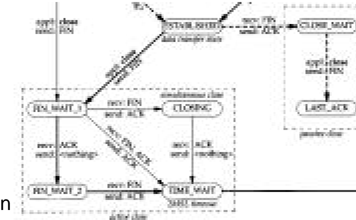
CS290i – Lecture 5
Apache architecture

Scalable Internet Services and Systems, Winter 2002

Thorsten von Eicken
Department of Computer Science
University of California at Santa Barbara

Closing HTTP1.1 sockets

- **HTTP 1.1 close**
  - Pipelined connections require independent close
  - Server sends "connection: close" and closes output
  - Client reads, and closes too
- **FIN_WAIT2 timeout**
  - TCP spec has none
    - Many OSes had none…
    - Correct while socket open, but not when it's closed
  - HTTP1.1, persistent connection timeout
    - Server closes socket, but client never notices
      - E.g. client waits for user input, modem hung-up, … never checks currently open sockets
    - OS ends up with sockets in FIN_WAIT2

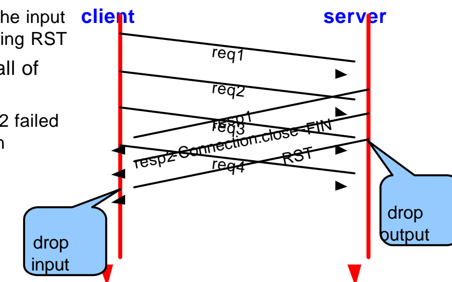


HTTP 1.1 close (2)

- **SO_LINGER socket option**
  - When socket output closed, what to do with untransmitted data?
  - Windows (winsock2):
    - SO_DONTLINGER: close doesn't block, data sent in background
    - SO_LINGER(0): close doesn't block, data dropped, RST sent
    - SO_LINGER(>0): close blocks, after timeout, "connection terminated" (I.e. RST sent???)
  - Solaris 8:
    - Seems to be the same as winsock
    - But man page is even less clear…
- **HTTP 1.1**
  - In principle: use SO_LINGER(your_best_guess)
  - In practice: ouch!

HTTP 1.1 close (3)

- **Why not close input too?**
  - Or: what happens if SO_LINGER is implemented incorrectly?
  - Server closes input&output
  - Closing input sends RST
    - Many OSes flush the input buffer when receiving RST
  - Client may not see all of response 2
    - And think request 2 failed for no good reason



Server operation

- **Server loop**
  - Bind `int bind(int s, struct sockaddr *name, int namelen);`
  - Listen `int listen(int s, int backlog);`
  - Forever
    - Accept `int accept(int s, struct sockaddr *addr, socklen_t *addrlen);`
    - Read
    - Write
    - Close
- **Inetd**
  - Performs bind, listen, accept
  - Forks & execs executable per /etc/inetd.conf
    - `ftp stream tcp6 nowait root /usr/sbin/in.ftpd in.ftpd`
    - `telnet stream tcp6 nowait root /usr/sbin/in.telnetd in.telnetd`

Server architectures

- **Single threaded**
- **Single-threaded non-blocking I/O driven**
- **Single-threaded event driven**
- **Process per request**
- **Thread per request**
- **Process/thread worker pool**

Issues:

- **Performance**
- **Resources per request**
  - Cpu, memory, file desc, …
- **Parallelism, synchronization**
  - Parallelism of cpu & I/O
- **Plug-in architecture**
- **Robustness**
- **Implementation complexity**

Single-thr, non-block I/O

Issues:
- Performance
- Resources
- Parallelism
- Synchronization
- Plug-ins
- Robustness

- **Loop:**
  - Select on accept
    - Accept connection
  - Select on read (all socks with expected input)
    - Read request
    - Mark ready when request complete
  - Forall ready requests
    - Handle
  - Select on write (all socks for which there's output)
    - Write response
    - Close when done
- **Select & poll semantics/performance**
  - Select: bit-set, must loop through bitset to find active sockets
  - Poll: array of file descriptor numbers
  - /dev/poll: array moved into kernel

Single-thr, event-driven

Issues:
- Performance
- Resources
- Parallelism
- Synchronization
- Plug-ins
- Robustness

- **Loop**
  - Same as for "single-threaded, non-blocking I/O"
  - But more general event mechanism
- **Events**
  - All I/O is non-blocking (network, IPC, disk, …)
  - I/O completion generates event
  - Modules written as event-handlers
  - Example: serve static file
    - Request-start event
    - Request-read-ready event
    - Disk-read-ready event
    - Response-write-ready event
  - Leads to complex "request state" descriptors
    - `Event_handler(request_record, event_descriptor);`

Process per request

- **Master (e.g. inetd):**
  - Loop
    - Accept
    - Fork (, exec)
- **Child**
  - Read request
  - Write response
  - Close, exit

Issues:
- Performance
- Resources
- Parallelism
- Synchronization
- Plug-ins
- Robustness

Thread per request

- **Same as "process per request"**
  - But all in one address space
- **Threads vs. Events**
  - Threads
    - Many potential points of suspension
    - Straight-line code
  - Events
    - Few well-known points of suspension
    - More code fragments

Issues:
- Performance
- Resources
- Parallelism
- Synchronization
- Plug-ins
- Robustness

Apache 1.3

- **Characteristics**
  - Open source
    - Apache Software Foundation www.apache.org
  - NCSA httpd legacy
    - Completely rewritten since
    - Some legacy configuration options
  - Process per request
    - No thread usage in Apache 1.x
    - Processes persist ("workers")
  - Plug-in modules
    - In-process dynamically loadable modules
    - Most of the server functionality is implemented as modules
  - External request handling programs
    - Basically content-handlers not written in C
    - CGI, fastCGI, ASP, Perl, Java, Python, you-name-it-they've-got-it

Process worker pool

- **Aka. "pre-forking server"**
- **Processes**
  - "master process" + N child processes
  - Children listen+accept+serve
    - Max M times, M is configurable
  - Master
    - forks new children when too few are idle
    - Kills children if too many are idle
  - Communication via table in shared memory
- **Modules live in each process**
  - Same lifetime constraints as host process
  - Same communication constraints
  - Some modules do IPC to external process

Issues:
- Performance
- Resources
- Parallelism
- Synchronization
- Plug-ins
- Robustness

Low-level issue: accept

- Listening on multiple sockets (e.g. ports 80 + 443)
  - See http://httpd.apache.org/docs/misc/perf-tuning.html
-

```
int do_accept( int first_socket, int last_socket) {
  for (;;) {
    fd_set accept_fds; FD_ZERO(&accept_fds);      // build fd set for select
    for (i=first_socket; i<=last_socket; ++i)
      FD_SET(i, &accept_fds);
    rc = select(last_socket+1, &accept_fds, NULL, NULL, NULL);
    if (rc < 1) continue;
    for (i=first_socket; i<=last_socket; ++i)    // find socket to accept
      if (FD_ISSET(i, &accept_fds)) {
        new_connection = accept(i, NULL, NULL);
        if (new_connection != -1) return new_connection;
      }
  }
```

- What happens when N children idle and connection arrives?
  - Solution 1: non-blocking accept
  - Solution 2: accept lock

Request handling phases

- What is it for? (URI xlat)
  - Alias, files, etc…
- Where is it from? (access ctrl)
  - E.g. source IP based
- Who is it from? (authent.)
- Who is allowed? (authoriz.)
  - /etc/passwd, DB, …
- What is the type? (MIME chk)
  - File ext, directory, etc…
- Who will generate response?
  - Content-handlers
- Who will log response?
  - Log file, DB, …
- Who will clean-up?

Sample modules

- mod_userdir: translate the user home directories into actual paths
- mod_rewrite: rewrites URLs based on regular expressions, it has additional handlers for fix-ups and for determining the mime type
- mod_auth, mod_auth_anon,mod_auth_db, mod_auth_dbm : User authentication using text files, anonymous in FTP-style, using Berkeley DB files, using DBM files.
- mod_access: host based access control.
- mod_mime: determines document types using file extensions.
- mod_mime_magic: determines document types using "magic numbers" (e.g. all gif files start with a certain code)
- mod_alias: replace aliases by the actual path
- mod_env: fix-up the environment (based on information in configuration files)

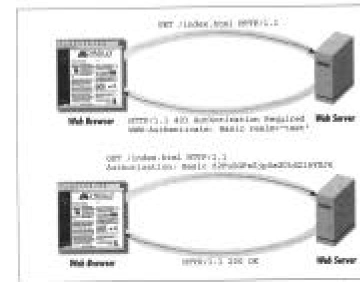
Sample modules (cont.)

- mod_speling: automatically correct minor typos in URLs
- mod_actions: file type/method-based script execution
- mod_asis: send the file as it is
- mod_autoindex: send an automatic generated representation of a directory listing
- mod_cgi: invokes CGI scripts and returns the result
- mod_include: handles server side includes (documents parse by server which includes certain additional data before handing the document to the client)
- mod_dir: basic directory handling.
- mod_imap: handles image-map file
- mod_log_*: various types of logging modules

Apache 2.x

- **Hybrid multithreaded multiprocess**
  - Processes isolate faults
  - Threads reduce resource requirements
- **Implementation improvements**
  - Native process/thread implementations
  - Better build environment (autoconf)
- **Filters**
  - E.g. output of content-handlers can be filtered by mod_include to parse server-side-includes
  - Allows output to drain while handling next request?

HTTP authentication



- Server fails request and asks for authentication
- Browser re-submits request with auth fields
- Browser automatically submits auth fields on subsequent requests
- Defined schemes
  - Basic: clear-text passwords
  - Digest: challenge-response

Challenge-response passwds

- **Example 401 response:**

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="wow@host.com", qop="auth,auth-int",
    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

  - Realm = displayed to user to select password
  - Qop = quality-of-protection auth=authentication, auth-int=authentication+integrity-protection
  - Nonce = random challenge
  - Opaque = “cookie” expected back in response
- **Challenge nonce**
  - Random “challenge” issued to client
  - Example: MD5(time-stamp:ETag:private-secret)

Challenge-resp (cont.)

- **Example repeat request**

```
Authorization: Digest username="Mufasa", realm="wow@host.com",
    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
    uri="/dir/index.html", qop=auth, nc=00000001, cnonce="0a4f113b",
    response="6629fae49393a05397450978507c4ef1",
    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

  - URI = URI used in digest (proxies can change real URI)
  - NC = sequence count to prevent replay
  - Cnonce = client-chosen nonce (prevent chosen-plaintext attacks)
  - Response = MD5 hash of nonce & passwd (see below)

```
MD5(MD5(username:realm-value:passwd):
    nonce:nc-value:cnonce-value:qop-value:
    MD5(method:uri:entity-body))
```