CSC323 2010S *Software Design*

Class 17: Sockets

Held: Tuesday, April 6, 2010

Summary: We consider Sockets, one of the simple standard mechanisms of program communication.

Related Pages:

- EBoard.

Notes:

- Welcome back! I hope you had a great break.
- For Thursday: Please skim An Introductory 4.4BSD Interprocess Communication Tutorial and submit two questions.
- Please work hard on your projects. (Remember, at least ten hours per person, unless you've arranged with me and your colleagues in advance.)
- EC for Today's CS and Disability talk (4:30 in Science 3821).
- EC for Wednesday's CS and Disability talk (4:15 in JRC101).
- EC for Thursday's convocation.
- EC for Thursday's "Teaching Millenials" CS Extra (4:30 in Science 3821).
- EC for Friday's "Computational Games" CS Extra (noon in Science 3821; Free Pizza; I need an approximate count).

Overview:

- Sockets: Background.
- Building Server Sockets.
- Building Client Sockets.
- Sending and Receiving Data.
- Avoiding Blocking.
- Detour: Representing Numbers.
- Exercises.

Sockets: Some Background

- Sockets use a client-server model.
  - Servers provide socket connections, clients bind to those connections.
- Once bound, sockets are symmetric: Both client and server treat them more or less as "files" that can be read or written.
- Typically, sockets are streams of data; you need to decide on the protocol that determines when a message is finished.
- A long-standing technology for network communication.

- Introduced in BSD Unix. (4.2, I think, but perhaps earlier.)
- Originally implemented in C.
- Sockets are clearly tied to the TCP/IP model.
- Sockets existed when I was your age.
- But they were relatively new then.

- What does it mean that it's long-standing.
  - It clearly works well (in this case, it's simple and powerful)
  - It may have some difficulty adapting to all of the demands of modern systems.
  - It's relatively low-level.

Creating Server Sockets

- On the server side, you typically need to do four things for an Internet stream socket
  - Create a primary socket to use for setting up connections.
  - Bind that socket to a port (so that a client knows which service it is connect to)
  - Wait until connections are available
  - Accept connections from that socket
- When you accept a connection, you get another socket, which you then use to communicate with the client.
- In Python, we write

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), port))
s.listen(5)  # 5 is how many pending connections we support
(s2,address) = s.accept()
```

- Note that the call to `accept` is a *blocking* call. Your program pauses until a connection is made.
  - We'll come back to how to deal with this issue in a bit.

Client Sockets

- It's a bit easier to create client sockets. There are only two steps.
  - Create a socket.
  - Connect to the server on a particular port.
- In Python, we write

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((hostname, port))
```

- As you might guess, `connect` is typically a blocking operation (although it can timeout).
- Note that `connect` takes *one* parameter. In this case, it's a two-element tuple of host/port.

Communicating with Sockets

- The two basic operations are `send` and `recv`
- `send` takes something of type "bytes" as a parameter.
  - Given a string, you convert it to bytes with the `encode` method.
- `send` returns the number of bytes sent (which will be less than or equal to the number of bytes you attempted to send.
- `recv` takes the maximum number of bytes to receive. It returns a bytes object.
  - You may get less data than you expect.
  - You can convert the bytes back to a string with the `decode` method.
- Some implementations of `send` buffer data before sending. The data are sent when the buffer fills or when you explicitly flush the socket.
- When you're finally done with a socket, be nice and close it with `close`.

Avoiding Blocking

- As you've noted, `recv` and `listen` (and even `send`) are potentially blocking operations.
- For modern programs, blocking operations are considered potentially harmful.
- What's the solution?
  - Do blocking operations in a separate thread.
  - Hope that the library provides a non-blocking version of the operation.
  - Hope that the library provides a "will this block" predicate.
- We'll leave threads to another day.
- You can make sockets nonblocking with `setblocking(0)`
- You can make sockets parially blocking with `settimeout()`
- But it's actually better to use the `select` procedure provided by the `select` module.
- `select` takes four parameters
  - A list of sockets (or other file descriptors) from which you might read (or accept connections)
  - A list of sockets (or other file descriptors) to which you might write
  - A list of sockets that might have errors (we tend to skip these)
  - A timeout to wait
- `select` returns a tuple of three lists
  - The sockets which are available for reading
  - The sockets which are available for writing
  - The error sockets
- You can then iterate through those sockets, doing what you want with them.

Detour: Dealing with Numeric Data

- How can we efficiently send numbers?
  - `struct.pack`
- How do we handle differences in internal representations?
  - `socket.htonl`

- `socket.ntohl`

Activities

One-shot responses

- Write a server that
  - Accepts a connection
  - Reads string of length up to 128 from that connection
  - Generates an “interesting” response string (e.g., you can prefix the string with “Hello”, play “The Name Game”, or whatever.
  - Sends back the response.
  - Closes the connection.
- Write a client that
  - Gets a string from the user
  - Creates a connections
  - Sends the string over the connection
  - Reads a response from the connection (no more than 256 characters)
  - Prints out the response
  - Closes the connection

One-shot responses, revisited

- Update your previous program so that the client can accept an arbitrary length response from the server

A dialog

- Extend your prior program so that the client can send multiple requests to the server through one connection.
  - The user provides each request (after prompting)
  - The client closes the connection when the user no longer provides input.
- Note that the client still needs to accept arbitrarily long responses from the server.

A numeric dialog

- Rewrite your prior program so that the dialog uses four-byte numbers, rather than strings. For example, your client might send an integer and the server could respond with the square of that integer.
- Please don’t print and parse your strings.
- Note that for safety, you should use `socket.htonl` and `socket.ntohl`.

A chat server

- Write a program in which the server accepts multiple simultaneous connections, reads text from any of the active connections, and sends that text to all of the connections (except the one from which it received).
- If you have time, add support for disconnection.

Copyright © 2010 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/2.5/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.