Design Principles for Distributed Applications

“A Distributed Application is a system in which the crash of a machine you’ve never heard of can cause your program to break.”

~ Famous Quote, sometimes attributed to Peter Deutsch
Design Principles for Distributed Applications

- First step: getting the protocol right
- What’s the right protocol?
- One that’s:
  - Inherently reliable: either party can tell if something’s gone wrong
  - Easily parseable: simple to write clients and servers
  - Highly efficient: requires sending as few messages as possible
  - Structurally simple: easy to debug
The Seven Fallacies of Distributed Computing

- Definitely attributed to Peter Deutsch
- Assumptions that people make that result in bad distributed applications:
  - The network is reliable
  - Latency is zero
  - Bandwidth is infinite
  - The network is secure
  - Topology of the network doesn’t change
  - There is one administrator
  - Transport cost is zero
## What This Means in Practice:

<table>
<thead>
<tr>
<th>The network is reliable</th>
<th>Don’t assume that data you send will be received, that a server or client you’re talking to will always be alive, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency is zero</td>
<td>It takes time for a recipient to get your message! Don’t assume transmission is instantaneous.</td>
</tr>
<tr>
<td>Bandwidth is infinite</td>
<td>You must make messages as small as possible.</td>
</tr>
<tr>
<td>The network is secure</td>
<td>Don’t assume that clients that connect to you will have good intentions. Guard against evil!</td>
</tr>
<tr>
<td>Topology of the network doesn’t change</td>
<td>Don’t assume that hosts will retain the same IP address; don’t assume that routes between hosts won’t change</td>
</tr>
<tr>
<td>There is one administrator</td>
<td>You can’t “reboot” the Internet, to upgrade clients and services to a new protocol at the same time.</td>
</tr>
<tr>
<td>Transport cost is zero</td>
<td>Sending bytes across a network is expensive, relative to doing local computation. Trade computation for transmission whenever possible.</td>
</tr>
</tbody>
</table>
Why It’s Important to Get It Right the First Time!

- Once you deploy a service (for real), the protocol is largely set in stone
  - Why? Because if you change it, you have to change every other client in existence to speak the new protocol
  - HTTP is only at version 1.1 (which came out early...), and will likely never see 1.2
- The protocol, to a large extent, determines what you can build on top of it
  - Example: in our IM system, impossible to (easily) send text before the chat is created... because the protocol doesn’t support it
  - Example: in most email systems, impossible to “retract” a sent message... because the SMTP protocol doesn’t support it
  - You can fake certain things, but often difficult. The underlying infrastructure constrains what you can build on top of it
Common Protocol Design Idioms

- **Request/response/notification**
  - Client sends a *request*
  - Server replies with a *response*
  - Server may also send *notification* asynchronously

- Often uses a *sequence number* at the front to allow easy message processing, pipelining of requests and responses

- **Example:**
  - Client sends request 101, request 102, request 103, ...
  - Server replies with response 101, response 102, response 103, ...
  - Allows clients to have multiple requests “in flight” at once, pair up responses as they come in
Common Idioms for Delimiting Messages

- **“Stuffing”**
  - Example: SMTP: each header on a line by itself, multiline data begins with “DATA” and ends with “.”
  - Client: you don’t need the entire message assembled before you start sending it
  - Server: easy to process, but also slow to process: you have to “look” at every line to see if you’re at the end.

- **“Counting”**
  - Example: HTTP: messages indicate how many bytes they contain
  - Client: need to know the length of the entire message before you can send it
  - Server: don’t need to examine every byte to process it; just get the length and read this much

- **“Blasting”**
  - Example: FTP: open an entirely new socket for sending a file; blast the file across and then close it
  - Client: no need for parsing at all
  - Server: no need for parsing at all. Expensive if you’re sending lots of small files though (need to set up, tear down socket for each one)
Common Idioms for ASCII Encoding

- Most ASCII-encoded protocols are *line oriented*
- Example: SMTP
  - Textual headers (FROM, SUBJECT, DATE, etc.)
  - Each header consists of *name : value* followed by a carriage return
  - Message body starts with DATA, then message body, then a “.” on a line by itself to terminate
  - If message body contains “.” on a line by itself anyway, it’s replaced by “..” and then decoded on the receiver
- HTTP uses essentially the same format
- Very easy to parse, as long as you’re not sending complex data types
- Very easy to debug
Common Idiom: Reply Codes

- Reply messages from a server often contain structured codes to indicate what happened.
- Defined as a part of the protocol spec, intended to allow very easy parsing
- SMTP: 3 digit codes at start of replies
  - 1st digit: success or failure
  - 2nd digit: the subsystem of the mail server that is responding
  - 3rd digit: the situation that occurred
- HTTP: same deal
  - Error 404 anyone?
Debugging ASCII-Oriented Protocols

- You can telnet to a server that speaks an ASCII protocol to talk to it directly

```
telnet www.cc.gatech.edu 80
Trying 130.207.7.237...
Connected to rhampora.cc.gatech.edu.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Wed, 16 Feb 2005 14:55:22 GMT
Server: Apache/2.0.46 (Unix) mod_ssl/2.0.46 OpenSSL/0.9.7a
Last-Modified: Mon, 14 Feb 2005 12:01:38 GMT
ETag: "e97772-7573-69118c80"
Accept-Ranges: bytes
Content-Length: 30067
Connection: close
Content-Type: text/html

<HTML><HEAD><TITLE>Georgia Tech - College of Computing</TITLE><META
```

Blank line indicates end of request
Protocol Design Techniques

- “Lo-fi prototypes for protocols”
- Fence sketches
- Shows interactions among hosts involved in a protocol exchange
- Time starts at the top of the sketch, goes down
- There’s even software to create these for you
Examples

1. Client \rightarrow Server: INVITE
2. Server \rightarrow Client: INVITATION
3. Client \rightarrow Server: SEND_MESSAGE
4. Server \rightarrow Client: MESSAGE
5. Client \rightarrow Server: MESSAGE
Dealing With Errors
Networking Errors, Other Errors

- From last week: *defensive programming*
  - Would like to ensure that a server can’t crash your client
  - Would like to ensure that a client can’t crash your server
- How do you do this?
- One good tool: *exceptions*
- Built into the Jython language
  - Also available in some form in many other languages: Java, C++, ...
What Are Exceptions?

- A way to skip out of an arbitrarily large chunk of code when an error happens
- Sort of a structured “super-goto”
- Any sort of runtime error that happens in your program may raise an exception
  - Meaning: it’s telling you that something has gone wrong
- By default, exceptions are not caught
  - Meaning: they simply cause your program to quit
- Jython provides a way for you to catch these exceptions, and handle them with your own code
  - Meaning: you can write code to recover from errors that may occur
- Exceptions are not just for network programming! All sorts of errors cause exceptions to be raised!
Example

```
list = [1, 2, 3]
print list[158]
```

Traceback (innermost last):
  File "<console>", line 1, in ?
IndexError: index out of range: 158

- This message is telling you than an exception--called `IndexError`--was raised, but not caught
Example with Exceptions

list = [1, 2, 3]
try:
    print list[158]
except IndexError:
    print "Dummy! You used a bogus index!"

Dummy! You used a bogus index!
Anatomy of an Exception

- **New keyword:** *try*
  - Specifies the start of a block of code that might cause exceptions you’d like to handle

- **New keyword:** *except*
  - Specifies the end of the block of code that might cause exceptions, and the beginning of your *exception handler*

- There are different types of exceptions.
  - Example: *IndexError* caused by list index out of bounds
  - Other operations define their own types of exceptions
  - You specify in the *except* statement which types you’re handling

- If an exception is raised, control passes to the handler, and then to the next statement after that

- If no exception is raised, control continues to the except keyword, then skips the except clause, then continues to the next statement after that
What Good Does This Do You?

• In the previous example, not much
  • The index problem was a *logic* error, caused by an actual bug in the program
  • Pretty much you’d just want to exit; the developer will need to find and fix the bug to make the program right

• Other sorts of exceptions are *not* caused by logic errors though

• Examples:
  • You’re trying to connect to a server and the server is down
  • You’re trying to write to a server and it crashes

• Caused not by bugs in your program, but by changes in the external situation
More Exception Syntax

- Syntax of `except` statement:
  - `except ExceptionType, exceptionInstance`

- `except`: keyword in the Jython language

- `ExceptionType`: indicates the type (class name) of the exception you want to catch
  - Can just use “Exception” as the type to catch everything
  - Can have multiple `except` statements, each of which catches a different type of exception

- `exceptionInstance`: an object that provides more details about the exception
  - Optional
  - Use it to print out more details about what specifically went wrong
A More Realistic Example

```python
import sys
import java.net as net

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

success = 0
while not success:
    # let the user type in a hostname. They might type something bogus!
    hostname = sys.stdin.readline()
    try:
        s.connect((hostname, 80))  # try to connect
        success = 1  # if no exception, we were successful
    except net.UnknownHostException:
        # if the name is bogus, we’ll get an exception
        print "The hostname you entered," hostname, "is not valid."
```

Detecting Read/Write Errors

- socket.send(), socket.recv() raise java.net.SocketException if they fail

```java
import java.net as net
try:
    socket.send(”hello”)
except net.SocketException, ex:
    socket.close()
    print “Error while trying to send data:” + str(ex)
# do whatever other cleanup is necessary here
```
Good Design When Using Exceptions

- In general, wrap any operations that commonly fail: opening files, socket calls, etc.
- If there’s no way you could ever possibly recover--and the only suitable response is to exit--then you could just let the default exception handler be used
- If you’ve got a function that might raise lots of exceptions, may be better to wrap the call to the function, rather than having lots of handlers inside the function
- Don’t catch too much: you don’t want to catch exceptions that flag programmer errors