notes Documentation

Release 1.0

Amit Uttamchandani

February 28, 2017

Contents

The following pages contain my notes and code snippets accumulated over the years. They are categorized into the following sections:

CHAPTER 1

Networking

Networking

HTTP

Contents

1. Hypertext Transfer Protocol. Based on RFC 822/MIME format.
2. For transferring binary it uses *base64* since HTTP is a protocol used to transfer text. To transfer binary it must be encoded as text and sent out. This is what *base64* is used for. It encodes 4 characters per 3 bytes of data plus

padding at the end. Thus, each 6 bits of input is encoded in a 64-character alphabet (efficiency is 4/3 = 1.333 times original).

3. HTTP 1.1 added persistent connections, byte ranges, content negotiations, and cache support.
4. Note that HTTP's protocol overhead along with connection setup overhead of using TCP can make HTTP a poor choice for certain applications. In these cases, UDP is recommended (for example DNS uses simple UDP requests/responses for most of the DNS queries). Can mitigate some of the overhead by using persistent HTTP connections.

Basic format for requests/responses:

```
message = <start-line>
          (<message-header>)*
          CRLF
          [<message-body>]

<start-line> = Request-Line | Status-Line
<message-header> = Field-Name ':' Field-Value
```

Request format:

```
Request-Line = Method SP(Space) URI SP(Space) HTTP-Version CRLF
Method = "OPTIONS"
       | "HEAD"
       | "GET"
       | "POST"
       | "PUT"
       | "DELETE"
       | "TRACE"
```

```
GET /articles/http-basics HTTP/1.1
Host: www.articles.com
Connection: keep-alive
Cache-Control: no-cache
Pragma: no-cache
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

Response format:

```
Status-Line = HTTP-Version SP(Space) Status-Code SP(Space) Reason-Phrase CRLF
```

```
HTTP/1.1 200 OK
Server: nginx/1.6.1
Date: Tue, 02 Sep 2014 04:38:20 GMT
Content-Type: text/html
Last-Modified: Tue, 05 Aug 2014 11:18:35 GMT
Transfer-Encoding: chunked
Connection: keep-alive
Content-Encoding: gzip
```

Query String

Notes from: Wikipedia - Query string and' RFC 3986 - Uniform Resource Identifier (URI).

1. Part of the URL that does not really fit in a path hierarchy. Example: `http://example.com/over/there?name=ferret`
2. Server may pass the query string to a CGI (Common Gateway Interface) script.

3. The `?` separates the resource from the query string. Example for Google search: `https://www.google.com/?gws_rd=ssl#q=test`. Thus, the URL can be bookmarked and shared.
4. Usually used to store content of web forms.
5. The format is key,value pairs. Series of pairs usually separated by `&`.
6. The # is known as a fragment.

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
   |   _____________________|__
  / \ /                        \
  urn:example:animal:ferret:nose
```

Chunked Transfer Encoding

1. New feature of HTTP/1.1.
2. According to RFC 2616: *The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator.*
3. Used to transfer dynamically produced content more efficiently.
4. Uses *Transfer-Encoding* header instead of *Content-Length* header. Since there is no content length header, server can start sending response as it gets content.
5. Size of each chunk is sent right before chunk so receiver knows when it has completed receiving chunks.
6. Data transfer is terminated by chunk of length 0.
7. Advantages, for example, is when response starts sending HTML page to browser (start with <*head*>, which includes external scripts location), so browser can start downloading this scripts in parallel.
8. Sometimes you want to upload data but don't know the length of data yet. A good use of this feature would be performing a database dump, piping the output to gzip, and then piping the gzip file directly to Cloud Files without writing the data to disk to compute the file size.
9. Example is below. Note that the first chunk has size **0x45ea** which is **17898 bytes**. Then last chunk is **0** length.

```
C: GET / HTTP/1.1
Host: www.google.com
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: HTTPie/0.8.0

S: HTTP/1.1 200 OK
Date: Thu, 16 Oct 2014 04:16:25 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie:
PREF=ID=26f17b4e26a810fd:FF=0:TM=1413432985:LM=1413432985:S=ZtumMxEG9KJAGJDr;
expires=Sat, 15-Oct-2016 04:16:25 GMT; path=/; domain=.google.com
Set-Cookie: NID=67=PW5SAvG5XSS2ptSNeN6WfK11dy7qJxM3MM7sRvn_M3CPp6zdr_QihMyA66yTEt47n1PZyGHvIVv_9ecJW2
expires=Fri, 17-Apr-2015 04:16:25 GMT; path=/; domain=.google.com; HttpOnly
P3P: CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657
```

```
for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic,p=0.01
Transfer-Encoding: chunked

45ea
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" ...

...
</script></div></body></html>
0
```

Codes

Summary:

Code	Classification
1xx	Informational
100	Continue
2xx	Success
200	OK
3xx	Redirection
301	Moved Permanently
302	Found
4xx	Client Error
401	Unauthorized
403	Forbidden
404	Not Found
5xx	Server Error
500	Internal Server Error

Examples

Code 301 Redirection

An example of this is when requesting a certain snapshot from the debian archives. Let's request for a date *(January 02, 2012 22:05:11) 20120102T220511Z*:

```
$ http --headers get http://snapshot.debian.org/archive/debian/20120102T220511Z/pool/main/b/bash/
HTTP/1.1 301 Moved Permanently
Accept-Ranges: bytes
Age: 0
Cache-Control: public, max-age=600
Connection: keep-alive
Content-Encoding: gzip
Content-Length: 224
Content-Type: text/html; charset=UTF-8
Date: Wed, 01 Oct 2014 18:36:27 GMT
Expires: Wed, 01 Oct 2014 18:46:26 GMT
Location: http://snapshot.debian.org/archive/debian/20120102T214803Z/pool/main/b/bash/
Server: Apache
Vary: Accept-Encoding
```

```
Via: 1.1 varnish
X-Varnish: 1485917301
```

Notice that we get back a *301* code that stands for redirection. We then get redirected to *http://snapshot.debian.org/archive/debian/20120102T214803Z/pool/main/b/bash/*.

Code 302 Found

Indicates resource resides temporarily under a different URI (10.3.3 302 Found).

```
$ http get amits-notes.readthedocs.org
  HTTP/1.1 302 FOUND
  Connection: keep-alive
  Content-Language: en
  Content-Length: 0
  Content-Type: text/html; charset=utf-8
  Date: Tue, 14 Oct 2014 18:37:30 GMT
  Location: http://amits-notes.readthedocs.org/en/latest/
  Server: nginx/1.4.6 (Ubuntu)
  Vary: Accept-Language, Cookie
  X-Deity: chimera-lts
  X-Fallback: True
```

Methods

GET

Fetch a resource. Example in python:

```
def get():
# Simple GET of index.html
headers = { 'User-Agent': 'http_client/0.1',
            'Accept': '*/*',
            'Accept-Encoding': 'gzip, deflate' }
http_conn = http.client.HTTPConnection("localhost")
http_conn.set_debuglevel(1)
http_conn.request("GET", "/", headers=headers)

## Response
resp = http_conn.getresponse()
print()
print("Status:", resp.status, resp.reason)

## Cleanup
http_conn.close()
```

Difference Between POST and PUT

1. POST is used for creating, PUT is used for updating (and creating). It's also worthwhile to note that PUT should be idempotent whereas POST is not.
    - Idempotent means that same request over and over has same result. Thus, if you are doing PUT and connection dies, you can safely do a PUT again.
2. Also, according to HTTP/1.1 spec:

The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI."

3. Thus, *POST* can be used to create. PUT can be used to create or udpate.
4. Difference is in terms of API calls. You usually do a *POST* to an API endpoint (or a URL that already exists).

```
POST https://www.googleapis.com/dns/v1beta1/projects/project/managedZones

{
    parameters*
}
```

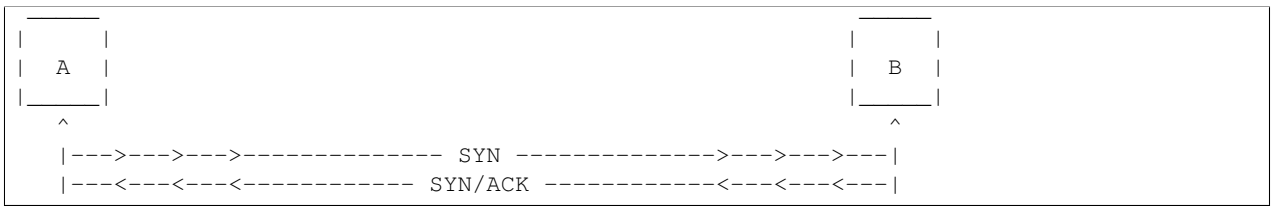
5. With *PUT* you actually create a valid path under the URL:

```
PUT /questions/<new_question> HTTP/1.1
Host: wahteverblahblah.com
```

6. Thus, you use PUT to create the resource and then use that URL for POST.
7. Note that with POST, server decides new URL path, with *PUT* user decides.

Persistent Connections

1. Uses *Connection: keep-alive* header request/response header.
2. Idea is to use single TCP connection to send and receive multiple HTTP Requests/Responses. Thus, avoiding expensive TCP handshake.
3. This is default in HTTP/1.1.
4. Disadvantages when single documents are repeatedly requested (e.g. images). This kills performance due to keeping unnecessary connections open for many seconds after document was retrieved.
5. When you set up a TCP connection, you associate a set of timers. Some of the timers are used for keepalive.
6. A Keepalive probe is a packet with no data and ACK flag turned on.
    - Note that in TCP/IP RFC, ACK segments with no data are not reliably transmitted by TCP. Thus, no retries.
    - Remote host doesn't need to support keepalive. It will see an ACK packet and send back an ACK reply.
7. Since TCP/IP is a stream oriented protocol, a zero length data packet is not dangerous for user program.
8. If no reply packets are received for keepalive probe, can assume that connection is broken.
9. Also useful when NAT terminates connection since it only can keep track of certain number of connections at a time.
10. Useful to know if peers have died before notifying you (e.g. kernel panic, reboot).



```
   |--->--->--->-------------- ACK --------------->--->--->---|
   |                                                          |
   |                                        system crash ---> X
   |
   |                                      system restart ---> ^
   |                                                          |
   |--->--->--->-------------- PSH --------------->--->--->---|
   |---<---<---<-------------- RST ---------------<---<---<---|
   |                                                          |
```

References:

1. TCP Keepalive HOWTO
2. Wikipedia - HTTP Persistent Connection
3. RFC 1122 Section 4.2.3.6 - TCP Keep-Alives

keepalive in Linux

Default is two hours before starting to send keepalive packets:

```
# cat /proc/sys/net/ipv4/tcp_keepalive_time
7200

# cat /proc/sys/net/ipv4/tcp_keepalive_intvl
75

# cat /proc/sys/net/ipv4/tcp_keepalive_probes
9
```

To add support to your application use *setsockopt()* and configure the socket connection for keepalive.

Can also use libkeepalive with *LD_PRELOAD* to add support to any C application.

Document Caching

From: Google Browser Security Handbook, Part 2

1. HTTP requests are expensive mainly because of overhead of setting up TCP connections. Thus, important to have the browser or intermediate system (proxy) maintain local copy of some of the data.
2. The HTTP/1.0 spec did define some headers to handle caching but it did not provide any specific guidance.
   - *Expires*: This is a response header that allows server to declare an expiration date. When this date is passed, browsers must retrieve new document. There is a *Date* header as well which defines the date and time which message was originated. Sometimes, however, *Date* header is not part of response. Thus, implementation is then browser specific.

     The RFC also does not specify if the *Expires* is based on browser's local clock. Thus, current practice is to compute *Expires-Date* delta and compare it to browser clock.
   - *Pragma* request header when set to *no-cache* permits clients to override intermediate systems to re-issue requests rather than retrieve cached data. For *Pragma* response header, it instructs browser not to cache this data.
   - *Last-Modified* response header indicates when resource was last updated according to server's local clock. Reflects modification date of file system. Used in conjunction with *If-Modified-Since* request header to revalidate cache entries.

- *If-Modified-Since* request header, permitting client to indicate what *Last-Modified* header it had seen on the version of the document already present in browser or proxy cache. If server calculates that no modification since *If-Modified-Since* date it returns *304 Not Modified* response instead of requested document. Thus, client will redisplay cached content.
- All of above was useful when content was static. Thus, with complex dynamic web apps, most developers turned off caching.

3. HTTP/1.1 acknowledges the issue and establishes ground rules for what and when should be cached.
   - Only 200 (*OK*), 203 (*Non-Authoritative*), 206 (*Partial Content*), 300 (*Multple Choices*), and 301 (*Redirection*) responses are cacheable, and only if the method is not POST, PUT, DELETE, or TRACE.
   - *Cache-Control* header introduced that provides a fine-grained control over caching strategies.
     - *no-cache* disables cache all together. Can disable cache for certain specific headers as well (e.g. *no-cache: Set-Cookie*).
       - Firefox still stores responses because of back and forward navigation between sessions. But it doesn't do this on *https* connections because of sensitive information such as banking, etc.
     - *no-store*: If in request don't store any request response in cache. If sent in response, client must not store anything from request/response headers.
     - *public/private*: Controls caching on intermediate systems.
     - *max-age*: Time to live in seconds.

Authentication

Basic Auth

This is the simplest form of authentication since it doesn't require cookies, session identifier or login pages. It uses standard HTTP *Authorization* header to send login credentials. Thus, no handshakes need to be done.

Typically used over *https* since encoding is done in *base64* (passwords sent as plain text). Passwords can be easily decoded.

On *Server*, status code 401 is sent back and the following header is used:

```
WWW-Authenticate: Basic realm="Restricted"
```

On *Client*, the *Authorization* header is used with the following format:

```
Authorization: Basic base64("username:password")
```

Example in python:

```
def get_auth():
# GET with authorization of index.html
authstring = base64.b64encode(("%s:%s" % ("amit","amit")).encode())
authheader = "Basic %s" % (authstring.decode())
print("Authorization: %s" % authheader)

headers = { 'User-Agent': 'http_client/0.1',
            'Accept': '*/*',
            'Authorization': authheader,
            'Accept-Encoding': 'gzip, deflate' }
http_conn = http.client.HTTPConnection("localhost")
http_conn.set_debuglevel(1)
http_conn.request("GET", "/", headers=headers)
```

```
## Response
resp = http_conn.getresponse()
print()
print("Status:", resp.status, resp.reason)

## Cleanup
http_conn.close()
```

Digest

Basically uses MD5 of password and *nonce* value to prevent replay attacks. Now, pretty much replaced by HMAC (keyed-hash message authentication code).

A basic digest authentication session goes as follows:

1. HTTP client performs a request (GET, POST, PUT, etc)
2. HTTP server responds with a 401 error not authorized. In the response, a *WWW-Authenticate* header is sent that contains:
   - *Digest algorithm* - Usually *MD5*.
   - *realm* - The access realm. A string identifying the realm of the server.
   - *qop* - Stands for quality of protection (e.g. *auth*)
   - *nonce* - Server generated hash, issued only once per *401* response. Server should also have a timeout for the nonce values.
3. Client then receives the 401 status error and parses the header so it knows how to authenticate itself. It responds with the usual header and adds an *Authorization* header containing:
   - *Digest username*
   - *realm*
   - *nonce* - Sends the server generated value back.
   - *uri* - Sends the path to the resource it is requesting.
   - *algorithm* - The algorithm the client used to compute the hashes.
   - *qop*
   - *nc* - hexadecimal counter for number of requests.
   - *cnonce* - client generated nonce, always is generated per request.
   - *response* - Computed hash of `md5(HA1:nonce:nc:cnonce:qop:HA2)`.
     - HA1 = `md5(username:realm:password)`
     - HA2 = `md5(<request method.:uri)`

   Notice how the client does not send the password in plain text.
4. Server computes hash and compares to client's hash and if it matches sends back *OK* with content. Note that *rspauth* sent back by server is a mutual authentication proving to client it knows its secret.
5. *Note* that each client needs to know the password and the password needs to be shared securely before hand.

Example HTTP Capture:

```
C:
GET /files/ HTTP/1.1
Host: localhost
User-Agent: http_client/0.1
Accept-Encoding: gzip, deflate
Accept: */*

S:
HTTP/1.1 401 Unauthorized
Server: nginx/1.6.1
Date: Sat, 06 Sep 2014 02:09:24 GMT
Content-Type: text/html
Content-Length: 194
Connection: keep-alive
WWW-Authenticate: Digest algorithm="MD5", qop="auth", realm="Access Restricted", nonce="2a27b9b6540a6

C:
GET /files/ HTTP/1.1
Host: localhost
User-Agent: http_client/0.1
Accept-Encoding: gzip, deflate
Accept: */*
Authorization: Digest username="amit", realm="Access Restricted", nonce="2a27b9b6540a6cd4", uri="/fil
response="421974c0c2805413b0d4187b9b143ecb", algorithm="MD5", qop="auth", nc=00000001, cnonce="e0819

S:
HTTP/1.1 200 OK
Server: nginx/1.6.1
Date: Sat, 06 Sep 2014 02:09:24 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Authentication-Info: qop="auth", rspauth="33fea6914ddcc2a25b03aaef5d6b478b", cnonce="e08190d5", nc=0
Content-Encoding: gzip
```

Example Python Code:

```
def get_auth_digest():
    resp = get()

    # Get dictionary of headers
    headers = resp.getheader('WWW-Authenticate')
    h_list = [h.strip(' ') for h in headers.split(',')]
    #h_tuple = re.findall("(?P<name>.*?)=(?P<value>.*?)(?:,\s)", headers)
    h_tuple = [tuple(h.split('=')) for h in h_list]
    f = lambda x: x.strip('"')
    h = {k:f(v) for k,v in h_tuple}
    print(h)

    # HA1 = md5(username:realm:password)
    ha1_str = "%s:%s:%s" % ("amit",h['realm'],"amit")
    ha1 = hashlib.md5(ha1_str.encode()).hexdigest()
    print("ha1:",ha1)

    # HA2 = md5(GET:uri) i.e. md5(GET:/files/)
    ha2_str = "%s:%s" % ('GET',path)
    ha2 = hashlib.md5(ha2_str.encode()).hexdigest()
    print("ha2:",ha2)
```

```
    # Generate cnonce
    cnonce = hashlib.sha1(str(random.random()).encode()).hexdigest()[:8]
    print("cnonce:",cnonce)

    # Generate response = md5(HA1:nonce:00000001:cnonce:qop:HA2)
    resp_str = "%s:%s:%s:%s:%s:%s" % (ha1,h['nonce'],"00000001",cnonce,h['qop'],ha2)
    resp_hash = hashlib.md5(resp_str.encode()).hexdigest()
    print("resp_hash:",resp_hash)

    # Do another get
    authheader = 'Digest username="%s", realm="%s", nonce="%s", ' \
                 'uri="%s", response="%s", algorithm="%s", qop="%s", nc=00000001, ' \
                 'cnonce="%s"' \
                 % ("amit", h['realm'], h['nonce'], path, resp_hash, h['Digest algorithm'], h['qop'],
    print(authheader)
    headers = { 'User-Agent': 'http_client/0.1',
                'Accept': '*/*',
                'Accept-Encoding': 'gzip, deflate',
                'Authorization': authheader
              }
    get(headers)
```

Cookie Based

Cookies are designed to maintain state. Thus, cookie based authentication inherits this stateful principle. Cookie authentication are the most common method used by web servers to know if the user is still logged in or not. The browser keeps sending back the same cookie to the server in every request.

Browser uses **Set-Cookie** header to ask client to store the cookie. The client uses **Cookie** header to send back the cookie to the server so the server knows which client it is talking to.

Cookies are incompatible with *REST* style/architecture since *REST* is stateless. According to *REST* style, cookies maintain site-wide state while *REST* styles maintains application state. In *REST*, cookie functionality can be achieved using anonymous authentication and client-side state. *REST* also defines an alternative to cookies when implementing shopping carts. According to *REST*:

Likewise, the use of cookies to identify a user-specific "shopping basket" within a server-side database could be more efficiently implemented by defining the semantics of shopping items within the hypermedia data formats, allowing the user agent to select and store those items within their own client-side shopping basket, complete with a URI to be used for check-out when the client is ready to purchase.

Cookies have certain rules and attributes:

1. Name/value pair can't contain spaces or *;* =. Usually only ASCII characters. The *;* is used as a delimiter.
2. The *Secure* attribute means this cookie is only used in encrypted communications.
3. The *HttpOnly* attribute means this cookie can only be used by http/https requests and not by JavaScript, etc. This prevents cross site scripting.

Other notes:

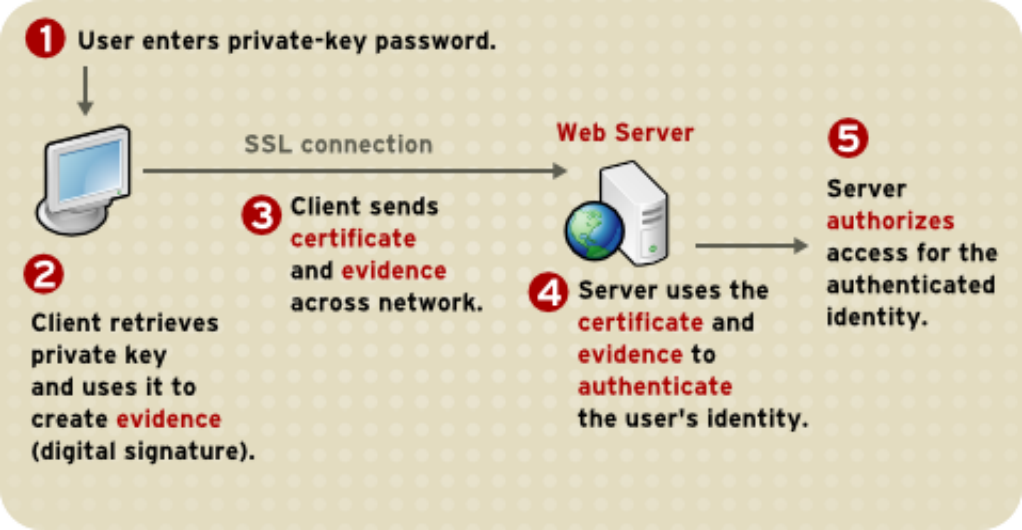
1. Not good practice to store username/password in cookies, even if it is hashed/salted, etc. Can be stolen and eventually cracked.
2. Cookie based authentication basically involves using the cookie the server sent to the client back to the server for every request.

Certificate Based

Idea is to separate those who verify password (the server will have a copy or a hash of the password) and those who define the user identity. Thus, certificate authority (CA) issues a private certificate to a user, and guarantees that it can communicate using this key with the public key issued to the other business party.

Note that the downside becomes apparent when large number of clients or users need to authenticate to the server. Thus, CA needs to issue certificate for each user. These certificates needs to be verified and if one user is compromised the certificate of that user can be used to authenticate to the server unless the certificate is revoked.

For the reasons stated above, client authentication is rarely used with TLS. A common technique is to use TLS to authenticate the server to the client and to establish a private channel, and for the client to authenticate to the server using some other means - for example, a username and password using HTTP basic or digest authentication.



The above image depicts certificate-based authentication. The client asks the user to enter a password which unlocks the database holding the private key. The client then uses this private key to sign a random data and sends a certificate to the server. Thus, the password is never sent.

The Red Hat Portal discusses this in great detail.

HTTPS

1. It's HTTP over TLS or HTTP over SSL (*https://* instead of *http://*). Thus, uses an added encryption layer (above Transport Layer, before Application Layer) of SSL/TLS to protect traffic.
2. Main motivation is to prevent wiretapping/man in the middle attacks.
3. HTTPS provides authentication of the website and associated web server that one is communicating with.
4. Provides a bidirectional encryption of communications between a client and server.
5. URL, query parameters, headers, are protected. However, it only protects HTTP layer. Thus, can infer host addresses, IP address, and sometimes port number of the webserver (since TCP layer is not encrypted). Can get data transferred and duration of TCP connection but not content.

Trusting a Web Site

1. Web browsers know how to trust HTTPS websites based on certificate authorities that come pre-installed in their software.

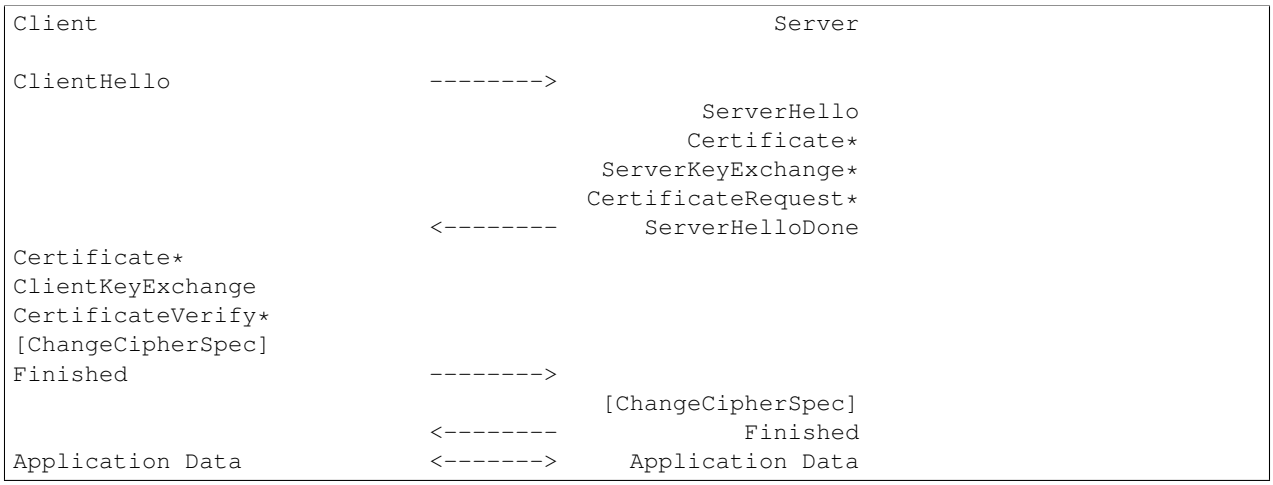
2. Certificate authorities, such as Comodo and GlobalSign, are in this way being trusted by web browser creators to provide valid certificates.
3. Must use a browser that correctly implements HTTPS with correct pre-installed certificates.
4. User trusts CA to "vouch" for website they issued certificate to.
5. The user trusts that the protocol's encryption layer (TLS/SSL) is sufficiently secure against eavesdroppers.

TLS/SSL

1. Uses assymetric cryptography:
   - Basically known as public key cryptography.
   - Requires two keys. A private/secret and a public key.
   - Public key is used to encrypt plain text or verify a digital signature. Private key is used to decrypt the plain text or create a digital signature.
2. The assymetric key is used for authentication and encrypting the channel. Then, a symmetric session key is exchanged.
3. The session key is used to encrypt data flowing between the parties. Important property is *forward secrecy*. This means that the short-term session key cannot be derived from long term assymetric secret key.
4. In OSI model equivalences, TLS/SSL is initialized at layer 5 (session layer) and works at layer 6 (the presentation layer). The session layer has a handshake using an asymmetric cipher in order to establish cipher settings and a shared key for that session; then the presentation layer encrypts the rest of the communication using a symmetric cipher and that session key.
5. TLS is the new name for SSL.
6. SSL got to version 3.0 and TLS is "SSL 3.1".
7. Current version of TLS is 1.2.
8. SSL (secure socket layer) often refers to the old protocol variant which starts with the handshake right away and therefore requires another port for the encrypted protocol such as 443 instead of 80.
9. TLS (transport layer security) often refers to the new variant which allows to start with an unencrypted traditional protocol and then issuing a command (usually STARTTLS) to initialize the handshake.
10. Differences between SSL and TLS in the protocol level:
    - In the ClientHello message (first message sent by the client, to initiate the handshake), the version is {3,0} for SSLv3, {3,1} for TLSv1.0 and {3,2} for TLSv1.1.
    - The ClientKeyExchange differs.
    - The MAC/HMAC differs (TLS uses HMAC whereas SSL uses an earlier version of HMAC).
    - The key derivation differs.
    - The client can send application data can be sent straight after ending the SSL/TLS Finished message in SSLv3. In TLSv1, it must wait for the server's Finished message.
    - The list of cipher suites differ (and some of them have been renamed from SSL_* to TLS_*, keeping the same id number).
    - There are also differences regarding the new re-negotiation extension.
11. Use port 443 by default.

12. TLS, which uses long-term public and secret keys to exchange a short term session key to encrypt the data flow between client and server.
13. X.509 certificates are used to guarantee one is talking to the partner with whom one wants to talk.
14. Need to ensure scripts are loaded over HTTPS as well and not HTTP.
15. In case of compromised secret (private) key, certificate can be revoked.
16. Use Perfect Forward Secrecy (PFS) so that short term session key can't be derived from long term assymetric secret key.

Handshake



1. Exchange hello messages to agree on algorithms, exchange random values, check for resume.
    - The ClientHello and ServerHello establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.
2. Exchange necessary crypto parameters for client/server to agree on premaster secret.
3. Exchange certs and crypto information to allow client/server to authenticate.
4. Generate a master secret from the premaster secret and exchanged random values.
5. Provide security parameters to the record layer.
6. Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

Server Setup

1. To prepare a web server to accept HTTPS connections, the administrator must create a public key certificate for the web server.
2. This certificate must be signed by a trusted certificate authority for the web browser to accept it without warning.
3. Web browsers are generally distributed with a list of signing certificates of major certificate authorities so that they can verify certificates signed by them.

Other Uses

1. The system can also be used for client authentication in order to limit access to a web server to authorized users. To do this, the site administrator typically creates a certificate for each user, a certificate that is loaded into his/her browser.

nginx *engineX*

Permissions

Make sure the permissions of the files in the directory are accessible to the *other* group. Or change the permissions to the user that *nginx* runs as (for debian it's *www-data*).

Setting up Basic Auth

1. Install **apache2-utils** to get **htpasswd**
2. Create an **.htpasswd** file in the web root. Make sure the permissions are *644*. Note that the password generated by *htpasswd* is an apache modified version of MD5.

```
sudo htpasswd -c /usr/share/nginx/html/.htpasswd amit
```

3. Update */etc/nginx/sites-available/default* in the location / and reload *nginx*:

```
# Basic auth
auth_basic "Restricted";
auth_basic_user_file /etc/nginx/.htpasswd;
```

Setting up Digest Auth

1. **apache2-utils** includes **htdigest** (similar to *htpasswd*) to generate digest key.
2. Create an **.htdigest** file in the web root. Make sure the permissions are *644*. Note that the *realm* here is *"Access Restricted"*.

```
sudo htdigest -c /usr/share/nginx/html/.htdigest "Access Restricted" amit
```

3. Need to build with *nginx-http-auth-digest* module from https://github.com/rains31/nginx-http-auth-digest. In order to do this, download *nginx* debian sources, copy *nginx-http-auth-digest* to *debian/modules*, and finally edit *debian/rules* to build *nginx-http-auth-digest* (look at *–add-module* config option).
4. Update */etc/nginx/sites-available/default* in the location / and reload *nginx*:

```
# Digest auth
auth_digest "Access Restricted";    # Realm
auth_digest_user_file /usr/share/nginx/html/.htdigest;
```

Others

HTTPie - Command Line HTTP Client

Very useful and feature rich command line http client written in Python (http://github.com/jakubroztocil/httpie).

Useful for debugging HTTP requests. For example:

```
$ http get http://localhost
HTTP/1.1 200 OK
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html
Date: Mon, 01 Sep 2014 18:31:03 GMT
Last-Modified: Tue, 05 Aug 2014 11:18:35 GMT
Server: nginx/1.6.1
Transfer-Encoding: chunked

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

DNS

Contents

Overview

Notes taken from:

1. A DNS Primer
2. Linux Network Administrator's Guide, 2nd Edition
3. RFC 1035 - DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION

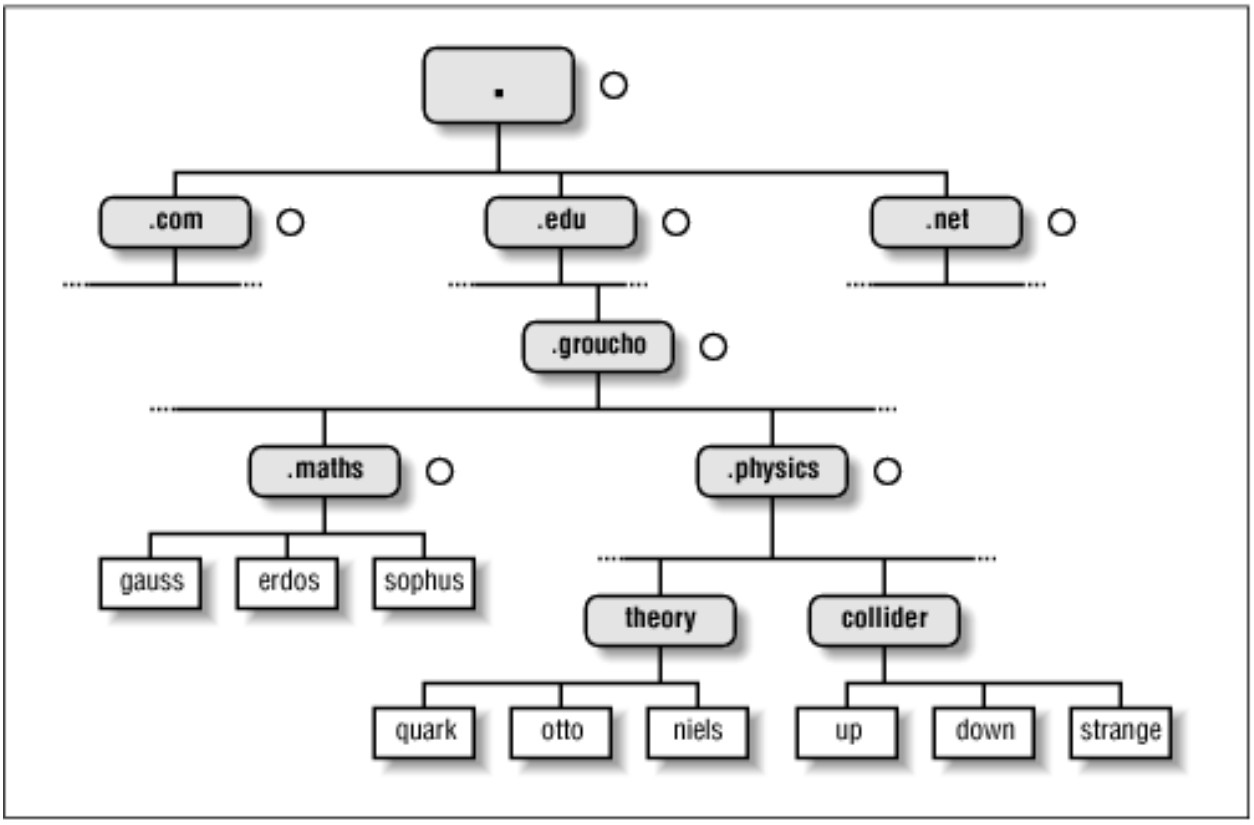
DNS is the Domain Name System used for obtaining IP Addresses from FQDN (Fully Qualified Domain Names). An FQDN is an absolute name and provides the exact location in the tree hierarchy of the domain name system. It uniquely identifies the host worldwide.

Also note that DNS was designed to be decentralized, thus no central authority manages all the hosts.

DNS organizes hostnames in a domain hierarchy. A domain is collection of sites that are related in some sense (all machines in a campus, organization, etc).

But it is more than that, given a name, it finds resources associated with that name. This is accomplished through a distributed database system where requests for names are handed off to various tiers of servers which are delinieated by the dot (.). It resolves from right to left:

1. The root domain (dot) encompases all domains. Sometimes to indicate a domain is fully qualified, rather than relative, it is written with a trailing dot, which signifies the name's last component is the root domain.
2. The top level domain (TLD) (List of top level domains: http://www.iana.org/domains/root/db)
3. The second level domain
4. The subdomain
5. The host/resource name



Note that organizing the namespace in a hierarchy of domain names nicely solves the problem of name uniqueness; with DNS, a hostname has to be unique only within its domain to give it a name different from all other hosts worldwide.

To this end, the namespace is split up into zones, each rooted at a domain. Note the subtle difference between a zone and a domain: the domain groucho.edu encompasses all hosts at Groucho Marx University, while the zone groucho.edu includes only the hosts that are managed by the Computing Center directly; those at the Mathematics department, for example. The hosts at the Physics department belong to a different zone, namely physics.groucho.edu. In the above figure, the start of a zone is marked by a small circle to the right of the domain name.

When clients make requests, they make recursive queries (rather thand iterative queries) which lets the DNS server to do the work of getting the answer iteratively and thus returning only the final answer to the client.

Typical Name Resolution

For each zone there are at least two, or at most a few, name servers that hold all authoritative information on hosts in that zone. Name servers that hold all information on hosts within a zone are called authoritative for this zone, and sometimes are referred to as master name servers. Any query for a host within this zone will end up at one of these master name servers.

Also, cache is very important for DNS Servers. If there were no caches, it would be very inefficient. The data in the cache does not stay forever though, it is configured by the admin using TTL (time to live) configuration for the DNS Server.

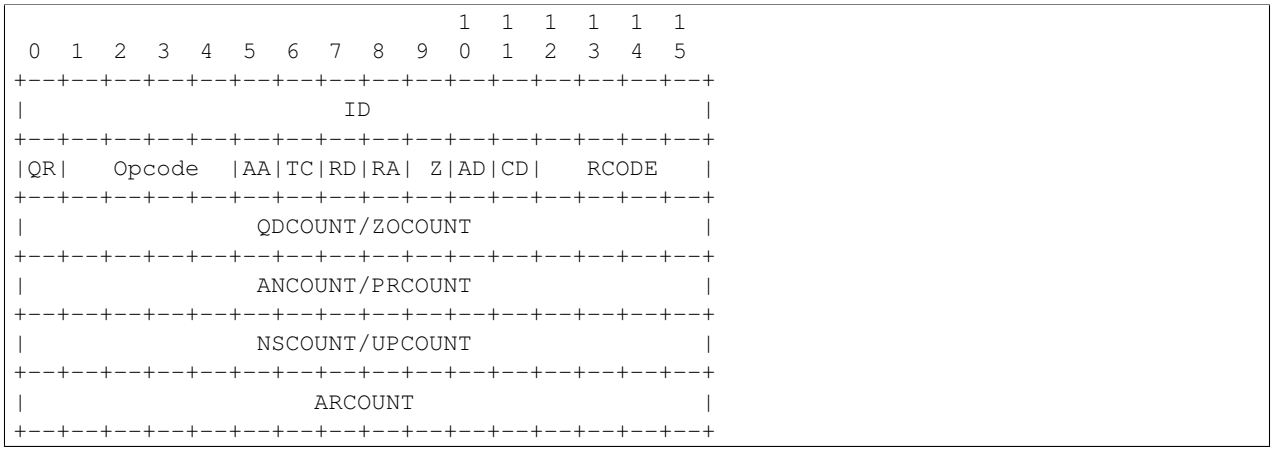
When a client makes a request, the following usually happens:

1. The DNS server is configured with an initial cached (hints) of known addresses of root name servers. This is updated periodically in an authoritative way.

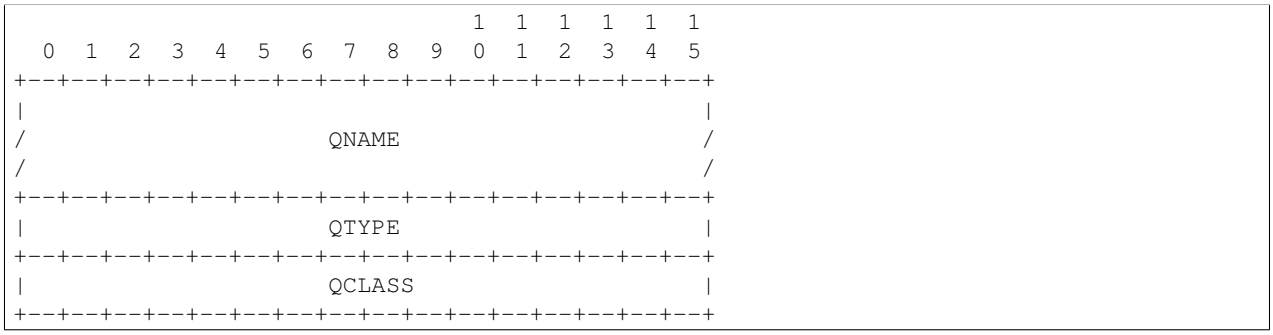
2. Server receives requests from client and services it using its cache first (usually an answer is cached from previous lookups). If not, it performs the following steps for client.
3. Query is made to one of root servers to find server that is authoritative for top-level domain being requested.
4. Answer is received that points to nameserver of the top-level domain resource.
5. Server *walks the tree* from right to left, sending requests from nameserver to nameserver, until final step which returns IP of host in question.
6. IP Address of requested resource is given to client.

Protocol

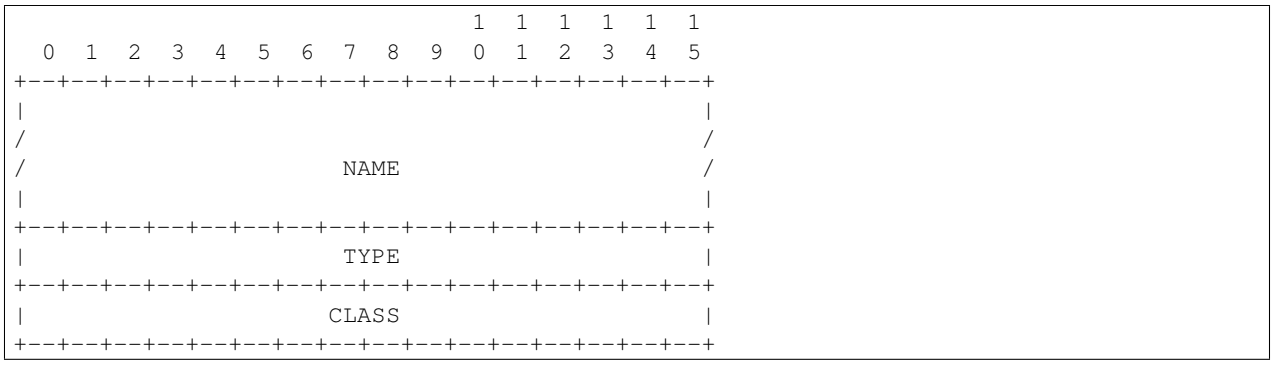
Header

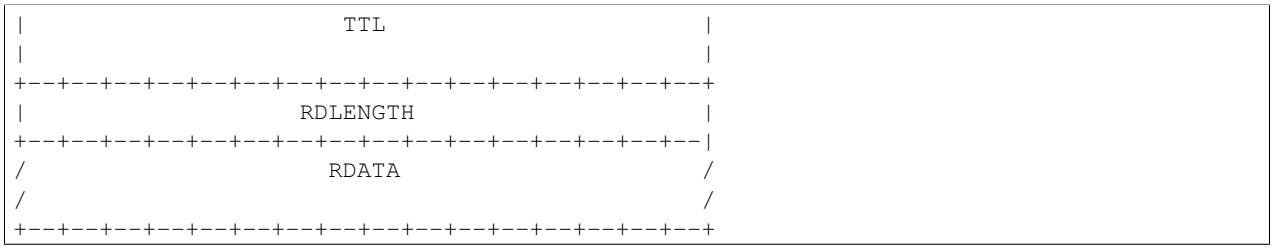


Question Section



Resource Record Format





1. DNS protocol uses port 53 for TCP and UDP.
2. DNS protocol is quite light (12 bytes header) and uses UDP so it is fast and much less overhead.
3. Zone Transfer and other heavy operations use TCP.
4. Fields in header:
    - *Identifier*: 16-bit field containing ID so requests and responses and can be matched.
    - *QR Flag*: 1-bit field indicating packet is query or response.
    - *OP*: Specifies type of message. 0 - standard query, 1 - inverse query (obsolete), 2 - server status, 3 - reserve and unused, 4 - notification, 5 - update (Dynamic DNS).
    - *AA* - Single bit indicating authoritative answer from server who authoritative for that domain.
    - *TC*: Single bit for truncation. If set, usually means sent via UDP but was longer than 512 bytes.
    - *RD*: Single bit indicating recursion desired.
    - *RA*: Single bit reply by server indicating recursion is available.
    - *Z*: Three bits reserved and set to 0.
    - *RCode*: 4-bit field set to 0s for queries but set for responses.
        - 1 - Format error
        - 2 - Server failure
        - 3 - Name error
        - 4 - Not implemented
        - 5 - Refused
        - 6 - Name exists but shouldn't
        - 7 - Resource records exists but shouldn't
        - 8 - Resource record that should exist but doesn't
        - 9 - Response is not authoritative
        - 10 - Name is response is not within zone specified.
    - *QCount*: How many questions in question section
    - *ANCount*: How many answers in answer section
    - *NSCount*: How many resource records in authority section
    - *ARCount*: How many resource records in additional section

DNS Database

1. DNS database does not only deal with IP Addresses of hosts but contains different types of entries.
2. Single piece of info from the DNS database is called a *RR (Resource Record)*.
3. Each record has a type associated with it describing the sort of data it represents, and a class specifying the type of network it applies to. The latter accommodates the needs of different addressing schemes, like IP addresses (the IN class), Hesiod addresses (used by MIT's Kerberos system), and a few more. The prototypical resource record type is the A record, which associates a fully qualified domain name with an IP address.
4. A host may be known by more than one name. For example you might have a server that provides both FTP and World Wide Web servers, which you give two names: *ftp.machine.org* and *www.machine.org*. However, one of these names must be identified as the official or canonical hostname, while the others are simply aliases referring to the official hostname. The difference is that the canonical hostname is the one with an associated A record, while the others only have a record of type CNAME that points to the canonical hostname.

Example named.hosts file for the Physics Department

```
; Authoritative Information on physics.groucho.edu.
@  IN  SOA niels.physics.groucho.edu. janet.niels.physics.groucho.edu. {
                  1999090200       ; serial no
                  360000           ; refresh
                  3600             ; retry
                  3600000          ; expire
                  3600             ; default ttl
                }
;
; Name servers
              IN    NS       niels
              IN    NS       gauss.maths.groucho.edu.
gauss.maths.groucho.edu. IN A 149.76.4.23
;
; Theoretical Physics (subnet 12)
niels         IN    A        149.76.12.1
              IN    A        149.76.1.12
name server   IN    CNAME    niels
otto          IN    A        149.76.12.2
quark         IN    A        149.76.12.4
down          IN    A        149.76.12.5
strange       IN    A        149.76.12.6
...
; Collider Lab. (subnet 14)
boson         IN    A        149.76.14.1
muon          IN    A        149.76.14.7
bogon         IN    A        149.76.14.12
...
```

1. The *SOA* record signals the Start of Authority, which holds general information and configuration on the zone the server is authoritative for.
2. *CNAME* always points to another name. This name then has an assiociated *A* record.
3. Note that all names in the sample file that do not end with a dot should be interpreted relative to the *physics.groucho.edu* (e.g. *boson*, *muon*) domain. The special name (@) used in the SOA record refers to the domain name by itself.
4. The name servers for the groucho.edu domain somehow have to know about the physics zone so that they can point queries to their name servers. This is usually achieved by a pair of records: the NS record that gives the

server's FQDN, and an A record that associates an address with that name. Since these records are what holds the namespace together, they are frequently called glue records.

Reverse Lookups

1. Sometimes you need to look up the *canonical* name from an IP address. This is called *reverse mapping*.
2. A special domain *in-addr.arpa* has been created that contains the IP addresses of all hosts in a reversed dotted quad notation. For instance, an IP address of *149.76.12.4* corresponds to the name *4.12.76.149.in-addr.arpa*. The resource-record type linking these names to their canonical hostnames is *PTR*.
3. Note that if the address is a subnet that ends in *0* the *0* is ommitted in the reverse dotted quad notation. For example, subnet *149.76.12.0* corresponds to name *12.76.149.in-addr.arpa*.

```
; the 12.76.149.in-addr.arpa domain.
@  IN  SOA  niels.physics.groucho.edu. janet.niels.physics.groucho.edu. {
                      1999090200 360000 3600 3600000 3600
            }
2         IN      PTR       otto.physics.groucho.edu.
4         IN      PTR       quark.physics.groucho.edu.
5         IN      PTR       down.physics.groucho.edu.
6         IN      PTR       strange.physics.groucho.edu.
```

1. in-addr.arpa system zones can only be created as supersets of IP networks. An even more severe restriction is that these networks' netmasks have to be on byte boundaries. All subnets at Groucho Marx University have a netmask of 255.255.255.0, hence an in-addr.arpa zone could be created for each subnet. However, if the netmask were 255.255.255.128 instead, creating zones for the subnet 149.76.12.128 would be impossible, because there's no way to tell DNS that the 12.76.149.in-addr.arpa domain has been split into two zones of authority, with hostnames ranging from 1 through 127, and 128 through 255, respectively.

DNS Caches vs. DNS Servers vs. DNS Resolvers

1. DNS Cache is a list of names and IPs you resolved recently. The cache can be located in the OS level (not for Linux). Cache can be at browser level, router level, ISP level.
2. A DNS server can act as a cache if it is not authoritative for any domain. Thus, performs queries for clients and caches resolved names.
3. A DNS server can be authoritative for that domain and holds authoritave answers for certain resources.
4. DNS Resolvers are just clients.
    - When the client requests for recursive queries, it asks the server to do all the work for it and just waits for the final answer.
    - Iterative queries gets a response from server on where to look next. For example, if the client asks for chat.google.com, it tells the client to check with the .com servers and considers its work done.

Authoritative vs Non-authoritative Responses

1. Authoritative responses come directly from a nameserver that has authority over the record in question.
2. Non-authoritave come from a second-hand server or more likely a cache.

Zone Tranfers

1. Uses TCP instead of UDP and during the operation, the client sends a query type of IXFR instead of AXFR.
2. Slave DNS servers pull records from master DNS servers.
3. Can use *dig* to perform Zone Transfer.
4. If you have control of the zone, you can set it up to get transfers that are protected with a TSIG key. This is a shared secret the the client can send to the server to authorize the transfer.

Anycast DNS

1. Allows for same IP to be served from multiple locations.
2. Network decides based on distance, latency, and network conditions which location to route to.
3. Like a CDN for your DNS.
4. When you deploy identical servers at multiple nodes, on multiple networks, in widely diverse geographical locations, all using Anycast, you're effectively adding global load-balancing functionality to your DNS service. Importantly, the load-balancing logic is completely invisible to the DNS servers; it's moved down the stack from the application to the network layer. Because each node advertises the same IP address, user traffic is shared between servers globally, handled transparently by the network itself using standard BGP routing.
5. An example of this would be to list your DNS servers as 1.2.3.4 and 1.2.3.5. Your routers would announce a route for 1.2.3/24 out of multiple datacenters. If you're in Japan and have a datacenter there, chances are you'd end up there. If you're in the US, you'd be sent to your US datacenter. Again, it's based on BGP routing and not actual geographic routing, but that's usually how things break down.

DNS Security

1. Main security issue is typing correct URL and pointed to IP of malicious server.
2. Easy to spoof because query and responses are UDP based.
3. DNSSEC is security oriented extensions for DNS. Main purpose is to ensure response comes from authorized origin.
4. Works by signing responses using public-key cryptography and uses new resource records.
    - *RRSIG*: DNSSEC signature for a record set. The DNS clients verify the signature with a public key stored in *DNSKEY* record.
    - *DNSKEY*: Contains the public key.
    - *DS*: Holds name of delegated zone.
    - *NSEC*: Contains link to next record name in zone. Used for validation.
    - *NSEC3*: Similar to NSEC but hashed.
    - *NSEC3PARAM*: Authoritative servers uses this which *NSEC3* records to use in responses.

Examples

Query All Records using dig

```
$ dig +nocmd com.google any +multiline +noall +answer
```

Or

```
$ dig com.google any
```

Firewall

Contents

Notes from: Wikipedia - Firewall

1. Firewall controls incoming and outgoing network based on applied rules.
2. Basically establishes a barrier between internal network and outside network.
3. Proxies can be firewalls by blocking certain connections from certain hosts or addresses.
4. Network Address Translation (NAT) has become an important part of firewalls. It hides addresses of hosts behind the firewall.

First Generation: Packet Filters

1. First paper on firewall technology published was in 1988 by DEC. Talked about a *packet filter* firewall.
2. Act by inspecting packets. If it matches set of filtering rules, it silently drops packet or reject it with error message back to source.
3. The mechanism does not look at whether the packet is part of a connection stream, etc. Thus, it doesn't really maintain a state. It rejects based on looking at combination of source, destination address, protocol, port number, etc.
4. Pretty much works on the first three layers of OSI Model. It does peek into transport layer sometimes for source/destination port numbers.
5. Term originated in context of BSD operating systems.
6. Examples are *iptables* for Linux and *PF* for BSD.

Second Generation: Stateful Filters

1. 1989-1990 from AT&T Bell Labs developed second gen firewall calling it circuit level gateway.
2. Operates up to Layer 4 (Transport Layer). Achieved by retaining enough packets in the buffer until enough information is availabe to make a judgement about its state.
3. Thus, it records all connections passing through it and determines if a packet is a part of current connection or new connection. Known as *stateful packet inspection*.
4. Can DoS by flooding firewall with thousands of fake connection packets.

Third Generation: application layer

1. Key benefit is that it can understand certain Application Layer protocols (FTP, HTTP, DNS).
2. Useful to detect if unwanted protocol is trying to use standard port from known applications (e.g. HTTP) to bypass firewall.
3. Can inspect if packet contains virus signatures.
4. Hooks into socket calls automatically.
5. Disadvantages are that it is quite slow and that rules can get complicated. It also can't possibly support of applications at application layer.

IP

Contents

Path MTU Discovery (PMUTD)

1. Determines the MTU (Maximum Transmission Unit) size on the network path between two IP hosts. (RFC 1191 - Path MTU Discovery).
2. The goal is to avoid IP fragmentation.
3. In IPv6, this function has been explicitly delegated to the end points of a communications session. (RFC 1981 - Path MTU Discovery for IP version 6).
4. For IPv4 packets, Path MTU Discovery works by setting the Don't Fragment (DF) option bit in the IP headers of outgoing packets. Then, any device along the path whose MTU is smaller than the packet will drop it, and send back an Internet Control Message Protocol (ICMP) Fragmentation Needed (Type 3, Code 4) message containing its MTU, allowing the source host to reduce its Path MTU appropriately. The process is repeated until the MTU is small enough to traverse the entire path without fragmentation.
5. IPv6 routers do not support fragmentation or the Don't Fragment option. For IPv6, Path MTU Discovery works by initially assuming the path MTU is the same as the MTU on the link layer interface through which the traffic is being sent.
   - Then, similar to IPv4, any device along the path whose MTU is smaller than the packet will drop the packet and send back an ICMPv6 Packet Too Big (Type 2) message containing its MTU, allowing the source host to reduce its Path MTU appropriately. The process is repeated until the MTU is small enough to traverse the entire path without fragmentation.
6. Many network security devices block all ICMP messages for perceived security benefits,[6] including the errors that are necessary for the proper operation of PMTUD. This can result in connections that complete the TCP three-way handshake correctly, but then hang when data is transferred. This state is referred to as a black hole connection.
7. A robust method for PMTUD that relies on TCP or another protocol to probe the path with progressively larger packets has been standardized in RFC 4821.

tcpdump

Contents

Network packet capture tool in the CLI. *tcpdump* is extremely useful for quick debugging of network issues. Use the *pcap-filter* manpage for the filter syntax reference.

Flags

tcpdump Flags:

TCP Flag	tcpdump Flag	Meaning
SYN	S	Syn packet, a session establishment request.
ACK	A	Ack packet, acknowledge sender's data.
FIN	F	Finish flag, indication of termination.
RESET	R	Reset, indication of immediate abort of conn.
PUSH	P	Push, immediate push of data from sender.
URGENT	U	Urgent, takes precedence over other data.
NONE	A dot .	Placeholder, usually used for ACK.

Examples

Capturing ARP Traffic

When using tcpdump to capture ARP, make sure to dump the hex output (*-X*) and also decode ethernet header using (*-e*). **Note: Use *-XX* to also show ethernet header dump**.

```
$ sudo tcpdump -nnvvv -e -X arp
  tcpdump: listening on wlan0, link-type EN10MB (Ethernet), capture size 262144 bytes
  20:01:28.452956 48:5a:b6:51:57:dd > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 60: Ethernet
      0x0000:  0001 0800 0604 0001 485a b651 57dd c0a8  ........HZ.QW...
      0x0010:  0117 0000 0000 0000 c0a8 0101 0000 0000  ................
      0x0020:  0000 0000 0000 0000 0000 0000 0000       ..............
  20:01:28.454472 bc:ee:7b:58:17:b8 > 48:5a:b6:51:57:dd, ethertype ARP (0x0806), length 42: Ethernet
      0x0000:  0001 0800 0604 0002 bcee 7b58 17b8 c0a8  ..........{X....
      0x0010:  0101 485a b651 57dd c0a8 0117            ..HZ.QW.....
```

Capturing Traffic on Localhost

During development, there is usually a local webserver setup in *http://localhost*. Custom apps/scripts are tested against this local webserver to make them functionally correct. Thus, it is important to be able to analyze traffic to and from the local webserver. Using the local webserver for traffic analysis helps as there are no external traffic that will confuse the analysis.

To capture *localhost* traffic:

```
sudo tcpdump -A -v --number -i lo tcp port http
```

- *-A* is used to decode protocol in *ASCII*.
- *-v* is used for verbose mode. This allows us to see *tcp* communication details (flags, sequence numbers, etc).
- *–number* denomitate the packets
- *-i lo* use local loopback interface
- *tcp port http* the filter specifying protocol and port to use for capture.

Use *-l* for line buffering to see data while capturing it to a file.

```
sudo tcpdump -l -A -v --number -i lo tcp port http | tee /tmp/capture
```

Capturing GMail Traffic

GMail goes over IMAP but not the standard IMAP port (143), it uses 993:

```
sudo tcpdump -vvv -X --number -i wlan0 host 192.168.1.24 and tcp port 993
```

Use `-vvv` (three is max) to decode max level of the packets. Then use *-X* to decode in Hex and ASCII.

Dropped Packets by the Kernel

tcpdump uses a little buffer in the kernel to store captured packets. If too many new packets arrive before the user process tcpdump can decode them, the kernel drops them to make room for freshly arriving packets.

Use *-B* to increase the buffer. This is in units of KiB (1024 bytes).

Capturing TCP SYN Packets

To capture SYN packets only:

```
$ sudo tcpdump -nnvvv host 192.168.1.116 and "tcp[tcpflags] & tcp-syn != 0"
```

To capture TCP keepalive packets 1-byte or 0-byte ACKs. Note that a keepalive probe is a packet with no data and ACK flag turned on:

```
$ sudo tcpdump -vv "tcp[tcpflags] == tcp-ack and less 1"
```

Capture Outgoing SSH Traffic

```
$ sudo tcpdump -nn src 192.168.1.116 and tcp port 22
```

Get Time Delta Between Request/Response

Pass the *-ttt* flag to get the time delta between current line and previous line.

```
$ sudo tcpdump -nS -ttt port http and host snapshot.debian.org

tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes

00:00:00.000000 IP 192.168.1.170.34233 > 193.62.202.30.80: Flags [S], seq 1140376233, win 29200, opti
00:00:00.228373 IP 193.62.202.30.80 > 192.168.1.170.34233: Flags [S.], seq 1460190713, ack 114037623
00:00:00.000040 IP 192.168.1.170.34233 > 193.62.202.30.80: Flags [.], ack 1460190714, win 229, optio
00:00:00.000119 IP 192.168.1.170.34233 > 193.62.202.30.80: Flags [P.], seq 1140376234:1140376399, ack
00:00:00.222658 IP 193.62.202.30.80 > 192.168.1.170.34233: Flags [.], ack 1140376399, win 54, options
00:00:00.001001 IP 193.62.202.30.80 > 192.168.1.170.34233: Flags [P.], seq 1460190714:1460191405, ack
00:00:00.000032 IP 192.168.1.170.34233 > 193.62.202.30.80: Flags [.], ack 1460191405, win 239, optio
00:00:00.008210 IP 192.168.1.170.34233 > 193.62.202.30.80: Flags [F.], seq 1140376399, ack 146019140
00:00:00.183523 IP 193.62.202.30.80 > 192.168.1.170.34233: Flags [F.], seq 1460191405, ack 114037640
00:00:00.000060 IP 192.168.1.170.34233 > 193.62.202.30.80: Flags [.], ack 1460191406, win 239, optio
```

Capturing WiFi Packets

First, the *wlan0* interface needs to be set to monitor mode:

```
$ sudo ifconfig wlan0 down
$ sudo iwconfig wlan0 mode Monitor
$ sudo ifconfig wlan0 up
```

Then, run *tcpdump* with the following flags:

```
$ sudo tcpdump -I -i wlan0 -w thermostat.pcap -e -s 0 ether host 00:d0:2d:xx:xx:xx
```

This captures all packets originating from the Honeywell thermostat for example.

Cheatsheet

Contents

Finding Duplicate IP Addresses on the Network

1. Install **arp-scan**.
2. Run the following command:

```
$ sudo arp-scan -I eth0 -l | grep 192.168.1.42
$ OR
$ sudo arp-scan -I eth0 -l | grep DUP
192.168.1.92    00:c0:b7:55:f6:1f       AMERICAN POWER CONVERSION CORP (DUP: 2)
```

```
192.168.1.152   00:1b:00:0f:55:0e       Neopost Technologies (DUP: 2)
192.168.1.158   00:14:38:48:75:b7       Hewlett Packard (DUP: 2)
```

Capturing cupsd Traffic

First to get the list of open ports used by *cupsd*, use *netstat*:

```
$ sudo netstat -lntup | grep cupsd

tcp        0      0 0.0.0.0:631         0.0.0.0:*  LISTEN      512/cupsd
tcp6       0      0 :::631              :::*       LISTEN      512/cupsd
udp        0      0 0.0.0.0:631         0.0.0.0:*              512/cupsd
```

tcpdump can't be used so need to use *strace* here:

```
$ sudo strace -p 512 -f -e trace=network -s 10000 -o capture.out &
$ sudo lpinfo -v
```

This will capture the HTTP messages (IPP) received by cupsd.

Troubleshooting

Contents

Identifying and Solving Performance Issues

Steps should be followed in order:

1. *Understand the problem*
   - Half the problem is solved when problem is understood clearly.
2. *Monitor and collect data*
   - Monitor the system and collect as much data as possible.
3. *Eliminate and narrow down issues*
   - Come up with a list of potential issues and possibly narrow them down eliminating any non issues.
4. *Make one change at a time*
   - Make one change and re-test. Don't make multiple changes at one time.

Common Server Problems

1. "Transfer-Encoding: chunked" isn't needed for progressive rendering. However, it is needed when the total content length is unknown before the first bytes are sent. Usually, it is used automatically by Web Server when you start sending data without knowing the length.

2. A server is used as a proxy in order to perform cross domain requests. If the server returns *502* error it means: *The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.*

3. Sometimes a server is not accessible why is that? Answer in terms of DNS.

   The DNS resolver's cache is controlled by the time-to-live (TTL) value that you set for your records and is specified in seconds. For example, if you set a TTL value of 86400 (the number of seconds in 24 hours), the DNS resolvers are instructed to cache the records for 24 hours. Some DNS resolvers ignore the TTL value or use their own values that can delay the full propagation of records.

   If you are planning for a change to services that requires a narrow window, you might want to change the TTL in advance of your change to a shorter TTL value. This change can help reduce the caching window and ensure a quicker change to your new record settings. After the change, you can change the value back to its previous TTL value to reduce load on the DNS resolvers.

Common AJAX Problems

1. Although, almost all browsers support JavaScript today, there could be some users accessing it with JavaScript disabled.
   - Should design to degrade gracefully when it detects JavaScript is disabled.
   - Also build the website to work without JavaScript. Then, JavaScript should be used as an enhancement. Most users will see the page with JavaScript anyways.
2. Sometimes user doesn't know a request has completed. Thus, display a message or status indicating request has been completed.
3. AJAX requests can't access third-party web services.
   - The *XMLHttpRequest* object, which is at the root of all Ajax requests, is restricted to making requests on the same domain as the page making the request.
   - Solution is to use your server as the proxy to access API of third party web service.
   - Can also use jQuery to perform cross-domain requests (*$.ajax()* API).
   - Can use <*script*> tags that load JavaScript files from other domains. *JSONP (JSON Padding)* uses this technique to dynamically create <*script*> tag with necessary URL.
4. Using back button on sites with AJAX can sometimes revert the page back to its initial state.
   - Solution to this is to use internal page anchors. Basically, save the current URL and use that.

Common App Engine Problems

1. If you have timeouts in your application, you maybe updating a single entity group in your datastore too rapidly (about 5 times/sec). Datastore will be in contention. Design issue of BigTable.
   - Can be a problem if you have a entity that is a counter that you want to update faster than 5 times/sec.
   - Way to reduce problem is taking advantage of extremely cheap and fast reads from datastore. Thus, use *sharding* to split counters into *N* counters.
     - When you want to increment the counter, pick a shard at random and increment it.
     - When you want to read the total count, read all shards and sum up all counters.
     - The more shards, the higher throughput on increments to counters.
   - Another way is to make updates to memcache and periodically flushing to datastore.

- Timeouts can happen if your data is in a tablet that is currently being moved around between servers for load balancing when you are trying to access it.
- Timeouts can happen if you are writing large data and thus tablets are being split while you are writing. Use exponential backoff strategy before retrying since it takes sometimes couple hunder millisecs to a sec or two for tablets to be available.

2. To handle errors in datastore gracefully, make transactions *idempotent*. Thus, if you repeat the transaction, the end result will be the same.
3. If daily budget is exceeded in app engine, application could serve errors.
4. When upgrading to a new application, don't move all your users right away. Use traffic splitting to gradually move new requests to new version of your application. Cached objects may no longer be cached thus an increase in read latency.
   - IP Address splitting works by hashing IP Address to value of 0-999 and splitting based on that.
   - Cookies give finer split control and more detailed statistics. Uses *GOOGAPPUID* cookie and range of 0-999.
   - Some issues with traffic splitting is cached documents. Use *Cache-Control* and *Expires* header to tell proxies that resource is dynamic.
5. To avoid thundering herd problem (where retries get compounded because first retry fails and all requests keep trying), use backoff on retry.

CHAPTER 2

Forensics

Forensics

Cheatsheet

Contents

Mounting E01 Images

1. Install **ewf-tools** which contains **ewfmount**.
2. Mount the E01 image. This mounts it as a raw file.

```
# ewfmount /srv/public/E01Capture/E01Capture.E01 /mnt/ewf
# or for multiple E01 files
# ewfmount /srv/public/E01Capture/E01Capture.E* /mnt/ewf
```

3. Then use **mmls** from the **sleuthkit** package to analyze the raw image and find out where the partition offsets are. Note, in this case partition offset is (*512*2048=1048756*).

```
# mmls /mnt/ewf/ewf1
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000000   0000002047   0000002048   Unallocated
```

```
02:  00:00   0000002048   0007890943   0007888896   NTFS (0x07)
03:  -----   0007890944   0007892991   0000002048   Unallocated
```

4. Mount the filesystem using regular *mount*:

```
# mount -t ext4 -o ro,loop,offset=1048576 /mnt/ewf/ewf1 /mnt/usb
# or for ntfs
# mount -t ntfs-3g -o ro,nodev,noexec,show_sys_files,loop,offset=1048576 /mnt/ewf/ewf1 /mnt/usb
```

Mounting ISO9660

1. Install *fuseiso9660*. Somehow, the following does not work:

```
# mount -t iso9660 -o loop test.iso /mnt/loop
```

2. Mount the disk image:

```
# fuseiso9660 ~/Downloads/BarracudaLP-ALL-CC35.iso /mnt/loop
```

Setting HPA

Use *hdparm* with the *-N* option to find out the maximum number of visible sectors:

```
# hdparm -N /dev/sde

/dev/sde:
 max sectors   = 64000/976773168, HPA is enabled
```

Then, to disable the HPA set it to the max visisble sectors:

```
# hdparm --yes-i-know-what-i-am-doing -N p976773168 /dev/sde

/dev/sde:
 setting max visible sectors to 976773168 (permanent)
  max sectors   = 976773168/976773168, HPA is disabled
```

Setting DCO

To identify DCO on disk:

```
# hdparm --dco-identify /dev/sdb
```

To erase DCO on disk:

```
# hdparm --yes-i-know-what-i-am-doing --dco-restore /dev/sdb
```

Cloning Partition Table

Use **sfdisk**, this is part of the **util-linux** package. In debian, it is found in */usr/sbin/sfdisk*.

For GPT based disks, use gdisk.

1. Copy the partition table from the source disk:

```
# sfdisk -d /dev/sda > mbr
```

2. Restore the partition table on destination disk:

```
# sfdisk /dev/sdb < mbr
```

Inspecting Process Syscalls Using *sysdig*

Use **sysdig** to get detailed information about process system calls. To install sysdig on a debian based system if the package is not available in the repos:

```
$ curl -s https://s3.amazonaws.com/download.draios.com/stable/install-sysdig | sudo bash
```

For example, to see what calls are being made by *iceweasel* do the following:

```
$ sudo sysdig proc.name=iceweasel
10903 11:19:00.961549300 0 iceweasel (17398) > poll fds=5:e1 4:u1 8:p3 10:u1 22:p1 24:u1 3:f0 timeout
10908 11:19:00.961558641 0 iceweasel (17398) > switch next=0 pgft_maj=611 pgft_min=148114721 vm_size=
```

For a specific process id:

```
$ sudo sysdig thread.tid=922
2543694 12:16:34.481253335 0 ongserver (922) > write fd=0(<u>) size=2069216
2543695 12:16:34.481409710 0 ongserver (922) > switch next=910(pic-host) pgft_maj=0 pgft_min=22625 vm
```

The format of the output is quite similar to *tcpdump*. The output is as follows:

```
<evt.num> <evt.time> <evt.cpu> <proc.name> <thread.tid> <evt.dir> <evt.type> <evt.args>

where:

· evt.num is the incremental event number
· evt.time is the event timestamp
· evt.cpu is the CPU number where the event was captured
· proc.name is the name of the process that generated the event
· thread.tid id the TID that generated the event, which corresponds to the PID for single thread proc
· evt.dir is the event direction, > for enter events and < for exit events
· evt.type is the name of the event, e.g.  'open' or 'read'
· evt.args is the list of event arguments.
```

You can also pass the *-w <capture>* to capture the trace to a file and read it back using filters or *chisels* with *-r <capture>*.

Can also list available chilses with *-cl* and use *i <chisel>* to get info on chisel. Then use *-c chisel* with *-r <trace>* to filter out capture.

References

1. Sysdig + Logs: Advanced Log Analysis Made Easy
2. Sysdig for ps, lsof, netstat + time travel
3. Hiding Linux Processes For Fun And Profit

Check for problematic I/Os

Use **iostat** to see current read/write rates:

```
$ sudo iostat -d 1
Linux 3.16-2-amd64 (amit-debian)        10/02/2014      _x86_64_ (8 CPU)

Device:            tps    kB_read/s    kB_wrtn/s    kB_read kB_wrtn
sda               5.31        48.49        95.74    8472327 16726100

Device:            tps    kB_read/s    kB_wrtn/s    kB_read kB_wrtn
sda               0.00         0.00         0.00          0 0
```

-d is to show disk stats and *1* is to query every second.

To see I/Os and its respective processes with CPU usage, use **iotop**.

```
$ sudo iotop
Total DISK READ :       0.00 B/s | Total DISK WRITE :       7.64 K/s
Actual DISK READ:       0.00 B/s | Actual DISK WRITE:      42.03 K/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO> COMMAND
    168 be/3 root        0.00 B/s    7.64 K/s  0.00 %  2.80 % [jbd2/sda5-8]
    28565 be/4 root      0.00 B/s    0.00 B/s  0.00 %  0.27 % [kworker/1:5]
    26449 be/4 root      0.00 B/s    0.00 B/s  0.00 %  0.21 % [kworker/1:2]
    ...
```

-o shows only processes that are active and *-a* shows accumulated data read/written.

Tracing SUID Programs

You can use *strace* to trace SUID programs. Note that by default SUID programs can't be debugged or traced by ordinary users because this would allow tracing user to excute code as a different user (with privileges as user executing SUID program).

Thus, SUID programs can be executed without SUID bit and then traced. However, this is not ideal because you don't really want to change the program behavior by removing SUID bit.

You can also run strace as root. This will then run the program you are tracing as root which might be dangerous. Another way is to temporary set SUID root for strace. This also runs program as root.

Note that *strace* calls *ptrace* internally and affects program performance. Can use *ltrace* to just trace library calls.

CHAPTER 3

Debian

Debian

Setup

Install

The latest jessie install as of Sept 02, 2014 is *debian-jessie-DI-b1-amd64-netinst.iso*. This already comes by default with *systemd* as the init.

During the install, select all defaults and just select *SSH Server* and *Standard System Utilities* in tasksel.

Basic Setup

1. Enable *force_color_prompt* in *.bashrc*.
2. Update apt lists and upgrade all packages.
3. Login using *su* and install *sudo*.
4. Add the following lines using *visudo*:

```
# User privilege specification
logicube        ALL=(ALL:ALL) ALL
```

5. Install *avahi-daemon* so we don't have to remember IP address. We don't have to use no recommends here since the list of packages it recommends is only one.

```
$ sudo aptitude install avahi-daemon
```

6. Add *UseDNS no* to */etc/ssh/sshd_config* and restart *ssh* service.

systemd

1. Create */var/log/journal* where *systemd* can store persistent journals.
2. The directory should be owned by *systemd-journal*.

```
$ sudo chgrp systemd-journal /var/log/journal
```

3. Fix the permissions for this group so that any user that is a member of *systemd-journal* should be able to access it.

```
$ sudo chmod g+rwx /var/log/journal
```

4. Add the user to the *systemd-journal* group:

```
$ sudo usermod -a -G systemd-journal logicube
```

5. Reboot the system using *systemctl reboot*.

Enlightenment

1. Install basic *X*, use *-R* to not install recommended packages:

```
$ sudo aptitude install -R xserver-xorg xserver-xorg-core xinit xinput xserver-xorg-video-intel xserv
```

2. Install *e17*:

```
$ sudo aptitude install -R e17 fonts-droid librsvg2-common
```

3. Install a small display manager *CDM* https://wiki.archlinux.org/index.php/CDM.
   (a) Install *dialog* package as a dependency.
   (b) Install *git* (–without-recommends) to get the latest CDM from https://github.com/ghost1227/cdm.
   (c) Clone cdm to */tmp*.
   (d) Run *sudo ./install.sh*.
   (e) `sudo cp /usr/share/doc/cdm/profile.sh /etc/profile.d/zzz-cdm.sh`
   (f) `chmod +x /etc/profile.d/zzz-cdm.sh`
   (g) This procedure **doesn't work**, CDM gets stuck.
4. Install *nodm*. Edit */etc/default/nodm* and set *NODM_ENABLED* to *true*. Finally, change *NODM_USER* to *logicube*. Reboot the system.

Applications

1. Install *chromium* with recommended packages.
2. Install *rxvt-unicode-256color*.
3. Create *.Xdefaults* with the following settings:

```
URxvt.font: xft:Droid Sans Mono:style=Regular:pixelsize=15
!URxvt*letterSpace              : -1

! make a scrollbar that's nearly black
URxvt*scrollBar:                                false
URxvt*scrollBar_floating:                       true
URxvt*scrollBar_right:                          false
URxvt*scrollColor:                              #202020
URxvt*urllauncher:                              chromium

! matcher.button # 3 is a right-click
URxvt*matcher.button:                           3
URxvt*saveLines:                                8192
URxvt.perl-ext-common:                          default,matcher

! Theme
```

```
URxvt*background: #000000
URxvt*foreground: #ffffff
```

4. Install *vim* and *vim-gtk* with recommended packages.
5. Install *lxde-icon-theme*, *lxappearance*, *gnome-themes* with recommended packages.
6. Install *cifs-utils* for *mount.cifs* without recommended packages.

Commands

Contents

Finding hardlink of file

For example `zipinfo`:

```
$ ls -l /usr/bin/zipinfo
-rwxr-xr-x 2 root root 158360 Apr 24 14:41 /usr/bin/zipinfo
```

You can see the number 2 indicating number of links (including this one).

1. Use `ls -i` to find `inode` number of file:

```
$ ls -i /usr/bin/zipinfo
1187726 /usr/bin/zipinfo
```

1. Use find to search for that specific `inode`:

```
$ find /usr/bin/ -inum 1187726
/usr/bin/unzip
/usr/bin/zipinfo
```

When DKMS Build Fails Due to Missing Source

The kernel headers are enought to build a module. However, if building the DKMS fails and kernel headers package are installed, it usually means there is no *build* symlink under */lib/modules/‘uname -r‘*.

Do the following to create the symlink:

```
$ sudo ln -s /usr/src/linux-headers-$(uname -r)/ /lib/modules/$(uname -r)/build
```

No configure script, only configure.ac

Need to run the following set of commands to build a working configure script from a configure.ac:

```
$ libtoolize --force
$ aclocal
$ autoheader
$ automake --force-missing --add-missing
$ autoconf
$ ./configure --prefix=/usr
```

Transferring files over netcat

Sometimes this is useful when you need to transfer some files from a unit that has *busybox* or booted up to an *initramfs* shell.

First, on PC that you would like the file to be transferred to:

```
$ nc -l -p 6666 > dmesg.out
```

You can check if port *6666* is open by running:

```
$ lsof -i :6666
COMMAND PID USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
nc      953 amit    3u  IPv4 54834351      0t0  TCP *:6666 (LISTEN)
```

Then on busybox shell:

```
(initramfs) ip link set dev eth0 up
(initramfs) ip addr add 192.168.1.123/24 dev eth0
(initramfs) dmesg > /tmp/dmesg
(initramfs) nc 192.168.1.175:6666 < /tmp/dmesg
```

CHAPTER 4

Study

Study

Google I/O 2011: Life in App Engine Production

Notes taken from Google I/O 2011: Life in App Engine Production

1. Standalone computers/nodes usually have no problems. Problems are compounded when they are interconnected. Whethere in a data center or data centers connected to other data centers all over the world.
2. Traffic with peak and valleys coming from users. Systems needs to be designed to work in these cases.
3. *Leslie Lamport* is a computer scientist working with distributed systems. Creator of paxos distributed consensus algorithm used widely at Google.
4. App Engine is a cloud computing environment built inside Google's cloud computing environment.
    - It does not run in its own data center. It runs across nodes.
    - Cloud computing environment depends on a lot of services (storage, lock, computer, networking, etc.). Each of these are managed by other SREs.
    - Handshake between different layers of cloud computing with gurantees on certain reliability parameters. Then, each layer lives within the bounds of the parameters set in the layer below it.
    - Services is as reliable as the weakest (least reliable) service it depends on.
5. Google products and environment are designed for in-place updates. No disruption to users.
    - However, other upgrades such as network, power (generator), other hardware upgrades are intrusive. In this case, data is probably going to be re-routed to other data centers in the meantime.
    - Asynchronous replication happening from one datacenter to another.
6. Typical Google server serving storage has a storage process talking to a power management process that monitors power and battery (UPS) local to server.
    - Advantage of this communication is that when power's out, server does not accept anymore writes from network service.
    - UPS backup basically designed to allow cached writes to be committed to disk.
7. When error happens in server (in the case of power outage and no writes happening), design allows error to ripple back to user allowing user to retry request again. Thus, no lost state in user's end.
8. Monitoring processes lie to you sometimes. Idea is to *trust, but verify*.
    - Strong checksums at many layers of the stack are best.

- It's like infinite monkey, eventually the monkey will write out shakespeare. When there are so many hardware, somehow something will end up deleting your data.

9. With asynchronous replication, data needs time to catch up to the slave data center. What if the link goes down? In this case, slave datacenter can't start serving data since it will be serving stale data.
   - Google has high replication datastore (synchronous writes).
   - Uses paxos algorithm do handle the synchronous writes.
   - If local data store is too slow, service will ignore its own local data store and write over network to remote data store.
10. How to design reliable services?
    - Your application/data should live in multiple data centers.
      - Data centers can't be in same location, power, network, continent, etc.
      - Be capable of handling multiple data center failures simultaneously.
    - Use synchronous writes to most of your datacenters.
    - Let your system/infrastructure decide what datacenters to read and write from. Don't wait for humans to react and make decisions.
    - Instantaneous traffic rerouting on demand.
11. Advantages of advance monitoring.
    - Idea is you should know you are having problems before customers start calling you.
    - For example, if data center is facing slightly longer latency for requests, the monitoring tools will alert you before your customers do. **Your customers are not your monitoring tools.**
    - Example:
      - CPU usage rockets up.
      - Explore requests, not that many to the home page but a few extra requests to /news.
      - Then, look at memcache and datastore rates. Memcache rates are about the same, but datastore rates are way up.
      - Thus, requests to /news requires a lot of work and data is not cached properly (not using memcached). Results are not cached.
      - Monitoring API running on app engine itself! High reliability platform.

Google I/O 2011: More 9s Please: Under The Covers of the High Replication Datastore

Notes taken from Google I/O 2011: More 9s Please: Under The Covers of the High Replication Datastore

1. Two types of datastore in App Engine:
   - Master/Slave
     - This is the old style. There is one master that handles all the reads/writes and asynchronous writes happen to the slave.
   - High Replication
     - This is the new default style. There is no master in this one as writes happen to all nodes synchronously. All act as a collective master.

2. Datastore Stack:
    - The actual datastore is the highest level. This is schema-less storage and has advance query engine.
    - This sits atop *megastore* which is defined by a strict schema and queried using standard SQL.
    - Megastore is powered by *Bigtable* which is a distributed key-value store.
        - Big Table is super fast and highly scalable. However, this design has some tradeoffs. Mainly data can be unavailable for short periods of time
    - Finally, the file system that powers all this is *GFSv2*, a distributed filesystem.
3. Writes to Datastore
    - In a Master/Slave, write happens to Datacenter A and gets asynchronously written to Datacenter B at a later time.
    - In High Replication, write happens to a majority of the replicas synchronously. The other replica(s) that don't get the write synchronously gets an asynchronous write scheduled. Or can be on-demand replication when Read comes in to that datastore and it realizes that it doesn't have that data.
    - Writes to Master/Slave is faster (20ms) compared to High Replication Datastore (45ms).
    - Read latency is about the same but read error rate in High Replication is way less (0.001% vs 1%). Thus, resulting in 5m vs 9h downtime.
4. Planned Maintenance
    - Master/Slave
        - Datacenter A becomes readonly, thus app running on app engine will be readonly. In the meantime, the *catchup* happens to datacenter B.
        - Once that is done, then the switchover will happen.
        - Requires engineer to initiate switchover.
    - High Replication
        - Seamless migration. Switching is almost transparent.
        - *Memcache flush* + 1 min no-caching.
        - This is primarily hosted in a single datacenter. Reason is memcache is quite fast, and doing replication across datacenters is too slow.
5. Unplanned Maintenance
    - Master/Slave experiences immediate switchover. Thus, some data is lost and app is serving stale data. Up to devs to manually flush partial data that was written to Datacenter A to Datacenter B.
    - For High Replication, this is the same as a planned maintenance. Designed to withstand multiple datacenter failures.
6. Some Issues with Bigtable
    - Since multiple apps share the same Bigtable instance, a short period that the Bigtable is unavailable for that Datacenter can cause apps hosted by that datacenter to be unavailable. Note that this is only for Master/slave setup.
    - High replication does not get affected, since it will try a request on another bigtable in another datacenter.

The Art of Unix Programming

Notes taken from The Art of Unix Programming

Chapter 1: Philosophy

Contents

1. The Unix API is the closest thing to a hardware-independent standard for writing truly portable software that exists. It is no accident that what the IEEE originally called the Portable Operating System Standard quickly got a suffix added to its acronym and became POSIX. A Unix-equivalent API was the only credible model for such a standard.
2. Doug McIlroy (invertor of Pipes): This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.
3. Rob Pike (Master of C): Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.
4. *Rule of Repair: When you must fail, fail noisily and as soon as possible.*
5. *Rule of Modularity: Write simple parts connected by clean interfaces.* The only way to write complex software that won't fall on its face is to hold its global complexity down — to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can have some hope of upgrading a part without breaking the whole.
6. *Rule of Separation: Separate policy from mechanism; separate interfaces from engines.* Another way is to separate your application into cooperating front-end and back-end processes communicating through a specialized application protocol over sockets; we discuss this kind of design in Chapter 5 and Chapter 7. The front end implements policy; the back end, mechanism. The global complexity of the pair will often be far lower than that of a single-process monolith implementing the same functions, reducing your vulnerability to bugs and lowering life-cycle costs.
7. *Rule of Representation: Fold knowledge into data, so program logic can be stupid and robust.* Data is more tractable than program logic. It follows that where you see a choice between complexity in data structures and complexity in code, choose the former. More: in evolving a design, you should actively seek ways to shift complexity from code to data.
8. *Rule of Silence: When a program has nothing surprising to say, it should say nothing.* One of Unix's oldest and most persistent design rules is that when a program has nothing interesting or surprising to say, it should shut up. Well-behaved Unix programs do their jobs unobtrusively, with a minimum of fuss and bother. Silence is golden.
9. *Rule of Optimization: Prototype before polishing. Get it working before you optimize it.*
10. *Rule of Extensibility: Design for the future, because it will be here sooner than you think.* If it is unwise to trust other people's claims for "one true way", it's even more foolish to believe them about your own designs. Never assume you have the final answer. Therefore, leave room for your data formats and code to grow; otherwise, you will often find that you are locked into unwise early choices because you cannot change them while maintaining backward compatibility.
11. *To do the Unix philosophy right, you have to be loyal to excellence.* You have to believe that software design is a craft worth all the intelligence, creativity, and passion you can muster. Otherwise you won't look past the easy, stereotyped ways of approaching design and implementation; you'll rush into coding when you should be thinking. You'll carelessly complicate when you should be relentlessly simplifying — and then you'll wonder why your code bloats and debugging is so hard.
12. Software design and implementation should be a joyous art, a kind of high-level play.

    If this attitude seems preposterous or vaguely embarrassing to you, stop and think; ask yourself what you've forgotten. Why do you design software instead of doing something else to make money or pass the time? You

must have thought software was worthy of your passion once....

To do the Unix philosophy right, you need to have (or recover) that attitude. You need to care. You need to play. You need to be willing to explore.

Chapter 2: Origins and History of Unix

Contents

1. Multics designed for time-sharing mainframe systems. Was too complicated and thus Unix was born from the ashes of Multics. The design was to not repeat same mistakes and thus much simpler design.
2. Invented by Ken Thompson from Bell Laboratories. Partnered with Dennis Ritchie (labeled co-inventor) and creator of C programming language. This was around 1969-1970.
3. Developed on PDP-7.
4. Unix was originally called "UNICS" (UNiplexed Information and Computing Service).
5. Multics had thousand of pages of specs while Unix was designed and programmed originally by three people (third person was Doug McIlroy).
6. Unix was originally written in assembler and an interpreted language called *B*. But *B* was not powerful enough to do systems programming, thus Ritchie added data types and structure. Thus, *C* was born in 1971.
7. In 1973, Unix was re-written completely in C to achieve better performance.
8. Ritchie and Thompson wrote: **"constraint has encouraged not only economy, but also a certain elegance of design"**.
9. Unix was given to corporations, academia, etc. The improvements were shared back to Bell Labs and Version 7 of Unix was released by Bell Labs in late 70s included all these improvements.
10. Ken Thompson ended up teaching at Berkley during 75-76 sabbatical and further influenced an already strong influence of Unix research at Berkley. First BSD release was in 77 and a lot of great software came out of Berkley labs (including *vi* editor). Bill Joy was a grad student heading the labs at Berkley that released the BSDs.
11. DARPA chose Berkley Unix as a platform to implement its brand new TCP/IP protocol stack (which was running on VAX at the time). First released in 83 with Berkley 4.2 Unix.
12. 1981 Microsoft partnered with IBM and marketed MS-DOS (re-packaged QDOS "Quick and Dirty OS".
13. 1982 Bill Joy founded Sun Microsystems with two others. Found the workstation industry by building together hardware designed from Stanford and OS from BSD.
14. DEC cancelled successor to PDP-10 and VAXs running Unix were powering Internet backbone (until being displaced by Sun Microsystems). When DEC cancelled PDP-10's successor, MIT AI Lab's PDP-10 hacker named **Richard Stallman** became motivated to build a completely free clone of Unix called GNU.
15. Productization of Unix happened and there were many commercial versions. AT&T marketed System V licenses around 1983 when anti-trust department broke them up again. This destroyed free exchange of source code.
16. Unix became heavily fragmented, which each commercialized version marketing their differences. Also, Unix players ignored Microsoft's rise in the commercial personal computer market.
17. Rivalry between System V and BSD - sockets vs streams. Corporations sided with AT&T System V while programmers and hackers backed BSD.

18. Around the early 80s, a programmer/linguist named Larry Wall quietly invented the **patch** utility. Huge impact on Unix development. Now, programmers could send *diffs* instead of whole files.
19. When Intel shipped the first 386 chip in 1985, it could address 4GB of memory and was powerful enough to run Unixes. This started the end of workstation companies such as Sun Microsystems.
20. 1985 was also the year Stallman published GNU Manifesto and X window was released with full source code under the X permissive license. Which resulted in X becoming de-facto graphics engine in all Unixes.
21. Unix standardization started in 1983 with System V and BSD started reconciling their APIs. This became officially the POSIX standard in 1985. Used superior Berkley job control and signal handling with System V terminal handling. Only major Unix API to come after was Berkley **sockets**.
22. Larry Wall created Perl in 1986, first and most widely used open-source scripting language. 1987 first version of GCC is released. Thus, GNU now had compiler, editor, debugger, and other basic tools to arm next gen developers in the 90s (along with almost all workstations running X).
23. While Unix wars were going on, Microsoft released Windows 3.0 in 1990 and sealed its dominance in the personal computer market.
24. Unix hackers preferred Motorola's elegant RISC based 68000 processor compared to Intel's ugly 8086 arch. But Motorola lost out to Intel's inexpensive chips. Also, GNU failed to release a free Unix clone by this time.
25. Finally in 1991 Linus Torvalds a grad student from Finland announced Linux. He wanted a free and cheap clone of Unix running on 386 hardware. There was 386BSD that started in 90s but was not shipped until 92.
26. 1993-1994 Internet exploded and so did development on Linux and BSD. BSD suffered because AT&T started lawsuits alleging copied source code. Thus, motivated some BSD developers to jump and develop for Linux.
27. XFree86 used Internel development and was more effective than X consortium and provided BSD and Linux with graphics engine in 1992.
28. Internet Engineering Task Force (IETF) originated the tradition of standardization through Requests For Comment (RFCs). This started from same group of *hackers* who managed ARPANET at MIT AI Lab.
29. Interesting was that these *geeks* were not Unix programmers. Early Unix programmers were from academia and corporations that were directly involved in Unix. However, these *geeks* were young and bright and sharing through the Internet was their *religion*.
30. Eventually, ARPANET hackers learned Unix and C and Unix hackers learned TCP/IP.
31. RMS created *Free Software* term that labeled the goal of a lot of *hackers*. However, not all hackers believed in it. BSD license remained popular as well.
32. Most hackers did not want to get into the GPL/anti-GPL debate and just wrote code. Linus Torvalds used this effectively and licensed his kernel with GPL to protect it and used the mature GNU user land tools. He avoided the *religious* aspects of the GPL and did not like the strong ideology behind it. He sometimes even used proprietary programs when there was no better Free Software alternative. This made hackers follow his ideology more.
33. Around 1993-1997, Linux already had a strong technical foundation and also had distributions, support services, and strong development community.
34. When Linux 0.1 was released in 1995, it could beat proprietary Unixes in performance and uptime. At this time *Apache* webserver was released for Linux and immediately was the most popular web server due to its stability and free nature. This cemented Linux as a server platform.
35. According to Eric S. Raymond (author of *TAOUP*) **"Given a sufficiently large number of eyeballs, all bugs are shallow"**. Thus, the argument changed from **Free software because all software should be free** to **Free software because it works better**.
36. The paper's contrast between *cathedral* (centralized, closed, controlled, secretive) and *bazaar* (decentralized, open, peer-review-intensive) modes of development became a central metaphor in the new thinking.

37. Early 1998, Netscape inspired by the new thinking released Mozilla browser as open source. This brought Linux to Wall Street with the tech boom.
38. In March of 1998 an unprecedented summit meeting of community influence leaders representing almost all of the major tribes convened to consider common goals and tactics. That meeting adopted a new label for the common development method of all the factions: **open source**.
39. Most Unix hackers and tribes adopted this new *open source* banner. However, the major standout was RMS. He specifically did not want *Free Software* = *Open Source*. He claimed an ideological difference.
40. The other main intention behind *open source* was to present the hacker community's method to the rest of the world. And this was a success.
41. The largest-scale pattern in the history of Unix is this: when and where Unix has adhered most closely to open-source practices, it has prospered. Attempts to proprietarize it have invariably resulted in stagnation and decline.
42. The lesson for the future is that over-committing to any one technology or business model would be a mistake — and maintaining the adaptive flexibility of our software and the design tradition that goes with it is correspondingly imperative.
43. Never bet against cheap-plastic and commodity solutions in Economy. They always win. This is how Linux has thrived.

Chapter 3: Contrasts

Contents

1. Different operating systems were designed by the influences of culture, limitations (usually economic), and ideas of their designers.
2. The designer's idea is usually is baked into the operating system and thus unifies its design. For Unix, this idea was *everything is a file* and *pipes* metaphor that builds on top of this.
3. To design the perfect anti-Unix, have no unifying idea at all, just an incoherent pile of ad-hoc features.
4. One way in which OSes differ is in the way they handle multiple processes or *Multitasking*. *DOS* and *CP/M* were basically sequential loaders with to multitasking abilities.
5. *Cooperative Multitasking* is the ability to share multiple processes. However, there was no memory management unit or locking. Thus, a bug in a program could freeze the entire system.
6. Unix has *preemptive multitasking*, in which timeslices are allocated by a scheduler which routinely interrupts or pre-empts the running process in order to hand control to the next one. Almost all modern operating systems support preemption.

7. Note that *multitasking* does not mean *multiuser*. Many OSes are multitasking but can only support one user at a time logged in to the machine. True multi user requires multiple user privilege domains (multi console).
8. In the Unix experience, **inexpensive process-spawning and easy inter-process communication (IPC)** makes a whole ecology of small tools, pipes, and filters possible.
9. A subtle but important property of pipes and the other classic Unix IPC methods is that they require communication between programs to be held down to a level of simplicity that encourages separation of function. Conversely, the result of having no equivalent of the pipe is that programs can only be designed to cooperate by building in full knowledge of each others' internals.
10. In operating systems without flexible IPC and a strong tradition of using it, programs communicate by sharing elaborate data structures.
11. *Doug McIlroy*: Word and Excel and PowerPoint and other Microsoft programs have intimate — one might say promiscuous — knowledge of each others' internals. In Unix, one tries to design programs to operate not specifically with each other, but with programs as yet unthought of.
12. Unix encourages interal boundaries by encouraging creating different users with different privileges. System programs often have their own pseudo-user accounts to confer access to special system files without requiring unlimited (or superuser) access.
13. Unix has at least three levels of internal boundaries:
    - Unix uses its hardware's memory management unit (MMU) to ensure that separate processes are prevented from intruding on the others' memory-address spaces.
    - A second is the presence of true privilege groups for multiple users — an ordinary (nonroot) user's processes cannot alter or read another user's files without permission.
    - A third is the confinement of security-critical functions to the smallest possible pieces of trusted code. Under Unix, even the shell (the system command interpreter) is not a privileged program.
14. OSes need strong internal boundaries for stability and security.
15. Unix files have neither record structure nor attributes. Other OSes know about the file and the type of the file. For example, other OSes associate file extension with application to open that file. In Unix, applications recognize the files by their *magic number* or other data type within the file itself.
16. OS-level record structures are generally an optimization hack, and do little more than complicate APIs and programmers' lives. They encourage the use of opaque record-oriented file formats that generic tools like text editors cannot read properly.
17. Critical data in Unix is stored in text files. Thus, it can easily read by programs (not a security risk since critical data such as passwords are salted and stored as hashes). Binary data is evil.
18. In Unix, belief is that OS should have strong CLI facilities because of the following reasons:
    - Easy remote administration.
    - Programs will not be designed to cooperate with each other in a nice way.
    - Servers, daemons, and other background programs will be difficult to program.
19. Unix is designed for programmers by programmers. Thus, it makes no assumptions on what the user needs or wants. Other OSes designed for end users often make these assumptions and sometimes get them wrong.
20. In Unix, there is no major barrier for a user to become a developer. The culture promotes it and the development tools are freely available for everyone. *Unix* pioneered casual programming.

Operating System Comparisons

VMS

1. VMS was released by DEC in 1978 and still kind of survives (maybe receives support). It is also a CLI based OS.
2. VMS has full preemptive multitasking, but makes process-spawning very expensive. The VMS file system has an elaborate notion of record types (though not attributes).
3. Had elaborate COBOL system commands and extensive help system. But commands where quite long to type and help system had no good search functionality.
4. VMS had MMU and true multiuser capabilities. Security cracks on VMS were quite rare.
5. VMS dev tools and docs were expensive. Docs were only available in paper form and thus was tiresome to go through and search through.

MacOS

1. Debut in 1984 with the Macintosh and has heavy GUI influenced designed obtained from Xerox's Palo Alto Research Center.
2. MacOS's very strong unifying idea was its GUI guidelines. Specified in great detail how the application should look like and behave.
3. One key idea of these guidelines was that all the documents, directories, and other persistent objects had its place in the desktop and desktop context was preserved across reboots.
4. All programs have GUIs. MacOS's captive-interface GUI metaphor (organized around a single main event loop) leads to a weak scheduler without preemption. The weak scheduler, and the fact that all MultiFinder applications run in a single large address space, implies that it is not practical to use separated processes or even threads rather than polling.
5. MacOS applications are not, however, invariably monster monoliths. The system's GUI support code, which is partly implemented in a ROM shipped with the hardware and partly implemented in shared libraries, communicates with MacOS programs through an event interface that has been quite stable since its beginnings. Thus, the design of the operating system encourages a relatively clean separation between application engine and GUI interface.
6. MacOS files have both a 'data fork' (a Unix-style bag of bytes that contains a document or program code) and a 'resource fork' (a set of user-definable file attributes). Mac applications tend to be designed so that (for example) the images and sound used in them are stored in the resource fork and can be modified separately from the application code.
7. The MacOS system of internal boundaries is very weak. There is a wired-in assumption that there is but a single user, so there are no per-user privilege groups. Multitasking is cooperative, not pre-emptive.
8. Security cracks against MacOS machines are very easy to write; the OS has been spared an epidemic mainly because very few people are motivated to crack it.
9. Mac OS X merged the above ideas with the strong internals of BSD Unix. At the same time, leading-edge Unixes such as Linux are beginning to borrow ideas like file attributes (a generalization of the resource fork) from MacOS.

OS/2

1. Currently (2003) still used in some automated teller machines. Never really was competition to MacOS or Windows. Was initially designed as an *advanced DOS*.
2. OS/2 was designed with preemptive multitasking and thus would not run on systems without an MMU. However, it was not designed to be multiuser. Also, it allowed for relatively inexpensive process spawning but had a difficult IPC.
3. Had networking support for LAN protocols but TCP/IP was later added.

4. Had both CLI/GUI. The OS/2 WPS (Workplace Shell) was its desktop. It was licensed from AmigaOS and had strong and clean object-oriented design and good extensibility. This would become the model from GNOME desktop.
5. OS/2 had the internal boundaries one would expect in a single-user OS. Running processes were protected from each other, and kernel space was protected from user space, but there were no per-user privilege groups. This meant the file system had no protection against malicious code. Another consequence was that there was no analog of a home directory; application data tended to be scattered all over the system.
6. Since there were no per-user privilege group, trusted programs would be jammed into kernel or WPS thus resulting in bloat.
7. Used both text and binary formats.
8. Eventually IBM released tools for free and hobby groups evolved but was pushed towards Java because of Microsoft's dominance on the desktop. Finally, a lot of devs moved towards Linux.
9. Lesson learned, can't really go too far with multitasking OS with no multi-user capabilities.

Windows NT (New Technology)

1. Designed for high-end personal and server use. All Microsoft's OSes from Windows 2000 onwards are NT based.
2. NT genetically descended from VMS. NT grew by accretion (continuous growth by adding layers) and doesn't really have a unifying design idea like MacOS or Unix.
3. Technology becomes obsolete every few years and devs have to re-learn APIs, concepts.
4. Pre-emptive multitasking is supported but process spawning is several times more expensive (0.1s) than Unix.
5. Makes extensive use and distinction between binary formats and text files.
6. Programs communicate via complex and fragile RPCs.
7. System configuration is stored in registries.
    - The registry makes the system completely non-orthogonal. Single-point failures in applications can corrupt the registry, frequently making the entire operating system unusable and requiring a reinstall.
    - The registry creep phenomenon: as the registry grows, rising access costs slow down all programs.
8. NT has weak internal boundaries. Although it has access control lists, they are ignored by older programs.
9. To achieve speed, recent versions of the NT wire the webserver into the kernel to achieve the same speed as Unix.
10. These holes in the boundaries have the synergistic effect of making actual security on NT systems effectively impossible.
11. Because Windows does not handle library versioning properly, it suffers from a chronic configuration problem called "DLL hell", in which installing new programs can randomly upgrade (or even downgrade!) the libraries on which existing programs depend.
12. Microsoft started to publish all APIs and kept tools inexpensive. However, around Windows 95 time frame, they started to hide APIs and did not publish internal APIs to the general public. Only devs who signed NDAs could use them.

BeOS

1. Started out as a hardware vendor building machines around PowerPC arch in 1989.

2. BeOS was Be's attempt to add value to the hardware by inventing a new, network-ready operating system model incorporating the lessons of both Unix and the MacOS family, without being either. The result was a tasteful, clean, and exciting design with excellent performance in its chosen role as a multimedia platform.
3. BeOS's unifying ideas were 'pervasive threading', multimedia flows, and the file system as database. Designed also to minimize latency in the kernel. BeOS 'threads' were actually lightweight processes in Unix terminology, since they supported thread-local storage and therefore did not necessarily share all address spaces. IPC via shared memory was fast and efficient.
4. Followed Unix by having no file structure above byte level but halso had file attributes ala MacOS. The filesystem database could be indexed by any attribute.
5. One of the things BeOS took from Unix was intelligent design of internal boundaries. It made full use of an MMU, and sealed running processes off from each other effectively. While it presented as a single-user operating system (no login), it supported Unix-like privilege groups in the file system and elsewhere in the OS internals. Easy to add multi-user capability. There was a guest user (default) and a root user.
6. BeOS tended to use binary file formats and the native database built into the file system, rather than Unix-like textual formats.
7. Had clean GUI but also good CLI (port of bash). Had a POSIX compatibility layer as well.
8. Was designed as a multimedia workstation. Followed Apple in only allowing BeOS to run in its own hardware. Eventually there were lawsuits by Microsoft and Linux started gaining some multimedia capabilities. Finally, it tried releasing an x86 port but it was too late and by 2001 it was pretty much obscure.

MVS

1. Multiple Virtual Storage was IBM's flagship OS for mainframes.
2. Older than Unix so there really isn't much Unix design principles in it. Unifying idea is that all work is a *batch*. The system is designed to make the most efficient possible use of the machine for batch processing of huge amounts of data, with minimal concessions to interaction with human users.
3. Process spawning is a slow operation. The I/O system deliberately trades high setup cost (and associated latency) for better throughput. These choices are a good match for batch operation, but deadly to interactive response.
4. MVS uses the machine MMU; processes have separate address spaces. Interprocess communication is supported only through shared memory. There are facilities for threading (which MVS calls "subtasking"), but they are lightly used, mainly because the facility is only easily accessible from programs written in assembler.
5. Many system configuration files are in text format, but application files are usually in binary formats specific to the application.
6. File system security was an afterthought in the original design. However, when security was found to be necessary, IBM added it in an inspired fashion: They defined a generic security API, then made all file access requests pass by that interface before being processed. As a result, there are at least three competing security packages with differing design philosophies — and all of them are quite good, with no known cracks against them between 1980 and mid-2003.
7. There is no concept of one interface for both network connections and local files; their programming interfaces are separate and quite different.
8. Casual programming for MVS is almost nonexistent except within the community of large enterprises that run MVS.
9. The intended role of MVS has always been in the back office.

VM/CMS

1. VM/CMS is IBM's other mainframe operating system. Historically speaking, it is Unix's uncle: the common ancestor is the CTSS system, developed at MIT around 1963 and running on the IBM 7094 mainframe. The group that wrote CTSS went on to write Multics.
2. The unifying idea of the system, provided by the VM component, is virtual machines, each of which looks exactly like the underlying physical machine.
3. A scripting language called Rexx supports programming in a style not unlike shell, awk, Perl or Python. Consequently, casual programming (especially by system administrators) is very important on VM/CMS.
4. VM/CMS even went through the same cycle of de facto open source to closed source back to open source, though not as thoroughly as Unix did.
5. What VM/CMS lacks, however, is any real analog to C. Both VM and CMS were written in assembler and have remained so implemented.
6. Since the year 2000, IBM has been promoting VM/CMS on mainframes to an unprecedented degree — as ways to host thousands of virtual Linux machines at once.

Linux

1. Linux does not include any code from the original Unix source tree, but it was designed from Unix standards to behave like a Unix.
2. The desire to reach end users has also made Linux developers much more concerned with smoothness of installation and software distribution issues than is typically the case under proprietary Unix systems. One consequence is that Linux features binary-package systems far more sophisticated than any analogs in proprietary Unixes, with interfaces designed (as of 2003, with only mixed success) to be palatable to nontechnical end users.
3. Linux 2.5's incorporation of extended file attributes (using `getfattr(1)` and `setfattr(1)`), which among other things can be used to emulate the semantics of the Macintosh resource fork, is a recent major one at time of writing. This mainly to support other filesystems from other OSes natively on Linux.
4. Indeed, a substantial fraction of the Linux user community is understood to be wringing usefulness out of hardware as technically obsolete today as Ken Thompson's PDP-7 was in 1969. As a consequence, Linux applications are under pressure to stay lean and mean that their counterparts under proprietary Unix do not experience.

What Goes Around Comes Around

1. Many of the major OSes today have adopted Unix principles. For example, MacOS merged Unix to its core. Windows is the only major alternative.
2. In a world of pervasive networking, even an operating system designed for single-user use needs multiuser capability (multiple privilege groups) — because without that, any network transaction that can trick a user into running malicious code will subvert the entire system (Windows macro viruses are only the tip of this iceberg).
3. Windows gets away with having severe deficiencies in these areas only by virtue of having developed a monopoly position before networking became really important, and by having a user population that has been conditioned to accept a shocking frequency of crashes and security breaches as normal. This is not a stable situation, and it is one that partisans of Linux have successfully (in 2003) exploited to make major inroads in the server-operating-system market.
4. The trend toward client operating systems was so intense that server operating systems were at times dismissed as steam-powered relics of a bygone age.

5. But as the designers of BeOS noticed, the requirements of pervasive networking cannot be met without implementing something very close to general-purpose timesharing. Single-user client operating systems cannot thrive in an Internetted world.
6. Retrofitting server-operating-system features like multiple privilege classes and full multitasking onto a client operating system is very difficult, quite likely to break compatibility with older versions of the client, and generally produces a fragile and unsatisfactory result rife with stability and security problems.
7. Retrofitting a GUI onto a server operating system, on the other hand, raises problems that can largely be finessed by a combination of cleverness and throwing ever-more-inexpensive hardware resources at the problem. As with buildings, it's easier to repair superstructure on top of a solid foundation than it is to replace the foundations without trashing the superstructure.
8. The Unix design proved more capable of reinventing itself as a client than any of its client-operating-system competitors were of reinventing themselves as servers.

Chapter 4: Modularity

Contents

1. In the beginning, everything was one big lump of machine code. The earliest procedural languages brought in the notion of partition by subroutine. Then we invented service libraries to share common utility functions among multiple programs. Next, we invented separated address spaces and communicating processes. Today we routinely distribute program systems across multiple hosts separated by thousands of miles of network cable.
2. Modularity of hardware has of course been one of the foundations of engineering since the adoption of standard screw threads in the late 1800s.
3. The only way to write complex software that won't fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local and you can have some hope of fixing or optimizing a part without breaking the whole.
4. Dennis Ritchie encouraged modularity by telling all and sundry that function calls were really, really cheap in C. However, this wasn't really the case at first but Dennis tricked everyone! However, by then everyone was hooked.
5. The first and most important quality of modular code is encapsulation. Well-encapsulated modules don't expose their internals to each other. They don't call into the middle of each others' implementations, and they don't promiscuously share global data. They communicate using application programming interfaces (APIs) — narrow, well-defined sets of procedure calls and data structures. This is what the Rule of Modularity is about.
6. The APIs enforce a strong isolation and also it defines what the architecture of the program is.
7. One good test for whether an API is well designed is this one: if you try to write a description of it in purely human language (with no source-code extracts allowed), does it make sense?
8. Some of the most able developers start by defining their interfaces, writing brief comments to describe them, and then writing the code — since the process of writing the comment clarifies what the code must do. Such descriptions help you organize your thoughts, they make useful module comments, and eventually you might want to turn them into a roadmap document for future readers of the code.

9. There has to be a trade-off. Can't really have super small modules or very large modules. There has to be a balance.
10. **Brooks's Law predicts that adding programmers to a late project makes it later. More generally, it predicts that costs and error rates rise as the square of the number of programmers on a project.**
11. In nonmathematical terms, Hatton's empirical results imply a sweet spot between 200 and 400 logical lines of code that minimizes probable defect density, all other factors (such as programmer skill) being equal.

Compactness

1. Compactness is the property that a design can fit inside a human being's head. A good practical test for compactness is this: Does an experienced user normally need a manual? If not, then the design (or at least the subset of it that covers normal use) is compact. Idea is compact tools make you productive.
2. Apparently Lisp language is compact but has difficult concepts. But once the user masters these concepts, the concepts become simple.
3. The Unix system call API is semi-compact, but the standard C library is not compact in any sense.
4. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information [Miller] is one of the foundation papers in cognitive psychology (and, incidentally, the specific reason that U.S. local telephone numbers have seven digits). It showed that the number of discrete items of information human beings can hold in short-term memory is seven, plus or minus two. This gives us a good rule of thumb for evaluating the compactness of APIs: Does a programmer have to remember more than seven entry points? Anything larger than this is unlikely to be strictly compact.
5. Among general-purpose programming languages, C and Python are semi-compact; Perl, Java, Emacs Lisp, and shell are not (especially since serious shell programming requires you to know half-a-dozen other tools like sed(1) and awk(1)). C++ is anti-compact — the language's designer has admitted that he doesn't expect any one programmer to ever understand it all.
6. Sometimes, can't really make programs compact but this has to be last choice. An example is BSD sockets API. Can't really make this compact because of the complexity of the problem it is trying to solve.

Orthogonality

1. Orthogonality is one of the most important properties that can help make even complex designs compact. In a purely orthogonal design, operations do not have side effects; each action (whether it's an API call, a macro invocation, or a language operation) changes just one thing without affecting others. There is one and only one way to change each property of whatever system you are controlling.
2. One common class of design mistake, for example, occurs in code that reads and parses data from one (source) format to another (target) format. **A designer who thinks of the source format as always being stored in a disk file may write the conversion function to open and read from a named file. Usually the input could just as well have been any file handle.** If the conversion routine were designed orthogonally, e.g., without the side effect of opening a file, it could save work later when the conversion has to be done on a data stream supplied from standard input, a network socket, or any other source.
3. There is an excellent discussion of orthogonality and how to achieve it in *The Pragmatic Programmer [Hunt-Thomas]*. As they point out, orthogonality reduces test and development time, because it's easier to verify code that neither causes side effects nor depends on side effects from other code.
4. The concept of refactoring, which first emerged as an explicit idea from the 'Extreme Programming' school, is closely related to orthogonality. To refactor code is to change its structure and organization without changing its observable behavior.

5. The basic Unix APIs were designed for orthogonality with imperfect but considerable success. We take for granted being able to open a file for write access without exclusive-locking it for write, for example.
6. There are large non-orthogonal patches like the BSD sockets API and very large ones like the X windowing system's drawing libraries.

The SPOT Rule

1. Coined by Brian Kernighan: *Single Point of Truth*
2. Constants, tables, and metadata should be declared and initialized once and imported elsewhere. Any time you see duplicate code, that's a danger sign. Complexity is a cost; don't pay it twice.
3. Use tools such as code generators to generate common data that is being represented in multiple places. Have the data in one place and use the tool to generate the representations in different places.
4. If documentation duplicates what you say in the code, use a document generator that generates the docs from your code comments.
5. Try and generate header files and interface declarations automatically.
6. *No junk, No confusion*. Data structures should be designed to fit one representation of data well. Don't make it so generic. Try to also make data structure represent the real thing you are trying to model.
7. This is an often-overlooked strength of the Unix tradition. Many of its most effective tools are thin wrappers around a direct translation of some single powerful algorithm.
8. Doug McIlroy: By virtue of a mathematical model and a solid algorithm, Unix diff contrasts markedly with its imitators. First, the central engine is solid, small, and has never needed one line of maintenance. Second, the results are clear and consistent, unmarred by surprises where heuristics fail.
9. Other examples, are *grep* which is a thin wrapper around a formal algebra of regexs. *yacc* is based on LR-1 grammars at its core.
10. The opposite of a formal approach is using heuristics—rules of thumb leading toward a solution that is probabilistically, but not certainly, correct.
11. Sometimes, can't avoid designing using heuristics. Mail spam filtering uses heuristics since there really isn't a mathematical model describing spam.
12. Virtual memory management is also built on heuristics.
13. *"...constraint has encouraged not only economy, but also a certain elegance of design"*. That simplicity came from trying to think not about how much a language or operating system could do, but of how little it could do — not by carrying assumptions but by starting from zero (what in Zen is called "beginner's mind" or "empty mind").

Software Is a Many-Layered Thing

1. Can approach from bottom up or top-down. Bottom up is like *seeking to physical block*, *writing to physical block*, *turn on/off LED*. Top-down is more like *write to logical block*, or *toggle activity indicator*. Top-down is more generic and can apply to different hardware.
2. A very concrete way to think about this difference is to ask whether the design is organized around its main event loop (which tends to have the high-level application logic close to it) or around a service library of all the operations that the main loop can invoke.
3. In the example of web browser, top-down approach focuses on what user will input in the URL (e.g. *file*, *http*, *ftp*, etc.). Bottom up will focus on establishing network connections or handling GUI.

4. Which end of the stack you start with matters a lot, because the layer at the other end is quite likely to be constrained by your initial choices.
5. From top-down you might feel constrained about some domains your application logic initially did not plan for. For bottom-up, you might be designing unnecessary functions that you might never use.
6. Usually programmers are encouraged top-down approach. But the problem sometimes designing that way will involve some redesign since it doesn't pass real-world checks.
7. In self-defense against this, programmers try to do both things — express the abstract specification as top-down application logic, and capture a lot of low-level domain primitives in functions or libraries, so they can be reused when the high-level design changes.
8. Unix programmers, are more focused on systems programming. Thus, they write low-level wrappers for hardware operations and build from that. Thus, they are more bottom-up.
9. Bottom-up can give you time to redefine what the application is going to be. So you can start with the building blocks first without really knowing what the actual design on the application will be.
10. Real code, therefore tends to be programmed both top-down and bottom-up. Often, top-down and bottom-up code will be part of the same project. That's where 'glue' enters the picture.

Glue

1. One of the lessons Unix programmers have learned over decades is that glue is nasty stuff and that it is vitally important to keep glue layers as thin as possible. Glue should stick things together, but should not be used to hide cracks and unevenness in the layers.
2. The thin-glue principle can be viewed as a refinement of the Rule of Separation. Policy (the application logic) should be cleanly separated from mechanism (the domain primitives), but if there is a lot of code that is neither policy nor mechanism, chances are that it is accomplishing very little besides adding global complexity to the system.
3. *C* is an example of a very good thin glue. Designed for the *classic architecture*. Basically, a typical computer architecture: *unary representation, flat address space, a distinction between memory and working store (registers), general-purpose registers, address resolution to fixed-length bytes, two-address instructions, big-endianness, and data types a consistent set with sizes a multiple of 4 bits*.
4. *C* was designed to run on architectures similar to PDP-11 (which it was developed on). PDP-11 arch became a good model for future microprocessor architectures. Thus, *C* was a natural fit in future microprocessors.
5. This history is worth recalling and understanding because C shows us how powerful a clean, minimalist design can be. If Thompson and Ritchie had been less wise, they would have designed a language that did much more, relied on stronger assumptions, never ported satisfactorily off its original hardware platform, and withered away as the world changed out from under it.
6. Antoine de Saint-Exupéry once put it, writing about the design of airplanes: *La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever. ("Perfection is attained not when there is nothing more to add, but when there is nothing more to remove".)*

Libraries

1. If you are careful and clever about design, it is often possible to partition a program so that it consists of a user-interface-handling main section (policy) and a collection of service routines (mechanism) with effectively no glue at all. Effectively, these are libraries.
2. An important form of library layering is the plugin, a library with a set of known entry points that is dynamically loaded after startup time to perform a specialized task. For plugins to work, the calling program has to be organized largely as a documented service library that the plugin can call back into.

Unix and Object-Oriented Languages

1. In object-oriented programming, the functions that act on a particular data structure are encapsulated with the data in an object that can be treated as a unit. By contrast, modules in non-OO languages make the association between data and the functions that act on it rather accidental, and modules frequently leak data or bits of their internals into each other.
2. The OO design concept initially proved valuable in the design of graphics systems, graphical user interfaces, and certain kinds of simulation. To the surprise and gradual disillusionment of many, it has proven difficult to demonstrate significant benefits of OO outside those areas. It's worth trying to understand why.
3. Unix programmers don't really like OO since it encourages abstractions and thick glue layers. Since it is easy to create abstractions, it is everywhere. Unix programmers like the thin glue layer C provides.

Coding for Modularity

1. A good test for API complexity is: Try to describe it to another programmer over the phone. If you fail, it is very probably too complex, and poorly designed.
2. Do any of your APIs have more than seven entry points? Do any of your classes have more than seven methods each? Do your data structures have more than seven members?
3. Globals also mean your code cannot be reentrant; that is, multiple instances in the same process are likely to step on each other.

Chapter 5: Textuality

Contents

It's a well-known fact that computing devices such as the abacus were invented thousands of years ago. But it's not well known that the first use of a common computer protocol occurred in the Old Testament. This, of course, was when Moses aborted the Egyptians' process with a control-sea.

– Tom Galloway rec.arts.comics, February 1992

1. **Good Protocols Make Good Practice**
2. Two different designs in Unix are closely related. Both involve serialization of in-memory data structures.
   - Design of file formats for storing application data.
   - And design of application protocols for passing data between programs and network.
3. For transmission and storage, the traversable, quasi-spatial layout of data structures like linked lists needs to be flattened or serialized into a byte-stream representation from which the structure can later be recovered.
4. Interoperability, transparency, extensibility, and storage or transaction economy: these are the important themes in designing file formats and application protocols. Note that these are in order of most important to least important.
   - For interoperability and transparency it is important to have clear and clean data representations rather than thinking of what is best for performance.
   - Extensibility is suitable for text formats since they can be extended very easily.

- Storage or transtion economy often pushes the opposite direction but it is not wise to put this of importance first.

5. Pipes and sockets will pass binary data as well as text. However, it is always better to use textual data because they are easy for humans to read and write.
6. Some programmers are concerned about performance and size. Thus, they design binary formats. But implementing compression below or above the application protocol will probably result in a better, faster design since text compresses very well.
7. Textual protocol future proofs your system. IPv4 allowed only for 32-bits to an address, changing to 128-bits took a long time and a lot of effort.
8. Sometimes binary protocol should be used. For example, to get the most bit-density out of your data (e.g. multimedia) or very concerned about speed (e.g. network protocols with hard latency requirements).
9. SMTP and HTTP are text protocols but are very bandwidth intensive. The smallest X-server request is 4 bytes, while the smallest HTTP request is 100 bytes. Thus, X requests can be executed in about 100 instructions while Apache can take around 7000 instructions to process an HTTP request.
    - Problem eventually showed up in X, where it was very difficult to extend it.
10. For the Unix `passwd` file, there isn't really much saving you can do with a binary file format. Colon ':' separators are quite small, binary equivalent would not be much smaller.
    - Also need to understand the application of the passwd file. Its not read often and not really modified that often either (relatively speaking in terms of other admin operations).
11. In the case of PNG format, it is a losless graphics format and using text to store data would cause significant problems in transaction economy (long download times for large files). However, transparency was sacrificed.
    - Very thoughtful design with interoperability in mind.
    - PNG specifies byte orders, integer word lengths, endianness, and (lack of) padding between fields.
    - A PNG file consists of a sequence of chunks, each in a self-describing format beginning with the chunk type name and the chunk length. Because of this organization, PNG does not need a release number. New chunk types can be added at any time; the case of the first letter in the chunk type name informs PNG-using software whether or not each chunk can be safely ignored.
    - Also designed to easily detect file corruption.
12. In Unix, there are already established textual format designs that should be used since libraries are already written to parse them. Also, users probably recognize them already. Examples are *DSV (Delimiter Separated Values)*. On the other hand, *CSV (Comma Separated Values)* are not really used in Unix.
    - CSV has a lot more complicated parser since if comma is found in a record, the whole record needs to be escaped with quotes. And if the record has quotes already, need to escape with more quotes.
    - In DSV, only need to escape ':' with a backslash. And backslashes are escaped by another backslash.
13. RFC 822 is a textual format for Internet electronic messages. It allows for attachments using MIME (Multipurpose Internet Media Extension).
    - Record attributes are stored one per line. Thus, new records are easily added (we can see this with mail headers).
    - Used by HTTP 1.1 (and later).
    - One weakness is when putting more than one RFC 822 message in a file (e.g. Mailbox). Hard to distinguish where one starts and the other message ends.
14. XML syntax resembles HTML. It is basically a low-level syntax. Needs document type definition (DTDs) such as XHTML to give it semantics.

- Suited for complex data formats but overkill for small ones.
- Useful for complex nested/recursive structure for which RFC 822 is not useful at all.
- Down side is it can get bulky. Hard to see real data in between all the syntax. Traditional Unix tools don't play well with it so you need external and complex parsers.

15. Windows INI format is useful for two-level data organization. A section header and related records under that. If all you need is a simple key-value pair, it is better to use DSV.
16. When designing a format, include version or include the data in self-describing independent chunks.
17. Conversion of floating-point numbers from binary to text format and back can lose precision, depending on the quality of the conversion library you are using. Sometimes you have to dump the floating point field as raw binary.

The Pros and Cons of File Compression

1. Many modern Unix projects (OpenOffice, AbiWord) use XML compressed with *zip* or *gzip* as the data file format.
   - When compressing whole file, the compression tool can look at whole file for repetitive data to compress. Thus, in some cases, results in smaller file than binary data format (e.g. Microsoft Word's native format).
   - Can also change compression method in future, since it is not tied to XML.
2. Compression can limit data transparency. Some Unix tools such as file can't see past the compression header (as of 2003). And other tools might require you to uncompress data before you can, for example, *grep* through it (can use *zgrep* today I guess).
   - Can probably use straight up *gzip* compressed XML data without self-identifying structure or header provided by *zip*.
3. Idea is to think of tradeoffs from the beginning of the design on the program.

Application Protocol Design

1. All the good reasons for being textual apply to application-specific protocols as well.
2. When your application protocol is textual and easily parsed by eyeball, many good things become easier. Transaction dumps become much easier to interpret. Test loads become easier to write.
3. In the case of SMTP, command requests are simple <*command*> <*arguments*> format. While responses are <*status code*> <*message*>. It is easy to debug. SMTP is a *push* protocol where requests are initiated by the client.
4. POP3 is a *pull* protocol with transactions initiated by the mail receiver. POP3 is also textual and line oriented. Similar to SMTP in request/response formats. POP3, however, uses status tokens instead of status codes like SMTP.
5. IMAP is similar as well. However, instead of ending payload with a dot, it sends back the length in bytes first. It makes it easier on client so client knows how much buffer to allocate.
   - IMAP also adds a sequence number to each request. Thus, requests can be sent to server in bulk at once.
6. Most applications nowadays layer their special purpose protocols on top of HTTP. HTTP has become a universal application protocol.
   - Can use existing HTTP methods *GET (fetch resource)*, *PUT (modify/create resource)*, and *POST (ship data to a form or backend process)*.

- Has a RFC 822/MIME message format. Thus, can contain arbitrary messages in them.
- Also has support for authentication and extensible headers.
- Application can tunnel through native HTTP port 80 instead of a custom TCP/IP port which may need to be opened up in the firewall.
  - However, it is not good practice to use same port, especially if the application is serving data quite different from normal HTTP.
  - Thus, with a separate port, you can also easily distinguish the traffic and maybe filter it out if necessary.
  - Also note that if different port is used, a new URL scheme needs to be registered (e.g. *git://*).
- However, there is definitely a risk. When the webserver and plugins become more complicated, cracks in the code can have large security implications.
- RFC 3205 Use of of HTTP as a Substrate has good advice for using HTTP as under layer of an application protocol.
  - Be careful when re-using HTTP status codes. For example, a 200 error your application returns means success in HTTP. Thus, a proxy caching responses will send that response back to other requests. Similarly with 500 error, the proxy might respond and add a helpful message but the 500 error means something else to your application.
  - If the different codes needs to be returned, they should not be returned in the standard HTTP headers but in the body of the message.
  - A layered application which cannot operate in the presence of intermediaries or proxies that cache and/or alter error responses, should not use HTTP as a substrate.

7. *IPP (Internet Printing Protocol)* is used to control network-accessible printers.
   - Uses HTTP 1.1 as a transport layer. All IPP requests are passed via an HTTP POST method call. Responses are ordinary HTTP responses.
   - HTTP 1.1 allows persistent connections that make a multi-message protocol more efficient. Thus you can chunk the files without having to pre-scan the files to determine length of request. *Note that in this case, Transfer-Encoding header is used instead of Content-Lenght*. Thus, keep sending requests as the process scans the files.
   - Also, using HTTP redirection (301 Code), the server can tell client to redirect the submission of the job to another printer server sine this is not available.
   - Can run over TLS/SSL to encrypt messages.
   - Most network aware printers already embed a web server do display status to users. Thus, natural to use HTTP to also control printer.
   - The only drawback in that the protocol is completely driven by client requests. No way to really ship asynchronous alerts from printers back to client (can use AJAX nowadays to do polling).
   - Note that IPP uses HTTP as underlying protocol but uses different port (631) for security reasons.
8. XML-RPC, SOAP, and Jabber all use XML with MIME to structure requests and payloads.
   - XML-RPC is simple and extensible. However, currently being replaced by JSON.
   - SOAP is more heavy weight and includes arrays and C-like structs. Considered bloated by many.
   - Jabber is a peer-to-peer protocol that support instant messaging and presence. Passes around XML forms and live documents.

Chapter 6: Transparency

Contents

1. Elegant code is not only correct but visibly, transparently correct. It does not merely communicate an algorithm to a computer, but also conveys insight and assurance to the mind of a human that reads it.
2. Emacs Lisp libraries are discoverable but not transparent. Which means it is easy to modify a certain part but difficult to comprehend whole system. Linux kernel on the otherhand is very transparent but not easily discoverable since it is easy to understand its organization but hard to modify a certain piece of the code.
3. *"Discoverability is about reducing barriers to entry; transparency is about reducing the cost of living in the code"*.
4. *fetchmail -v* outputs detailed protocol exhanges (IMAP, SMTP, or POP3). And thus makes this discoverable.
5. GCC is transparent.
    - GCC is organized as a sequence of processing stages knit together by a driver program. The stages are: preprocessor, parser, code generator, assembler, and linker.
    - The first three stages are all textual and thus easy to debug.
6. Transparency in UI code is when the UI does not hide too much from the user. For example, *kmail* shows the SMTP transactions in its status bar. Thus, can easily be debugged if there are problems or failures. It is not loud thus does not bother simple users.
7. Terminfo uses the file system itself as a simple hierarchical database. This is a superb bit of constructive laziness, obeying the Rule of Economy and the Rule of Transparency. It means that all the ordinary tools for navigating, examining and modifying the file system can be used to navigate, examine, and modify the terminfo database;
8. If you want transparent code, the most effective route is simply not to layer too much abstraction over what you are manipulating with the code.
9. Static depth of a procedural-call hierarchy should really not be more than four i.e. to accomplish a certain procedure if the function you call is more than four deep, might need to re-think the design.
10. If there is a single global structure that reflects system state, make sure it is easily explorable.
11. For the reasons stated above, client authentication is rarely used with TLS. A common technique is to use TLS to authenticate the server to the client and to establish a private channel, and for the client to authenticate to the server using some other means - for example, a username and password using HTTP basic or digest authentication.
12. Some programs such as *sng* take a non-transparent format *png* and convert it back and forth to a transparent one. This is very important in debugging.
    - With *sng*, for example, you can convert *png* to text, run some scripts through it to add annotations and then convert it back.
13. If the binary object is dynamically generated or very large, then it may not be practical or possible to capture all the state with a textualizer. In that case, the equivalent task is to write a browser. This is used for visualizing databases.
14. Unix programmers learn a tendency to scrap and rebuild rather than patching grubby code.
15. One very important practice is an application of the Rule of Clarity: choosing simple algorithms. In Chapter 1 we quoted Ken Thompson: *When in doubt, use brute force*.
16. It is also important to include hacker guides. Helpful docs on helping discoverablity of the code.

Chapter 7: Multiprogramming

Contents

1. The most characteristic program-modularization technique of Unix is splitting large programs into multiple cooperating processes. This has usually been called *multiprocessing* in the Unix world, but in this book we revive the older term *multiprogramming* to avoid confusion with multiprocessor hardware implementations.
2. The Unix operating system encourages us to break our programs into simpler subprocesses, and to concentrate on the interfaces between these subprocesses. It does this in at least three fundamental ways:
   - by making process-spawning cheap;
   - by providing methods (shellouts, I/O redirection, pipes, message-passing, and sockets) that make it relatively easy for processes to communicate;
   - by encouraging the use of simple, transparent, textual data formats that can be passed through pipes and sockets.
3. While the benefit of breaking programs up into cooperating processes is a reduction in global complexity, the cost is that we have to pay more attention to the design of the protocols which are used to pass information and commands between processes. (In software systems of all kinds, bugs collect at interfaces.)
4. It is also important to have a state-machine design that is effectively deadlock free.
5. A closely related red herring is threads (that is, multiple concurrent processes sharing the same memory-address space). Threading is a performance hack.
   - The advice of this book is to not use threads until absolutely necessary.
   - Since processes are very cheap to spawn, use those.
6. Another important reason for breaking up programs into cooperating processes is for better security. Under Unix, programs that must be run by ordinary users, but must have write access to security-critical system resources, get that access through a feature called the *setuid* bit.
   - A setuid program runs not with the privileges of the user calling it, but with the privileges of the owner of the executable. This feature can be used to give restricted, program-controlled access to things like the password file that nonadministrators should not be allowed to modify directly.

Taxonomy of Unix IPC Methods

system and *popen*

1. Simplest form is to spawn another program using the *system(3)* command. This inherits user's keyboard and display and runs to completion.
   - In this case, calling program does not communicate with the called program.
   - The classic example is shelling out an editor from within a mail or news program.
   - For more complicated cases where the called programs needs to accept input and share output is to use *popen*.

- *mutt* uses the *EDITOR* environment variable when composing/replying to messages. It creates a temporary file in the filesystem and spawns the editor to use this file. It then reads this file when it sends out the mail. Thus, it uses the filesystem to communicate with its called program.
- Can use *EDITOR=emacsclient*, this is a proxy application that creates a new buffer in an already open emacs session.

Pipes, Redirection, and Filters

1. Doug McIlroy invented the *pipe* and the construct was very important through the design of Unix and its philosophy of do one thing and do it well.
   - This also inspired later forms of IPC (especially, socket abstraction used for networking).
2. Pipes depend on the convention that every program has initially available to it (at least) two I/O data streams: standard input and standard output (numeric file descriptors 0 and 1 respectively). Many programs can be written as filters, which read sequentially from standard input and write only to standard output.
3. Normally these streams are connected to the user's keyboard and display, respectively. But Unix shells universally support redirection operations (<, >) which connect these standard input and output streams to files.
4. It's important to note that all the stages in a pipeline run concurrently. Each stage waits for input on the output of the previous one, but no stage has to exit before the next can run. This property will be important later on when we look at interactive uses of pipelines, like sending the lengthy output of a command to *more(1)*.
   - Or when running *bc | espeak*.
5. **The major weakness of pipes is that they are unidirectional.** It's not possible for a pipeline component to pass control information back up the pipe other than by terminating (in which case the previous stage will get a SIGPIPE signal on the next write). Accordingly, the protocol for passing data is simply the receiver's input format.
6. There can be named pipes, where a file is opened between the two programs (one for reading and the other for writing). Largely displaced by sockets.
7. Code bloat can be avoided, for example, since utilities can use a *more* or *less* pagers instead of implementing their own pagers.

Slave Processes

1. Master uses *popen* to spawn and communicate with slave process.
2. Example is *scp(1)* calls *ssh(1)* as a slave process. intercepting enough information from ssh's output to format report as ASCII progress bar.

Peer-to-Peer Inter-Process Communication The previous sections depicted a hierarchy of communication where one program controls the other.

Tempfiles

1. Useful for simple one-off programs and simple shellscript or wrappers. Shellout to an editor is best example.
2. Drawback is it leaves garbage behind that needs to be cleaned up if process is interrupted before tempfile can be deleted.
3. Other problem is non unique filenames. Most shell scripts use *$$* which expands to PID of process in the filename (Linux kernel wraps around and re-uses old PIDs if it reaches max of 32768 */proc/sys/kernel/pid_max*).
4. There is also security risk if the tempfile name is easily predicted or known/visible attacker can modify file while process is running and thus inject own input back into program.

Signals

1. Simplest and crudest way for two processes to communicate with each other.
    - Signal handler is executed asynchronously when the signal is received.
2. Not really designed as an IPC but more of a way for OS to notify programs of certail errors and events.
    - The *SIGHUP* signal, for example, is sent to every program started from a given terminal session when that session is terminated. This is why *nohup* is used to spawn a program (ignores *SIGHUP*) and keeps running in background.
    - The SIGINT signal is sent to whatever process is currently attached to the keyboard when the user enters the currently-defined interrupt character (often control-C).
    - *SIGUSR1* and *SIGUSR2* are part of POSIX standard used for some IPC situation. A way for operator or another program to tell a daemon that it needs to either reinitialize itself, wake up to do work, or write internal-state/debugging information to a known location.
3. Technique used with signals is *pidfile*. Programs that will need to be signaled will write their PID to a file in a known location (*/var/run* for example).
    - Other programs can read that file to discover that PID. The pidfile may also function as an implicit lock file in cases where no more than one instance of the daemon should be running simultaneously.
4. SIGTERM ('terminate') is often accepted as a graceful-shutdown signal (this is as distinct from SIGKILL, which does an immediate process kill and cannot be blocked or handled). SIGTERM actions often involve cleaning up tempfiles, flushing final updates out to databases, and the like.

Sockets

1. Developed in BSD as a way to encapsulate access to data networks.
2. Two programs communicating over a socket see a bi-directional byte stream.
3. Byte streams are sequenced (single bytes will be received in the same order they were sent).
4. Byte streams are reliable (socket users are guaranteed that the underlying network will do error detection and retry to ensure delivery).
5. Socket descriptors once obtained, behave essentially like file descriptors.
6. Ken Arnold: **Sockets differ from read/write in one important case. If the bytes you send arrive, but the receiving machine fails to ACK, the sending machine's TCP/IP stack will time out. So getting an error does not necessarily mean that the bytes didn't arrive; the receiver may be using them. This problem has profound consequences for the design of reliable protocols, because you have to be able to work properly when you don't know what was received in the past. Local I/O is 'yes/no'. Socket I/O is 'yes/no/maybe'. And nothing can ensure delivery — the remote machine might have been destroyed by a comet.**
7. At the time a socket is created, you specify a protocol family which tells the network layer how the name of the socket is interpreted.
    - *AF_INET* family in which addresses are interpreted as host-address and service-number pairs.
    - *AF_UNIX* (aka *AF_LOCAL*) protocol family supports the same socket abstraction for communication between two processes on the same machine (names are interpreted as the locations of special files analogous to bidirectional named pipes). As an example, client programs and servers using the X windowing system typically use AF_LOCAL sockets to communicate.
8. To use sockets gracefully, in the Unix tradition, start by designing an application protocol for use between them — a set of requests and responses which expresses the semantics of what your programs will be communicating about in a succinct way.

9. For example in PostgresSQL: Because the front end and back end are separate, the server doesn't need to know anything except how to interpret SQL requests from a client and send SQL reports back to it. The clients, on the other hand, don't need to know anything about how the database is stored. Clients can be specialized for different needs and have different user interfaces.
10. Sockets inherently separates the address space of processes and implicitly defines a client/server or peer-to-peer model of communication.

Shared Memory

1. If your communicating processes can get access to the same physical memory, shared memory will be the fastest way to pass information between them.
2. Typically use *mmap* to map files into memory that can be shared between processes. Or can use POSIX *shm_open* API to create a file that can be shared. Basically, tells OS not to flush the pseudofile data to disk.
3. Because access to shared memory is not automatically serialized by a discipline resembling read and write calls, programs doing the sharing must handle contention and deadlock issues themselves, typically by using semaphore variables located in the shared segment.
4. X uses shared memory for performance gains to pass large images between client and server.

Problems and Methods to Avoid

Obsolete Unix IPC Methods

1. System V had IPC facilities in the form of message passing (*msgctl(2)*). This is still available in Linux.
2. Despite occasional exceptions such as NFS (Network File System) and the GNOME project, attempts to import CORBA, ASN.1, and other forms of remote-procedure-call interface have largely failed — these technologies have not been naturalized into the Unix culture.
    - Hard to query the interfaces for their capabilities.
    - Difficult to monitor them in action without building special tools.
    - Examples of bad designs outside Unix is COM/DCOM on Windows.
3. Unix tradition, on the other hand, strongly favors transparent and discoverable interfaces.
4. Today, RPC and the Unix attachment to text streams are converging in an interesting way, through protocols like XML-RPC and SOAP.

Threads — Threat or Menace?

1. Though Unix developers have long been comfortable with computation by multiple cooperating processes, they do not have a native tradition of using threads (processes that share their entire address spaces).
2. From a complexity-control point of view, threads are a bad substitute for lightweight processes with their own address spaces; the idea of threads is native to operating systems with expensive process-spawning and weak IPC facilities.
3. Threads are a fertile source of bugs because they can too easily know too much about each others' internal states.
4. There is no automatic encapsulation, as there would be between processes with separate address spaces that must do explicit IPC to communicate.
5. While threading can get rid of some of the overhead of rapidly switching process contexts, locking shared data structures so threads won't step on each other can be just as expensive.

6. Jim Gettys (Author of X): **The X server, able to execute literally millions of ops/second, is not threaded; it uses a poll/select loop. Various efforts to make a multithreaded implementation have come to no good result. The costs of locking and unlocking get too high for something as performance-sensitive as graphics servers.**
7. The upshot is that you cannot count on threaded programs to be portable.
    - Each OS has different implementations.

Process Partitioning at the Design Level

1. The first thing to notice is that tempfiles, the more interactive sort of master/slave process relationship, sockets, RPC, and all other methods of bidirectional IPC are at some level equivalent — they're all just ways for programs to exchange data during their lifetimes.
2. We've seen from the PostgreSQL study that one effective way to hold down complexity is to break an application into a client/server pair. The PostgreSQL client and server communicate through an application protocol over sockets, but very little about the design pattern would change if they used any other bidirectional IPC method.
3. If you can use limited shared memory and semaphores, asynchronous I/O using SIGIO, or poll(2)/select(2) rather than threading, do it that way. Keep it simple; use techniques earlier on this list and lower on the complexity scale in preference to later ones.

Python

Contents

Common Questions

Difference between *class A(object):* and *class A:*

Subclassing *object* yields a new-style class (in Python 3, *class A:* defaults to new style). Some differences:

1. Method Resolution Order (MRO) defined by *__mro__* attribute of class, defines how inheritance hierarchies are walked. Before, it was depth first. Now, it is more sane and is based on *__mro__*.
2. The *__new__* constructor is added. This allows class to act as factory method, rather than return new instance of class. Useful for returning particular subclasses, or reusing immutable objects rather than creating new ones without having to change the creation interface.
3. Descriptors. These are the feature behind such things as properties, classmethods, staticmethods etc. Essentially, they provide a way to control what happens when you access or set a particular attribute on a (new style) class.

```
class D(object):
    pass

class E:
    pass

dir(D)
# ['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',

dir(E)
# ['__doc__', '__module__']
```

How are arguments passed - by reference or by value?

It is actually *call by object*, *call by sharing*, or *call by object reference*, so the answer is neither.

In Python, everything is an object and all variables hold references to objects. So what's being passed are objects (references) and can only be changed if they are mutable (lists and dicts). Note that numbers, strings, and tuples are immutable.

Sum/multiply all the elements in a list

```
# basic
s = 0
for x in range(10):
    s += x

# the right way
s = sum(range(10))

# basic
s = 1
for x in range(1, 10):
    s *= x

# the other way
from operator import mul
s = reduce(mul, range(1,10))
```

This brings up the discussion of functional programming concepts in python. These functions can be used in conjunction with *lambda* instead of longer for loops.

map(function, seq)

Takes a function and applies it to each item in the sequence. The resulting object is an iterable object. Thus, apply *list()* to the map object to get a list output. Or loop through it.

```
a = map(lambda x: x*2, range(0,10))
for i in a:
    print(i)
```

filter(function, seq)

Filter extracts elements in the sequence that return *True*. Note that function can be *None*, and thus it will return items that are *True*.

```
a = filter(lambda x: x > 1, range(0,10))
list(a)
```

reduce(function, seq)

Reduce applies a function of *two* arguments, cumulatively to the items of a sequence. It returns one value back (the cumulative value).

```
from functools import reduce
a = reduce(lambda x, y: x * y, [1,2,3,4])
a == 24 # True
```

Note that that function takes two arguments. The sequence of operations goes as follows *(((1*2) * 3) * 4)*.

Difference between tuples and list

Lists are mutable while tuples are not. More importantly, tuples can be *hashed* (used as keys for dictionaries). Tuples are used if order of elements in a sequence matters (e.g. coordinates, points of a path, etc).

```
t = ((1,'a'), (2,'b'))
dict(t)
# OUT: {1: 'a', 2: 'b'}

dict((y,x) for x,y in t)
# OUT: {'b': 2, 'a': 1}

{y:x for x,y in t}
# OUT: {'b': 2, 'a': 1}
```

What are decorators and what is their usage?

Decorators allow you to inject or modify code in functions or clases. Basically, a wrapper to an existing function. Thus, allows you to execute a code before or after the original code. For example, logging a function.

```
from __future__ import print_function

def log(fn):
    def wrapper(*args, **kw):
        res = fn(*args, **kw)
        print("%s(%r) -> %s" % (fn.__name__, args, res))
        return res
    return wrapper

@log
def ispal(word):
    if len(word) < 2:
        return True
    return (word[0] == word[-1]) & ispal(word[1:-1])


ispal("test")
ispal("kayak")
```

Common Mistakes

Misusing expressions as defaults for function arguments

```
def foo(bar=[]):
    bar.append("paz")
    return bar
```

1. Expect to return *paz* everytime *foo()* is called. But this is not the case.
2. After calling *foo()* three times, you will get *["baz", "baz", "baz"]*
3. This is because, the the default value for a function argument is only evaluated once, at the time that the function is defined.
4. To get around it:

```
def foo(bar=None):
    if bar == None:
        bar = []
    bar.append("paz")
    return bar
```

Project Management

Contents

Agile

Notes from: http://martinfowler.com/articles/newMethodology.html

1. Agile is a compromise between no process and too much process, providing just enough process to gain a reasonable payoff.
2. *Agile methods are adaptive rather than predictive.* Engineering methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.
3. *Agile methods are people-oriented rather than process-oriented.* The goal of engineering methods is to define a process that will work well whoever happens to be using it. Agile methods assert that no process will ever make up the skill of the development team, so the role of a process is to support the development team in their work.

Predictive vs Adaptive

1. The usual inspiration for methodologies comes from engineering disciplines such as civil or mechanical engineering. Such disciplines put a lot of emphasis on planning before you build.
2. So what we see here are two fundamentally different activities. Design which is difficult to predict and requires expensive and creative people, and construction which is easier to predict. Once we have the design, we can plan the construction.
3. So the approach for software engineering methodologies looks like this: we want a predictable schedule that can use people with lower skills. To do this we must separate design from construction.
4. This is where design notations such as UML come into play. If we can make all the significant decisions using the UML, we can build a construction plan and then hand these designs off to coders as a construction activity.
5. For Engineering designs, mathematical analysis can show if a design will work. However, for UML it is only through peer-review that design can be checked.
6. For a bridge, cost of design is only about 10% of job, rest is construction. For code it is the opposite, apparently only 15% of project is code and unit tests.
7. Thus, source code is the design document and construction is the compiler/linker. Thus, the construction phase is cheap and and should be automated.
8. Conclusions:
    - In software: construction is so cheap as to be free
    - In software all the effort is design, and thus requires creative and talented people (source code)
    - Creative processes are not easily planned, and so predictability may well be an impossible target.
    - We should be very wary of the traditional engineering metaphor for building software. It's a different kind of activity and requires a different process.

The Unpredictability of Requirements

1. Developers shouldn't be surprised that the requirements are always changing. This is the norm in software.
2. One problem with this is that just trying to understand the options for requirements is tough.
3. Estimation is hard for many reasons. Part of it is that software development is a design activity, and thus hard to plan and cost. Part of it is that the basic materials keep changing rapidly. Part of it is that so much depends on which individual people are involved, and individuals are hard to predict and quantify.

4. It's very difficult to see what value a software feature has until you use it for real. Only when you use an early version of some software do you really begin to understand what features are valuable and what parts are not.
5. In today's economy the fundamental business forces are changing the value of software features too rapidly.
6. Predictability is not impossible. NASA's space shuttle software group are prime example of predictability. But they have huge resources, lots of time, and very stable requirements. But most software projects are not similar to this.
7. However letting go of predictability doesn't mean you have to revert to uncontrollable chaos. Instead you need a process that can give you control over an unpredictability. That's what adaptivity is all about.

Controlling an Unpredictable Process - Iterations

1. To get some form of predictability, we need an accurate way of knowing where we are at this point. This is where feedback mechanisms and feedback loops come into play. We need to learn about ourselves (the software, team, etc).
2. This is what iterative development is all about.
3. The key to iterative development is to frequently produce working versions of the final system that have a subset of the required features. These working systems are short on functionality, but should otherwise be faithful to the demands of the final system. They should be fully integrated and as carefully tested as a final delivery.
4. The point of this is that there is nothing like a tested, integrated system for bringing a forceful dose of reality into any project. Documents can hide all sorts of flaws. Untested code can hide plenty of flaws. But when people actually sit in front of a system and work with it, then flaws become truly apparent: both in terms of bugs and in terms of misunderstood requirements.
5. Iterative development makes sense in predictable processes as well. But it is essential in adaptive processes because an adaptive process needs to be able to deal with changes in required features.
6. This leads to a style of planning where long term plans are very fluid, and the only stable plans are short term plans that are made for a single iteration. Iterative development gives you a firm foundation in each iteration that you can base your later plans around.
7. A key question for this is how long an iteration should be. Different people give different answers. XP suggests iterations of one or two weeks. SCRUM suggests a length of a month. Crystal may stretch further. The tendency, however, is to make each iteration as short as you can get away with. This provides more frequent feedback, so you know where you are more often.

The Adaptive Customer

1. This kind of adaptive process requires a different kind of relationship with a customer than the ones that are often considered.
2. A fixed price contract requires stable requirements and hence a predictive process. Adaptive processes and unstable requirements imply you cannot work with the usual notion of fixed-price.
3. After all the customer wouldn't be wanting some software unless their business needed it. If they don't get it their business suffers. So even if they pay the development company nothing, they still lose. Indeed they lose more than they would pay for the software (why would they pay for the software if the business value of that software were less?)
4. This doesn't mean that you can't fix a budget for software up-front. What it does mean is that you cannot fix time, price and scope. **The usual agile approach is to fix time and price, and to allow the scope to vary in a controlled manner.**

5. In an adaptive process the customer has much finer-grained control over the software development process. At every iteration they get both to check progress and to alter the direction of the software development. This leads to much closer relationship with the software developers, a true business partnership.
6. All this yields a number of advantages for the customer. For a start they get much more responsive software development. A usable, although minimal, system can go into production early on. The customer can then change its capabilities according to changes in the business, and also from learning from how the system is used in reality.
7. Every bit as important as this is greater visibility into the true state of the project.
8. If bad news is lurking it tends to come earlier, when there is still time to do something about it. Indeed this risk control is a key advantage of iterative development.
9. Mary Poppendieck summed up this difference in viewpoint best for me with her phrase *"A late change in requirements is a competitive advantage"*. Often the most valuable features aren't at all obvious until customer have had a chance to play with the software. Agile methods seek to take advantage of this, encouraging business people to learn about their needs as the system gets built, and to build the system in such a way that changes can be incorporated quickly.
10. All this has an important bearing what constitutes a successful project. A predictive project is often measured by how well it met its plan. A project that's on-time and on-cost is considered to be a success.
11. This measurement is nonsense to an agile environment. For agilists the question is business value - did the customer get software that's more valuable to them than the cost put into it.
12. A good predictive project will go according to plan, a good agile project will build something different and better than the original plan foresaw.

Putting People First

1. Executing an adaptive process is not easy. In particular it requires a very effective team of developers. The team needs to be effective both in the quality of the individuals, and in the way the team blends together.

Plug-Compatible Programming Units

1. One of the aims of traditional methodologies is to develop a process where the people involved are replaceable parts. With such a process you can treat people as resources who are available in various types. You have an analyst, some coders, some testers, a manager. The individuals aren't so important, only the roles are important.
2. But this raises a key question: are the people involved in software development replaceable parts? One of the key features of agile methods is that they reject this assumption.
3. Furthermore his (Alastair Cockburn) studies of software projects have led him to conclude the people are the most important factor in software development.
4. "People" are highly variable and non-linear, with unique success and failure modes. Those factors are first-order, not negligible factors.
5. This creates a strong positive feedback effect. If you expect all your developers to be plug-compatible programming units, you don't try to treat them as individuals. This lowers morale (and productivity).
6. The notion of people as resources is deeply ingrained in business thinking, its roots going back to the impact of Frederick Taylor's Scientific Management approach. In running a factory, this Taylorist approach may make sense. But for the highly creative and professional work, which I believe software development to be, this does not hold.

Programmers are Responsible Professionals

1. A key part of the Taylorist notion is that the people doing the work are not the people who can best figure out how best to do that work.
2. Recent history increasingly shows us how untrue this is for software development. Increasingly bright and capable people are attracted to software development, attracted by both its glitz and by potentially large rewards.
3. When you want to hire and retain good people, you have to recognize that they are competent professionals. As such they are the best people to decide how to conduct their technical work.

Managing a People Oriented Process

1. It is about accepting the process not being forced to follow the process by management. Thus, requires active involvement of team.
2. Another point is that the developers must be able to make all technical decisions. XP gets to the heart of this where in its planning process it states that only developers may make estimates on how much time it will take to do some work.
3. Such an approach requires a sharing of responsibility where developers and management have an equal place in the leadership of the project. Notice that I say equal. Management still plays a role, but recognizes the expertise of developers.
4. An important reason for this is the rate of change of technology in our industry. After a few years technical knowledge becomes obsolete. This half life of technical skills is without parallel in any other industry. Even technical people have to recognize that entering management means their technical skills will wither rapidly. Ex-developers need to recognize that their technical skills will rapidly disappear and they need to trust and rely on current developers.

The Difficulty of Measurement

1. Despite our best efforts we are unable to measure the most simple things about software, such as productivity. Without good measures for these things, any kind of external control is doomed.
2. The point of all this is that traditional methods have operated under the assumption that measurement-based management is the most efficient way of managing. The agile community recognizes that the characteristics of software development are such that measurement based management leads to very high levels of measurement dysfunction. It's actually more efficient to use a delegatory style of management, which is the kind of approach that is at the center of the agilist viewpoint.

The Role of Business Leadership

1. This leads to another important aspect of adaptive processes: they (developers) need very close contact with business expertise.
2. This goes beyond most projects involvement of the business role. Agile teams cannot exist with occasional communication . They need continuous access to business expertise. Furthermore this access is not something that is handled at a management level, it is something that is present for every developer.
3. A large part of this, of course, is due to the nature of adaptive development. Since the whole premise of adaptive development is that things change quickly, you need constant contact to advise everybody of the changes.

The Self-Adaptive Process

1. However there's another angle to adaptivity: that of the process changing over time. A project that begins using an adaptive process won't have the same process a year later. Over time, the team will find what works for them,

and alter the process to fit.

2. The first part of self-adaptivity is regular reviews of the process. Usually you do these with every iteration. At the end of each iteration, have a short meeting and ask yourself the following questions (culled from Norm Kerth)
    - What did we do well?
    - What have we learned?
    - What can we do better?
    - What puzzles us?
3. While both published processes and the experience of other projects can act as an inspiration and a baseline, the developers professional responsibility is to adapt the process to the task at hand.

Flavors of Agile Development

Agile Manifesto

1. Started in 2001 where bunch of people met and came up with *Manifestor for Agile Development*.
2. There were other groups coming with similar approaches to iterative development. No common name for all these approaches but *lightweight* was being used a lot.
3. Decision was to use *agile* as the umbrella name.
4. No formal organization but there is an *Agile Alliance*. This group is a non-profit group intended to promote and research agile methods. Amongst other things it sponsors an annual conference in the US.

XP (Extreme Programming)

1. Got the lion's share of attention early in the agile movement.
2. The roots of XP lie in the Smalltalk community, and in particular the close collaboration of Kent Beck and Ward Cunningham in the late 1980's. Both of them refined their practices on numerous projects during the early 90's, extending their ideas of a software development approach that was both adaptive and people-oriented.
3. XP begins with five values (Communication, Feedback, Simplicity, Courage, and Respect). It then elaborates these into fourteen principles and again into twenty-four practices. The idea is that practices are concrete things that a team can do day-to-day, while values are the fundamental knowledge and understanding that underpins the approach.
4. XP has strong emphasis on testing. XP puts testing at the foundation of development, with every programmer writing tests as they write their production code. The tests are integrated into a continuous integration and build process which yields a highly stable platform for future development. XP's approach here, often described under the heading of Test Driven Development (TDD) has been influential even in places that haven't adopted much else of XP.

Scrum *Scrum (n): A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.*

1. Scrum also developed in the 80's and 90's primarily with OO development circles as a highly iterative development methodology. It's most well known developers were Ken Schwaber, Jeff Sutherland, and Mike Beedle.
2. Scrum concentrates on the management aspects of software development, dividing development into thirty day iterations (called 'sprints') and applying closer monitoring and control with daily scrum meetings. It places much less emphasis on engineering practices and **many people combine its project management approach**

with extreme programming's engineering practices. (XP's management practices aren't really very different.)

3. Asserts that knowledge comes from experience and making decisions based on what is known. Scrum employs an iterative, incremental approach to optimize predictability and control risk.
4. Scrum Events:
    - All events are timeboxed for duration.
    - A formal opportunity to inspect and adapt something.
    - Sprint is heart of scrum. Timeboxed one month or less (if *sprint goal* is accomplished early).
        - Sprint Planning
            * Timeboxed to a max of eight hours. Answers what can be delivered at the end of sprint? And how this work will be achieved.
            * Dev team discusses forecasts of functionality that will be implemented. Product Owner discusses objective of sprint.
            * Performance of past sprints is also reviewed. Number of items selected from backlog is solely up to dev team.
            * Sprint Goal is decided here. It is basically an objective to work towards for that sprint. e.g. "Has a basic UI for users to work with", "Can pass packets and has networking functionality"
            * Sprint Backlog is subset of Product Backlog. This includes more detailed planning.
            * By the end of the Sprint Planning, the Development Team should be able to explain to the Product Owner and Scrum Master how it intends to work as a self-organizing team to accomplish the Sprint Goal and create the anticipated Increment.
        - Daily Scrum
            * 15-minute timeboxed event to plan the next 24 hours that only dev team can participate in.
            * Members explain: What did I do to help Dev team meet Sprint Goal? What will I do today to help Dev team meet the Sprint Goal? Do I see any obstacles that prevents me from a accomplishing that goal?
        - Sprint Review
            * Timeboxed to 4 hours and held at end of Sprint to inspect *Increment* and adapt Product Backlog if needed.
            * Includes Scrum Master, Product Owner, Dev Team and other key stake holders.
            * Demonstration of work happens and discussion of obstacles, product backlog, etc.
            * Review of market, how it has changed, most business value on what to do next.
            * Review timeline and budget as well.
            * Backlog can be re-prioritized.
        - Sprint Retrospective
            * Opportunity for entire Scrum Team to inspect itself and create a plan for improvements during next sprint.
            * Occurs after sprint review and before next sprint planning. Time boxed to three hours.
    - Sprint maybe cancelled only by Product Owner but are very uncommong and not recommended.
5. The Scrum Team consists of a Product Owner, the Development Team, and a Scrum Master.

6. Scrum Teams deliver products iteratively and incrementally, maximizing opportunities for feedback.
7. *Product Owner* is sole person managing Product backlog.
8. *Development Teams* are structured and empowered by the organization to organize and manage their own work. The resulting synergy optimizes the Development Team's overall efficiency and effectiveness. Every member is known as a *Developer*. Accountability is to whole team. Usually between *3-9* members.
9. The *Scrum Master* is responsible for ensuring Scrum is understood and enacted. Scrum Masters do this by ensuring that the Scrum Team adheres to Scrum theory, practices, and rules.
    - Finding techniques for effective Product Backlog management (helping Product Owner).
    - Acts as a coack to development team. Removes impediments to the progress of Dev Team.
    - Helping employees and stakeholders understand and enact Scrum and empirical product development.
10. A Product Backlog is never complete. The earliest development of it only lays out the initially known and best-understood requirements. The Product Backlog evolves as the product and the environment in which it will be used evolves. The Product Backlog is dynamic; it constantly changes to identify what the product needs to be appropriate, competitive, and useful. As long as a product exists, its Product Backlog also exists.
11. Product Backlog can span multiple Scrum Teams and thus can apply grouping techniques to the backlog.
12. The Sprint Backlog is a highly visible, real-time picture of the work that the Development Team plans to accomplish during the Sprint, and it belongs solely to the Development Team.

Crystal

1. Different variations for different sized teams.
2. Despite their variations all crystal approaches share common features. All crystal methods have three priorities: safety (in project outcome), efficiency, habitability (developers can live with crystal). They also share common properties, of which the most important three are: Frequent Delivery, Reflective Improvement, and Close Communication.

Should you go agile?

1. In today's environment, the most common methodology is code and fix. Applying more discipline than chaos will almost certainly help, and the agile approach has the advantage that it is much less of a step than using a heavyweight method.
2. Simpler processes are more likely to be followed when you are used to no process at all.
3. The first step is to find suitable projects to try agile methods out with. Since agile methods are so fundamentally people-oriented, it's essential that you start with a team that wants to try and work in an agile way.
4. So where should you not use an agile method? I think it primarily comes down to the people.

Others

User Stories

Notes taken from: http://www.mountaingoatsoftware.com/agile/user-stories

1. User stories are short, simple description of a feature told from the perspective of the person who desires the new capability. This is usually a user or customer of the system. Template:

    As a <type of user>, I want <some goal> so that <some reason>.

2. Often written on index cards, sticky notes and placed in a shoe box, arranged on walls, tables, to facilitate planning and discussion.
3. Focus shifts from writing about features to discussing them.
4. User stories can be written at varying levels of detail. Thus, user stories can be written to cover large amounts of functionality. These are generally known as *epics*. An example:

   As a user, I can backup my entire drive

5. Epics are generally too large to complete in one agile iteration. It is split into smaller user stories. The above epic can be split into dozens (or hundreds) of user stories:

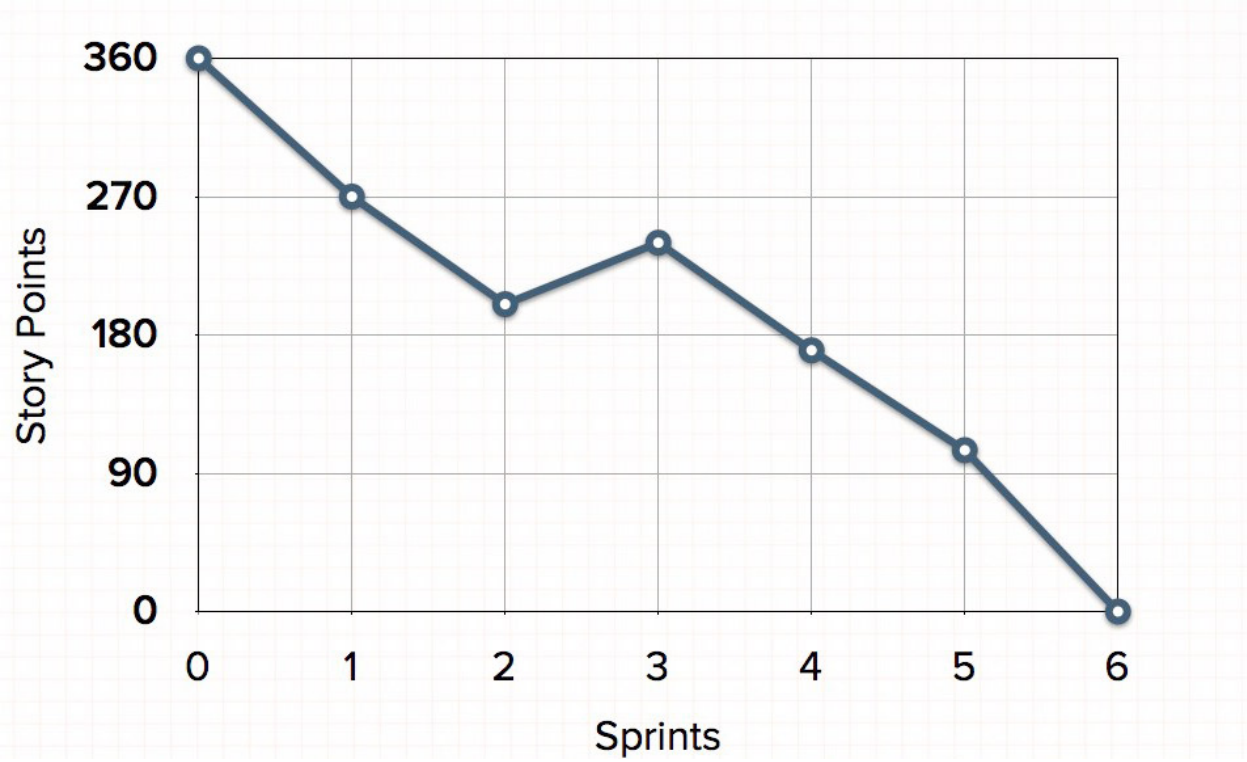
   As a power user, I can specify files or folders to backup based on file size, date created and date modified.

   As a user, I can indicate folders not to backup so that my backup drive isn't filled up with things I don't need saved.

6. Note that details can be added to user stories. These can be accomplished by splitting user stories into smaller user stories. Or by adding *conditions of satisfaction*.
7. *Conditions of satisfaction* are like high-level acceptance tests. For the example above:
   - *Make sure data is verified during copy.*
   - *Make sure there is a report generated of the backup.*
8. Note that user stories are usually written by product owner. However, during breakdowns of user stories, each team member can write it as well.
9. The important fact is that it doesn't matter who writes the user stories. It is more important to have everyone involved in the discussion of it.
10. User Stories are usually the main composition of a product backlog. Re-prioritization happens often and user stories can be added/removed throughout the agile development process.
11. Note that *Fibonnaci* sequence is used to estimate story points. The idea is, the larger the story is, the more uncertainty there is around it and the estimate is less accurate. Thus, total number of points give a number on complexity of project.
    - 0, 1 means user story doesn't take anytime at all
    - Bigger than 13 means very complex and probably needs to be broken up.

Burndown Charts

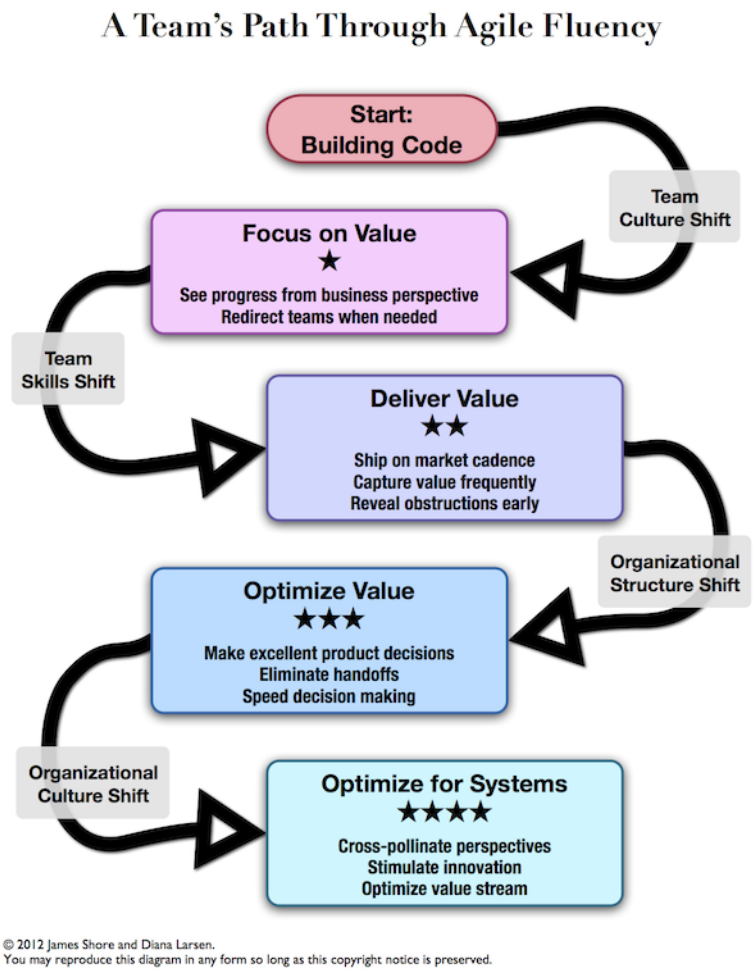
1. A burn down chart is a graphical representation of work left to do versus time.
2. The outstanding work (or backlog) is often on the vertical axis, with time along the horizontal.
3. It is useful for predicting when all of the work will be completed.



Agile Fluency

Reference: http://martinfowler.com/articles/agileFluency.html

This diagram explains stages agile teams go through as they gain more experience. This shows successful team progression. Note that *fluency* here means how a team develops software when it's under pressure.



Most teams are at one-star level. Number of teams with more stars are fewer as there are factors such as organizational culture, technical debt of code, etc.

Fluency is more about *habit* than skills and thus requires a lot of practice.

It's best to choose the level of fluency you want to achieve and to practice everything needed for that level from the beginning.

In other words, if your goal is to have a three-star team, use a three-star approach from the start. Although your team will still progress one level at a time, practicing all the techniques together will allow them to advance more quickly.

Important for organization to support team star goals and for team members to stick together.

1. Scrum is frequently used by one-star teams as their core goal/metric is business value of customer/stakeholders. The focus is on creating value. The idea is organization can realize its investment quickly (2-6 months) and have greater insight into team progress.
2. Two-star teams deliver on market cadence (shipping as often as market will accept it). Usually use XP combined with Scrum project management. Includes continuous integration, test drive development, test driven development, pair programming, and collective ownership.
    - Consistently and predictably deliver value.
    - Includes metrics used by one-star team to report business value. But core metric is to deliver low-defect product and ability to ship on market cadence.

- Takes significant investment in time as team needs a lot of skill and practice to consistently deliver these kinds of products.

3. Three-star teams deliver the most value possible for your investment. They understand what the market wants, what your business needs, and how to meet those needs.
    - However, Lean Startup is an example of a method that operates at the three-star level. It's most applicable to new product development. The ideas from Lean Software Development (no relation to Lean Startup) are also useful. Agile chartering, embedded product management teams, customer discovery, and adaptive planning are all examples of techniques used by three-star teams.
    - More formal business value reports are presented. There is a mutual trust between team and organization. Need to incorporate business experts full time in the team.
    - Takes several years to develop because it takes time to develop this level of trust between organization and team.
4. Four-star teams contribute to enterprise-wide success. Team members understand organizational priorities and business direction. Four-star teams will sacrifice their own needs to support the needs of a product more critical to business success. They work with other teams and with managers to optimize the overall value stream.
    - The teams we know that are striving for, and in some cases reaching, four-star fluency are at the "bleeding edge" of Agile practice. They adapt ideas from advanced management theories and innovative product development methods. Techniques include Agile portfolio management, systems thinking, value stream analysis, whole system planning, intact teams, open book management, and radical self-organization.
    - The core metric for four-star teams is whether the team shows understanding of the overall system and reports how its actions affect the enterprise.
    - To date, we've most often seen four-star fluency in single-team startups, where it's not much different from three-star fluency. It seems to be easiest to approach four-star fluency in organizations where trust is high, communication overhead is low, and business information is widely shared.

Managing Deadlines

Dealing with missed deadlines

1. As a project manager, this is where you earn your value. You have the opportunity to turn the project around.
2. Can perform Monte Carlo simulation to see the impact of business value if project is killed or if it is continued but delayed.
3. Identify the problems at this point and possibly *Reboot* project.
4. Also, it should never get to this point. Idea with agile is to constantly adjust scope if budget and deadline is fixed. Of course, this is with customer input.

Estimates/Deadlines

1. *Estimates* come from team and *Deadlines* come from stakeholders.
2. Sometimes deadlines can't be pushed, other factors are involved (media promotions, legal regulations, etc.).
3. A great PM is great not because they achieve project objectives all the time but because they communicate bad news early and often so that project owners can make decisions before they are surprised.

Adding People

1. Adding the right people can speed the project up, especially if they have specific domain expertise that is missing or weak in the original team,and they are being brought in to support the team rather than taking over and destroying morale.

Programming

Contents

Character Encodings For Modern Programmers

Reference

1. The story of Unicode really starts with ASCII.
2. ASCII is important because it basically defined the subset of punctuation marks and symbols that would be used in every programming language and operating system that followed.
3. ASCII is a 7-bit encoding
4. There were national variants of ASCII where certain symbols where replaced with their national variants. For example, # was replaced with £. This causes a lot of problems with programming and was eventually abandoned. **Note to enter unicode characters using *urxvt* hold *Ctrl+Shift* and type the hex code. Finally, release *Ctrl+Shift*. For example, for *£* type *Ctrl+Shift+a3***
5. ASCII 8-bit encodings add extra 128 code points and thus adds support for national variants.

UNIX, Terminals, and C1

1. UNIX is a terminal-based system (which means that it is usually operated by entering commands on a terminal).
2. UNIX used to be controlled by a dumb terminal. It is a physical equipment with a 80x25 screen and a keyboard that connects to the UNIX server by a serial line.
3. If you send text to the serial port of the terminal, it is the terminal that decides which actual character to print on the screen. Similarly, when you press a key on the keyboard, it is the terminal that decides which code to send to the UNIX server.
4. On top of this, this same communication channel needs to be used for sending control sequences, such as for moving the cursor, clearing the screen, and these sequences are all sent in-line intermixed with the character codes. This means that we can't assign characters to every code point. We need control codes, and these control codes cannot overlap the characters in the character encoding.

5. Note that *printable* ASCII range is from 32 to 126. Codes 128 and above, however, are a free-for-all. Terminals can use them as control codes, printable characters, or basically anything and it doesn't matter to the UNIX server.
6. Block of control codes 0 to 31 (known as the C0 block) as well as code 127 in ASCII are already allocated to control codes.
7. For reasons of consistency and interoperability, a recommendation was developed that codes 128 to 159 should be reserved for a second control block (called the C1 block), and only codes 160 to 255 be used for printable characters, and this formed the basis for most pre-Unicode UNIX encodings.
8. However, it is not a hard and fast rule, and UNIX itself does not treat the C1 block any differently from the rest of the upper 128 codes.
9. This makes UNIX basically language agnostic. Let's say we have a UNIX system and we have two files with names consisting of the character codes 68 196 and 68 228. When we connect a Greek language terminal to the system and list the files, we will see "DΔ" and "Dδ". When we connect a French terminal to the system, we will see "DÄ" and "Dä", and when we connect a Thai terminal we will see "D" and "D". The fact that the three different terminals show 3 different things doesn't matter. A Thai user can still open one of the files by typing "vi D" on their keyboard, the same as the Greek can by typing "vi DΔ". As far as the OS is concerned, as long as the underlying codes are the same, everything works fine. We might note here that "δ" is the lower case letter for "Δ" in Greek, whereas "" and "" are completely different unrelated letters in Thai. This doesn't matter to UNIX because the OS doesn't try to do anything fancy like case-insensitive file names. The same applies to file content. If you print a file to the screen, the codes gets passed directly to the terminal, and it is the terminal that does the rendering. The OS doesn't need to get involved.
10. Thus UNIX itself does not need an "encoding" setting. It simply doesn't care.
11. However, any command or system service that is going to output human-readable error messages needs to know which language to output. Similarly, any add-on programs, particularly those that might perform some kind of collation or text processing, want to know what language they should use. But again, this does not need to be a system-wide setting, and only need apply to a particular user's session. The system thus uses an environment variable (LOCALE) which specifies both the language and character encoding.

DOS/Windows

1. DOS and Windows does not use terminals and thus screen is addressed directly through the video adapter.
2. This means that it does not need any control codes. So DOS/Windows assigns printable characters to all of the ASCII control codes (0 to 31 and 127) and all codes above 128.
3. DOS/Windows is still considered ASCII based since maintains all of ASCII printable characters (32 to 126).
4. This obviously is a major problem as line feeds get messed up in text files. And there is a problem with printing to ASCII serial printers which uses ASCII control block for printer sequences.
5. Another problem with Windows is that it uses case-insensitive file names and thus the OS needs to know the encoding (whether Greek, Thai, etc). Thus, Windows requires a system wide encoding.
6. It should come as no surprise then that Microsoft was one of the big backers of Unicode and produced one of the first Unicode-based OSes.

Programming With 8-Bit Encodings

1. In most cases you don't need to write encoding-aware software. For example, the C compiler does not need encoding-awareness.

2. In regular expression, all characters are byte-based and thus user needs to only input correct expressions for their language (e.g. a Norwegian person wanting to match all upper case letters is going to have to use the regexp /[A-ZÅÆ]/ instead of /[A-Z]/, but the actual regular expression parser does not need to be changed).
3. The basic assumption is that we have the same encoding end-to-end, so tools don't have to do any conversion, they just spit out whatever input they get in.
4. If you want language-dependent operations, then use <*locale.h*> and <*ctype.h*>. You will need to perform *setlocale* and then do *isupper*, *islower*, etc.

Multibyte Encodings

1. Multibyte encodings are needed for East Asian languages of China, Japan, and Korea (CJK). ASCII is not enough to hold all possible combinations.
2. Other encodings based on ASCII, use multi-bytes to represent a single character.
3. The basic idea is that all non-ASCII letters are encoded as multi-byte sequences with all bytes in the 128 and higher code range. This means that we can suddenly start naming files in Japanese and Chinese without making any changes to the underlying OS.
4. However, the following will not work `char mychar = '';` since this will appear to the compiler as two characters. Also, regexes to do not work as expected. Finally, outputting to fixed-width terminal or printer will not work as expected since these characters now take up two bytes (thus two spaces) to display one character. The exception here is *Shift JIS*. Single byte characters in Shift JIS take up single character cell while double-byte characters always take two character cells.
5. However, with *Shift JIS*, they use the code 92 (backslash) as part of second byte. Thus, `printf("");` actually looks like `printf("X\");` to the compiler. There are 42 characters that use backslash in *Shift JIS*.
6. Prior to Unicode, the C and C++ standard libraries do not have any functions for handling CJK encodings. *Note that you can use ‘‘mblen()‘‘ to see how many bytes a character takes*.

Pre-Unicode Summary

1. Firstly, just about every application runs off the idea of a single encoding end-to-end. On UNIX, in particular, many protocols are encoding agnostic.
2. FTP is a great example. It has no concept of encoding or any way of specifying encodings. Whatever raw character data it receives it sends on through untouched. It's up to the sys admin to make sure everyone using it sticks to the same encoding.

Unicode

1. The very first thing we need to understand is that UTF-8 did not exist and was not envisioned when Unicode first came into use, and Unicode referred basically to what we now call UTF-16.
2. Windows NT was the first major OS based on Unicode, and was coupled with NTFS, the first major filesystem to support Unicode.
3. However, NT still had the concept of a default non-Unicode encoding, the same as DOS and Windows 95.
4. This non-Unicode encoding is called the ANSI code page on Windows. The Win32 API thus comes with two complete sets of API, the ANSI API which accept string arguments of type `char*` encoded in the default non-Unicode encoding, and the Unicode (or wide char) API which accept string arguments of type `wchar_t*`.

5. On UNIX, things were slightly different, and this is reflected in the C/C++ standard libraries. On UNIX, the expectation was that people would continue to use non-Unicode encodings in conjunction with the LOCALE environment variable, but programmers would be given the option of an automatic conversion to Unicode mode of file operation, that would allow them to code using Unicode while the underlying OS and file content would continue to use non-Unicode encodings.
6. When we open a file in C (using `fopen()`), the file does not have an orientation. If we call a non-Unicode file function such as `fgets()`, the file gets set to non-Unicode orientation, and we simply get passed the data from the file.
7. However, if we call a Unicode file function such as fgetws(), the file is put into Unicode orientation. This does not mean that the file on the disk is treated as containing Unicode. Instead, the C standard library assumes that the file is encoded in the non-Unicode encoding as specified by LOCALE and performs implicit automatic conversion of the file content from that non-Unicode encoding into Unicode in the form of `wchar_t*`.
8. On Windows NT, however, we have a problem. The C standard library still assumes that files are saved in a non-Unicode filesystem. There is no support for Unicode filenames. There is no variant of the `fopen()` function which accepts a `wchar_t*` for the filename.
9. UNIX chose 32-bit Unicode standard (ISO based) while Windows went with 16-bit universal encoding (Unicode organization). Thus, eventually we ended up with a 32-bit Unicode standard with UTF-16 as a variable-length 16-bit encoding.
10. Around the sametime UTF-8 was being developed, which was another variable length encoding.
11. Once UTF-8 was released, UNIX had a clear path towards total Unicode compatibility. Simply adopt UTF-8 as your standard encoding and all your problems go away.
12. UTF-8 is the basic standard encoding used in Linux and it's derivatives. Since UTF-8 can be encoded in a char*, it even works fine with the broken C standard libraries. UTF-8 plugs into the C standard library on what was meant to be the non-Unicode side since it is stored in char* and specified as an encoding using LOCALE. However, Linux/Windows interoperation is worse than ever.

Basic Programming With Unicode

1.