization period (respectively the largest RTT query/response that can be used to synchronize the clock and the time between resynchronizations) that will ensure that your clock tracks tioga's CSETP time with no more than a 200ms error, based on the observed typical delays and the known ρ. About what fraction of synchronization attempts do you expect will succeed?

   Recall from [Cri89] that this is a fundamental tradeoff: the smaller the max RTT, the more accurate the synchronization but the more likely we will fail to synchronize and have to try again. But the more accurate the synchronization, the longer we can let the clock run before we have to resynchronize.

3. Demonstrate that your CSETP implementation can correctly synchronize to the tier-0 server and pass that authoritative time on to children by using the CSETP online test facility. Set your parameters as described in question 2, so that you will synchronize within 200ms.

   Now set up two servers (on two machines or on two ports of one machine). Server one will connect to tioga as a client. Server two will connect to server one as a client (using the client-initiation interfaces), and will invite tioga to connect to it as a client using the server-initiation interface. With 200ms accuracy at each step, the tioga client should achieve tier-3 status and 600ms precision. See figure 2.

Be sure to put your valid email in the server-initiation message, so that our test client will email you with the results of the test. Our client also closes the connection when the test is complete.

Provided code I have made available, on the course web page, a Java class called `OffsetClock` which implements a clock that can be reset by maintaining an offset to the underlying system clock. It also allows for varying the clock drift (it is, in fact, the code that I use to implement grad-student time), but if you set that to zero it is useful for keeping your own time without changing the system clock. Feel free to use or not use this as you wish.

Languages and platforms This version of CSETP is better suited to a somewhat higher-level language like Java (or, I imagine, C#) than to C or C++. My own implementation is in Java, but I'm confident you could use any of these languages. If you choose to use C#, please take pity on a poor Java programmer and don't try to demonstrate your command of obscure language features. If you want to use something other than C/C++/C#/Java, talk to the TA, and we'll see if we can't work out something mutually acceptable. In terms of platforms, I've developed and tested everything on Linux, but I believe that it should be possible to complete the assignment on most any platform, and I'm hopeful that the online testing routine will make this feasible.

References

[Cri89] Flaviu Cristian. A Probabilistic Approach to Distributed Clock Synchronization. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*, pages 288–296, June 1989.

[NTP03] The Network Time Protocol. `http://www.ntp.org/`, 2003.