CSE461 Project 1: Registration

Out: Monday January 13, 2013
Due: Monday January 27, 2013 by **11:59pm**.
Teams Allowed: Yes
Teams Encouraged: Yes
Ideal Team Size: 2

Summary

In this project you'll write code that interacts with a registration service. The function of the server is to keep track of nodes that participate in some distributed application. When a node comes up, it announces its presence by registering with the service, providing it with an IP address and port it can be contacted at. When a node goes down, it should unregister. Any node can ask the registration service for a list of currently registered nodes. The registration service is therefore a discovery mechanism -- a way for nodes to find each other.

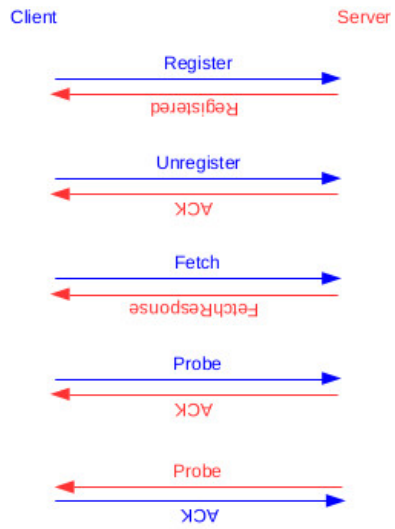
This sounds simple enough, but there are quite a few details required for it to work robustly. Those are described next.

The code is this project is intended to be a utility for the next project. In the next project we build a first cut at Tor routers. Those routers need to find each other. We'll use the registration infrastructure from this project. You'll be implementing this project as a standalone application, but you should keep in mind that you'll want to be including its functionality basically as a library in the next one. It's worth being careful about modularity now to avoid having to rewrite project 1 as part of doing project 2.

Operational Overview

We have implemented the registration service, and run an instance of it at `cse461.cs.washington.edu:46101`. The registration service defines a set of messages, how to encode them as bits, and a set of valid message exchanges (i.e., a protocol). All message encodings fit in a single UDP packet. There are seven kinds of messages. We'll elaborate on their use in a bit.

Communication involving the registration service is always "request-response": one side sends a message to the other, the other responds, and that's the end of it. The possible exchanges are these:



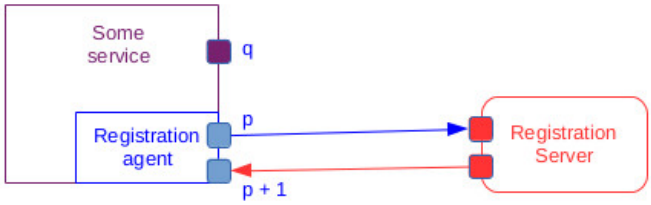
Register registers an instance. The server returns a **Registered** response. **Unregister** removes a registration. The server responds with an **ACK**. **Fetch** requests information about registrations. The server responds with a **FetchResponse** message that contains a few randomly chosen registrations. **Probe** is simply a request for an **ACK** response. It is used to test if the other end is still there. **Probe** may be initiated by the client or by the server. All other exchanges are initiated only by the client.

Note: We have purposefully omitted any kind of error response, in an attempt to contain the work this project requires. If the server detects an error, it simply doesn't respond. That may turn out to be a protocol design mistake though -- time will tell (and, by the end of the quarter, you can be the judge).

Overall Architecture

The client you're writing is really just an agent that supports some other application, which we'll call a service. That other service wants to register with the registration service. Rather than requiring every service to implement code implementing the registration service's protocol, we encapsulate that in the agent. In project 1 there is no actual client service; we'll implement one in project 2, and use the registration agent to register it. We're writing an agent that could register one, though, if one existed.

Here's a picture of the architecture we're headed to:



The small, filled rectangles are sockets, each bound to some port. The arrows show the direction of the first packet sent in a request-response exchange. The response is sent between the same pair of sockets, in the opposite direction, but isn't shown here.

"Some service" wants to register the port its socket is bound to (q) with the registration service. In this project you simply make up some port number to register, since we're not actually implementing a client service at this time.

With that small issue ignored, the rest of the figure is accurate, even in this project. The registration agent itself has a distinct pair of datagram sockets that it uses when communicating with the registration service. The sockets are bound to consecutive ports (p and p+1, for some arbitrary p). One socket is used when the agent wants to initiate a request-response exchange with the registration service. The other is there for the registration service to use if it wants to initiate an exchange. Having separate sockets avoids confusions like: (a) the agent sends a Register to the registration service, and expects to receive an Registered message back; but (b) at the same time, the registration service sends a probe to the agent. If only a single port were used at the agent for both purposes, the agent would read a Probe when it expected a Registered, and get confused. (It's possible to make the one port approach work, but it's clumsy.)

The registration service requires that ports to which the agent's two sockets are bound be consecutive because the agent messages never explicitly give the agent's own port numbers. Instead, the registration service learns the sending port number (p) because it's in the UDP header of the messages it receives from the agent. It deduces the port number it should send Probes to (p+1) because of the offline agreement that it should be one higher than the agent's sending port number.

Summarizing the basic operation, Register:

1. The service wants to register port q, so it asks the agent to do that for it.
2. The agent uses port p to send a Register message to the registration service, which responds with a Registered message sent back to port p.
3. At some later time, the registration service decides to probe the agent, so sends a Probe message to port p+1.
4. The agent responds with an ACK sent to whatever port the registration service used to send the Probe, a value it gets from the header information of the Probe packet.

Message Formats

All messages are sent using UDP, and must fit in a single packet. We describe here their formats. Remember that the UDP packets carry additional information in their headers, though. Of particular interest is the IP address and port of the sender of the UDP packet.

All registration messages begin with a common header consisting of the two-byte value 0xC461, a one-byte unsigned integer sequence number, and a one-byte message type. All multi-byte numeric values are in network order (big endian). Sequence numbers from a single sender form an increasing sequence, except that:

- all response messages (Registered, FetchResponse, and ACK) copy the sequence number from the message to which they are responding,
- if a packet is re-sent, all copies carry the originally assigned sequence number, and
- the sequence numbers (eventually) wrap.

Register

0xC461	seq num	0x01 [Register]	service IP	service port	service data	service name len	service name
2	*1*	*1*	*4*	*2*	*4*	*1*	*service name len*

As well as providing the IP address and port at which the service can be contacted, the service provides two additional pieces of information. The "service data" is an arbitrary 32-bit data item the service wants to publish. It is communicated as an unsigned 32-bit integer in network byte order. The "service name" is a variable length string describing the service. The service name len field gives the length of the service name as an unsigned, 8-bit integer integer.

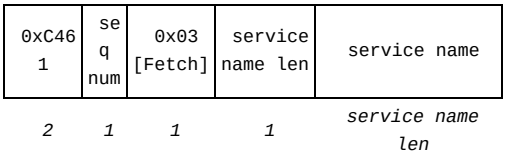
Registered

0xC461	seq num	0x02 [Registered]	lifetime
2	*1*	*1*	*2*

This message is sent as a success response to a Register message. Its sequence number must match the sequence number of the Register message to which it is responding.

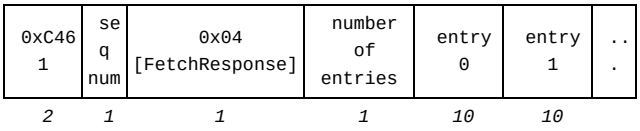
The lifetime field is an unsigned integer value indicating how long the registration service will keep the data it received in the Register message before deleting it. If the client service wants to maintain its registration, it must re-register before the lifetime expires. The lifetime is given in seconds.

Fetch



This message requests that the server send back some, possibly all, existing registrations. Any registrations returned must themselves have service names that begin with the service name specified in this Fetch message. (Specify a service name len of zero to match all registrations.)

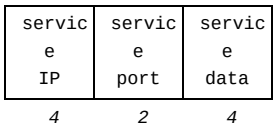
FetchResponse



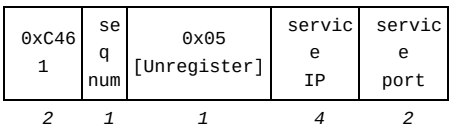
This message is sent as a success response to a Fetch message. Its sequence number must match the sequence number of the Fetch message to which it is responding.

The server tries to return all matching registrations, but imposes a maximum packet length for its response that may cause it to omit some registrations. The packet length limit ensures that the response fits in a single UDP packet, and, further, that there is at least a reasonable chance of delivery (based on Internet behavior observed in some simple experimentation).

Each returned entry looks like this:



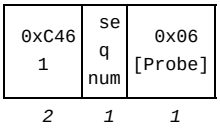
Unregister



Requests that the registration service remove the registration for the specified IP and port. Services should unregister before shutting down. It would be a mistake for the server to rely on them doing so, however. (For that reason the server automatically deletes registrations after a certain amount of time has passed; see Registered.)

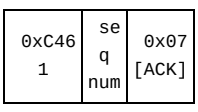
Unregister expects an ACK response on success.

Probe



A Probe message is a request that the other end respond with an ACK. It can be used to test whether or not the other end is still running.

ACK



```
2    1    1
```

A response message. Its sequence number must match the sequence number of the message it is ACK'ing.

Error Handling

Your code should be prepared to deal with both network and registration service errors. Networks errors are lost packets. You don't have to deal with packets arriving corrupted; UDP detects and drops such packets. You also don't have to deal with packets being reordered, or delayed beyond a practically reasonable amount of time. The expected implementations never have more than one packet outstanding at a time, and it is exceedingly rare for packets to be outlandishly delayed and still be delivered.

Service errors are things like responding with the wrong message type or a bad sequence number, or not at all (having crashed, for instance).

When network errors occur, you should try to overcome them. Service implementation errors will presumably just be repeated, so you can give up and report an error if you encounter one.

It's typically hard to debug the code intended to deal with network errors, because they occur infrequently. To help debug, the service instance deployed on `cse461.cs.washington.edu` (explained later) artificially drops some incoming packets. This is done probabilistically. The service simulates bursty errors. At any moment it is in one of two states, one with a low artificial drop rate and one with a very high rate. It transitions from one state to the other at random.

Client UI Specification

As in project 0, you should supply a `run` script that performs any necessary build and then launches your client, supplying it with any command line arguments given to `run`. Your client will be invoked like this:

$ ./run *<registration service host name> <service port>*

Your client should accept commands from `stdin` that cause it to send messages to the registration service identified by the command line arguments, to read its responses the service sends, and to print appropriate messages about the result of the interaction. You must support the following commands. You may also support additional command formats that you find useful. (For example, the sample solution often provides default arguments if required arguments are not provided.)

- `r` *`portnum data serviceName`*
  Send a Register message to the server, using the IP of the machine you're running on and the port, data, and service name given with the command. You should print an indication of whether or not the register was successful. (You don't need to support whitespace in the arguments. I.e., you don't need to support a service name like *461 Project Service*; you can assume that each argument will not contain whitespace, and that whitespace will only be used for delimiting between arguments.)
- `u` *`portnum`*
  Send an Unregister message for your host's IP and the specified port. Print an indication of whether or not the unregister was successful.
- `f` *`<name prefix>`*
  Send a Fetch message to the registration service. Print what it returns, if successful, or otherwise an indication that the Fetch failed.
- p
  Send a Probe to the registration service and display an indication of whether or not it succeeded.
- q
  Quit execution.

Additionally, when your agent is Probed by the service, it should print some indication that it has received the probe.

We aren't too picky about the format of your output, but you should try to make it clear to a TA reading it.

- This page shows the output produced by the sample solution. (Because our solution client shares code with our registration service implementation, we print time stamped output messages. You don't have to.) Note that it often prints nothing as its success indication, a design that we're sure you can improve on!
- This page shows the sample solution's error messages, produced by having it send packets to an IP/port where there is no server running.
- Finally, this page shows that the sample solution client prints an indication whenever it receives a Probe message from the registration service.

Debugging

A registration service is running on host `cse461.cs.washington.edu`, port `46101`. You should expect that its availability will be something less than the five-9's (99.999%) that multi-billion dollar corporations strive for.

It can be useful when debugging your client code to have some idea what the server is receiving and sending, and what registrations it holds. For that reason, the registration service provides a web interface:

`http://cse461.cs.washington.edu:8080/status`
Displays the current registration data.

`http://cse461.cs.washington.edu:8080/log`
The server produces a log of activity, including incoming packets shown as hex strings and as parsed messages. This page shows the most recently written (up to) 250 lines of the log.

Turn in

When you're ready to turn in your assignment, do the following:

1. The files you submit should be placed in a directory named `proj1`. There should be no other files in that directory.
2. Create a README.TXT file that contains the names, student numbers, and UW email addresses of the member(s) of your team.
3. Put the README.TXT file, your project 1 solution source code, and your `run` script in the `proj1`directory.
4. While in the directory that is the parent of `proj1/`, issue the command `tar czf proj1.tar.gz proj1`.
5. Verify that the tar file contains the files you intend to submit: `tar tf proj1.tar.gz`.
6. Submit the `proj1.tar.gz` file to the course dropbox. (There's a link to the dropbox in the navigation section of all course pages.)

Computer Science & Engineering University of Washington Box 352350 Seattle, WA 98195-2350 (206) 543-1695 voice, (206) 543-2969 FAX