Application Architectures

- Introduction
- Error Architectures
- Error Handling Example
- Centralize Cleanup
- Use Macros
- Error Macros
- Web Technology Overview
- 24 x 7 and Web Constraints
- Web Submission of Student Projects
- Design Constraints
- Overall Design
- Design Highlights
- Detailed Client-Server Protocol
- Upload Requests
- Restarting Server on Reboot
- Read-Only Database to Web Initial Specs
- Design Decisions
- Overall Design
- Detailed Protocol
- Performance
- Updating Data while Running
- Updating Server Code Without Loosing

Requests

- Requirements for E-Commerce Server for Web
- E-Commerce Server Architecture
- E-Commerce Server Highlights
- Design Overview
- Java Complications

Introduction

- Error architectures.
- Look at real-world application architectures for:
    - Server used for web submission of student projects.
    - Server used to present read-only database on web.
    - E-commerce server for web.
- Application architectures constrained by non-technical considerations:
    - What implementor is familiar with.
    - The need to integrate application with other functioning applications.
    - The need to evolve application while retaining compatibility.

Error Architectures

- For an application program, non-recoverable errors resulting in program termination is usually acceptable.
- For a library, program termination is usually not acceptable. The error needs to be conveyed to the application program so that it can decide what needs to be done.
- Error architecture taken from *Advanced UNIX Programming*, by Marc J. Rochkind, 2nd Edition, Addison-Wesley, 2004.

Error Handling Example

Function needs error-return indication. Also, need to cleanup any previously allocated resources on error.

```
if ((p = malloc(sizeof(buf))) == NULL) {
  return false;
}
if ((fdin = open(filein, O_RDONLY)) == -1) {
  free(p);
  return false;
}
if ((fdout = open(fileout, O_WRONLY)) == -1) {
  close(fdin);
  free(p);
  return false;
}
```

Centralize Cleanup

Use `goto` to centralize cleanup and variable state to determine what cleanup is necessary.

```
    char *p = NULL;
    int fdin = -1, fdout = -1;
    if ((p = malloc(sizeof(buf))) == NULL) {
      goto cleanup;
    }
    if ((fdin = open(filein, O_RDONLY)) == -1) {
      goto cleanup;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
      goto cleanup;
    }
    return true;
  cleanup:
    if (p != NULL) free(p);
    if (fdin != -1) close(fdin);
    if (fdout != -1) close(fdout);
    return false;
```

Use Macros

```
  char *p = NULL;
  int fdin = -1, fdout = -1;
  ec_null(p = malloc(sizeof(buf)));
  ec_neg1(fdin = open(filein, O_RDONLY));
  ec_neg1(fdout = open(fileout, O_WRONLY));
  return true;
EC_CLEANUP_BGN:
  if (p != NULL) free(p);
  if (fdin != -1) close(fdin);
  if (fdout != -1) close(fdout);
  return false;
EC_CLEANUP_END
```

Error Macros

- Check argument for error return: `ec_null()` checks for `NULL`, `ec_neg1()` checks for `-1`, etc.
- Use `errno` to accumulate errors in a error buffer which can be displayed using `atexit()`.
- `goto` cleanup-label defined by `EC_CLEANUP_BGN`.
- Need to cope with errors in cleanup-code.

Web Technology Overview

- HTTP protocol (v1.0, 1.1) used between a web browser (like Mozilla) and web server (like Apache).
- HTTP implemented on top of TCP/IP.
- HTTP is stateless. To retain state, usual solutions are:
  - URL rewriting.
  - Hidden parameters to user forms.
  - Cookies.
- To get a static file simply `GET` *Path* where *Path* is relative to the server document-root.
- CGI protocol allows web server to run external programs:
  - Can run a program with parameters passed in URL. Example: `GET /cgi-bin/program?param1=value1&param2=value2`.

    The parameters are passed to the program via the environmental variable `QUERY_STRING`.

    - Can run a program with parameters passed in `POST` request. The parameters are passed to the program via *name*`:` *value* pairs on standard input.

- Most web server can be extended with dynamically loaded modules which are accessed using a web-server specific API.

24 x 7 and Web Constraints

- Web sites must stay up 24 x 7 unattendended. If serious problems, must ask for help.
- Reality is that programs fail; possibly because of outright bugs, but often due to *impossible* changes in environment.
- A web application is distributed across many client browsers. Hence upgrading a web application is non-trivial (a program on the web server may be ugraded but it must still deal with inputs accepted by the previous version).
- Normally, a web server is accessed by a person using a web browser. But that is not always the case (robots); server programs need to be designed to deal with such situations:
    - Client-side parameter validation (using Javascript) is not adequate by itself.

    - Can receive rapid-fire requests much faster than the maximum possible by a human.

Web Submission of Student Projects

- Used before campus adopted Blackboard.
- Could be done using email attachments but, email does not provide a positive ack.
- Need moderate security.
- No root access on web server.

Design Constraints

- Any program run by web server is usually run by same user running web server.
- If this web server user would have access to my directories, then any other user would also have access to the directories.
- Need a separate server program run by my user ID.
- Web server CGI program communicates with server program.
- Server program saves student projects on disk under my user ID.
- Hopefully, other users on machine cannot access directories where projects are saved.

Overall Design

```
+---------+        +--------+       +--------+           +---------+
|         |        |  web   |       |        |   FIFO    |         |
| browser |<------>| server |<----->| upload |<--------->| server  |
|         |  HTTP  |        |  CGI  |  .cgi  |name=value |  .pl    |
+---------+        +--------+       +--------+           +---------+
                                        ^                     ^
                                        |                     |
                                        |                     V
                                    +---------+          +-----------+
                                    |  HTML   |          |passwd file|
                                    |templates|          |  project  |
                                    |         |          |directories|
                                    +---------+          +-----------+
```

Design Highlights

- Server set up as a concurrent server using double-fork technique to avoid zombies.
- Server writes its PID to a well-known PID-file.
- Students passwords and class password are encrypted using DES `crypt()` encoding.
- CGI program uses a timeout to deal with server being down.
- CGI program sends a panic email if it detects a timeout or exception.
- Perl code available.

Detailed Client-Server Protocol

- Initially, the client sends a initial request packet to the server on a well-known `REQUESTS_FIFO`, containing simply it's *pid*.

- The server creates `in.`*pid* and `out.`*pid* FIFO's (where `in`/`out` are relative to the client). It then opens the out FIFO for reading, blocking until the client opens it for writing.

- The client busy-waits until it sees that the `out` FIFO has been created, at which point it opens the `out` FIFO for writing and writes out a detailed request packet. It then opens the `in` FIFO for reading, blocking until the server opens it for writing.

- The server reads the detailed request packet, and then opens the `in` FIFO for writing. It then performs the request, and sends a response packet down the `in` FIFO.

- In all cases, except for a upload request, the client and server wind up processing: the server closes and deletes the `in` and `out` FIFO’s (they will not disappear until the client also closes them). The client reads the response packet and formats it for the user. It then closes its FIFOs.

Upload Requests

For upload requests, the server and client go thru additional steps to get the contents of the uploaded file.

- After reading the response packet from the server, the client starts stuffing the contents of the file down the `out` FIFO and on completion closes the `out` FIFO. It then waits for a upload response packet from the server on the `in` FIFO.

- The server reads the contents of the file from the `out` FIFO and squirrels it away, until it sees a EOF on the `out` FIFO caused by the client closing it. It then sends a upload response packet to the client on the `in` FIFO.

- Both processes then wind up as outlined in the non-upload protocol.

Restarting Server on Reboot

- No root access, hence not possible to directly start server on reboot.
- Have cron access. Used following line in `crontab`:

```
0,15,30,45 * * * * kill -0 ‘cat $HOME/upload_store/pid‘ ||
                  ($HOME/upload_code/server.pl $HOME/upload_store ;
                   echo "Upload server restarted" |
                   /usr/lib/sendmail umrigar )
```

Read-Only Database to Web Initial Specs

- Database represented using flat files.
- Each flat file represented a database table.
- There were about 15 tables containing a total of about 200,000 records containing a total of about 5 MB of data.
- There was a single parameterized query with about 4 parameters.
- Should run on single machine.
- No DBMS available.
- Problem was presented as:

  Here is a C program which takes a query and produces the HTML. Unfortunately, it runs very slow (30 secs cpu time / query). Speed it up.

Design Decisions

- Relatively small amount of data.
- Suck data into memory and build indexes to read data in memory.
- Code for indexing and accessing in-memory data generated automatically from table descriptions.
- Since loading all data into memory takes a large amount of time, amortize the data load time over a large number of queries.
- Using the loaded data over multiple queries requires a architecture where a server program containing the loaded data responds to CGI clients.
- For reliability, use a concurrent server.
- Copy-on-write key to preventing data copying on `fork()` of concurrent server.

Overall Design

```
+---------+        +--------+       +--------+            +---------+
|         |        |  web   |       |        |   FIFO     |  data   |
| browser |<------>| server |<----->|  CGI   |<---------->| server  |
|         |  HTTP  |        |  CGI  |program |name=value  |         |
+---------+        +--------+       +--------+            +---------+
                                         ^                     ^
                                         |                     |
                                         |                     |
                                    +---------+           +-----------+
                                    |  HTML   |           |           |
                                    |templates|           |flat files |
                                    |         |           |           |
                                    +---------+           +-----------+
```

Detailed Protocol

- When the server program is started up by an administrator (or bootup shell script), it loads all the database files. Then it spawns a child to be the daemon, so that the process initially started can terminate. It is the child which really becomes the server daemon. The daemon then listens on a well-known pipe `REQUESTS` for requests from CGI processes.

- When a CGI process is started by a web server, it first creates a new named pipe for the server's response, using its process identifier PID to name the pipe. It then sends a request down the well-known `REQUESTS` pipe; the request includes its PID, along with the query parameters. It then opens the pipe it previously created for reading; the open blocks until another process (the child spawned by the server daemon) opens it for writing. It then substitutes the name=value pairs it reads on this response pipe into a HTML template which it writes onto its stdout (from where the web-server transfers it to the web page).

- When the server daemon sees a request on the `REQUESTS` pipe, it immediately spawns a child (using a double-fork) to process the query and returns to listening at the `REQUESTS` pipe. The child parses the request, extracting the query parameters and the PID of the client. It uses the extracted PID to open the response pipe write-only. It then processes the query using the database it inherited from its parent after the fork. The query processing builds up the answer in an in-memory datastructure. Finally, it outputs the answer onto the response pipe as name=value pairs and terminates.

```
+----------+      REQUESTS pipe        +-----------+
|          |-------------------------->|           |
|    CGI   |                           |   SERVER  |
|  Program |                           |           |
|    PID   |<------------+             +-----------+
+----------+             |                   | double
                         |                   | fork
                         |             +-----V-----+
                         |             |           |
                         +-------------|   SERVER  |
                  RESPONSE pipe (PID)  |    CHILD  |
                                       |           |
                                       +-----------+
```

Performance

- 10-20 msec/query.
- 7-10 seconds for loading database.
- 10-20 MB memory footprint.

Updating Data while Running

- Need to periodically update data.
- Data needs to be updated without loosing user requests.
- Server uploads data on receipt of SIGHUP signal.
- SIGHUP handler writes a special pseudo request to `REQUESTS` FIFO with PID `-1`.
- No requests processed while loading occurs. However, the requests are not lost but merely queued in the FIFO.

Updating Server Code Without Loosing Requests

- Server uses PID file to detect that another server is running. Normally, if it detects that situation it bales out.

- If started with a special option, then it waits to get an exclusive lock on PID file.

- After newly started server is waiting, original server is killed. New server takes over.

- Complications because server executable may be demand-paged. Solution:

  a) Start new server with wait option using temporary location for server executable.

  b) Kill original running server. New server from temporary location takes over.

  c) Copy new server code to normal server executable location. Start it with wait option.

d) Kill server running from temporary location. New server from normal server executable location takes over.

Requirements for E-Commerce Server for Web

- Persistent shopping cart.
- User authentication.
- Credit card verification.
- Interface to legacy enterprise system.
- Necessary to evolve it into existing web site.
- Java used for server implementation.

E-Commerce Server Architecture

```
+---------+        +--------+       +--------+           +---------+
| browser |  HTTP  |  web   |       | Perl   |  TCP/IP   |  java   |
|   with  |<------>| server |<----->|  CGI   |<--------->| server  |
| cookie  |  HTTPS |        |  CGI  |program |name=value |         |
+---------+        +--------+       +--------+           +---------+
                                        ^                     ^
                                        |                     |
                                        |                     V
                                    +---------+         +-----------+
                                    |  HTML   |         |   mysql   |
                                    |templates|         | database  |
                                    |         |         |           |
                                    +---------+         +-----------+
                                                              |
                                                 Asynchronous |
                                                   Connection |
                                                              V
                                                        +-----------+
                                                        |   Legacy  |
                                                        | Enterprise|
                                                        |   System  |
                                                        +-----------+
```

E-Commerce Server Highlights

- Users authenticated using email address and password.
- Password stored in mysql database using a 1-way hash function.
- Credit cards encrypted using a symmetric encryption algorithm.
- Persistent sessions implemented using SessionID cookie in browser.
- Authentication implemented using AuthID cookie in browser. For security, AuthID cookies transmitted only over encrypted HTTPS connection.
- To prevent session capture, SessionID and AuthID cookies consist of two parts: a unique part (generated using a simple counter) and a random part (generated using a random number generator).

- Having asynchronous connection to legacy enterprise system permits 24x7 operation of web site even though legacy system is not 24x7.

Design Overview

- Each request is handled by a separate thread in Java server.
- Each request thread does a exclusive lock on the SessionID. Prevents data interference.
- Java server also has daemon threads for tasks like sending out email and logging.
- Uses agressive cacheing to prevent DBMS delays. When a user connects, most of that user’s information is pulled into in-memory cache (until it times out due to inactivity).
- The cache is write-thru. That is, all data updates are written back to database.
- To provide quick response to the user, the response is sent to the CGI program before database write-backs.

- Because of SessionID lock, a new request for same user cannot proceed until previous request has completed database write-back.

Java Complications

- Java is OS-agnostic. Does not know anything about processes, daemons, FIFOs, etc.
- Forced choice of TCP/IP as communication mechanism with the CGI program.
- Needed a Perl wrapper program to run Java server as a daemon.
- Stopping/reinitializing the server is clumsy:
  - Periodically, the server checks for the existence of a stop-file. Stops/reinitializes based on the existence/size of this stop-file.
  - A minimal web server was run within the Java server using a separate thread. The server can then be controlled using any web browser (over a authenticated connection).