Achieving Fault Isolation in a Surveillance System

Vincent Yeung
March 18, 2004

Table of Contents

1. Design Overview

This design seeks to achieve three primary goals: fault isolation, scalability, and graceful performance degradation. It achieves fault isolation by carefully separating tasks into processes with separate address spaces. A failure only has a permanent effect on the process generating the failure, while other processes in the system continue normal operation. The system is scalable because it only requires change in several global constants to support additional cameras. Raw processing power in hardware, not the software architecture, is the limiting factor on the number of cameras this design can support. If the system hardware occasionally experiences overload, the system adapts to the problem and decreases processing until system conditions return to normal.

The system begins in a main process, the master process. The master process contains only one thread, which is responsible for creating and maintaining two other types of processes. It begins by creating a single web server process, which is responsible for obtaining suspicious images from the system's video processing software and serving them to human spotters via the Internet. The web server process is assumed to be fairly reliable and does not need frequent attention of the master process, except to restart it in the event that it terminates unexpectedly. Besides the web server process, the master process also creates video analysis processes, each of which is responsible for a single camera for a small duration of time before it terminates and is replaced with a new process on the same camera.

I considered several approaches for organizing the ways in which the three major components of video processing—video retrieval from camera, transcoder conversion, and anomaly detection with the AI routine—interacted, and decided on the design which I feel best achieves the project's primary goals. In my design, every camera has its own video process performing all of the three aforementioned tasks. That is, the process (1) begins, (2) obtains a short and fixed amount of video from a camera using HTTP GET, (3) transcodes it, (4) runs the AI analysis, (5) notifies the web server via HTTP POST if it finds suspicious behavior, and (6) terminates. Once this video process terminates, the master process starts a new video process on the same camera. Therefore, for a system with 1000 cameras, under ideal system conditions there will be about 1000 concurrent processes, one working on a short video segment for each camera. Later in this report, I will describe in detail how such a design achieves fault isolation and copes with heavy load on the system.

A video process reports suspicious frames to the web server process through HTTP POST requests. The scheme keeps the communication between various processes simple because the web server already needs to have the capacity to handle HTTP requests, which are also sent by human spotters via the Internet. The video process includes additional information, such as the originating camera number and capture time, along with a suspicious still frame and posts it to the web server.

The web server uses a SPED architecture, which employs a single process to serve its two groups of clients, those within the system (the video processes) and outside the system (the human spotters on remote terminals). It is event-driven and checks for newly completed tasks or arriving requests at every iteration. The web server stores suspicious images in a queue and serves them to spotters sending HTTP GET requests.   Spotter votes are submitted using HTTP POST and interpreted by the web server accordingly.

2. Design Description

The system operates on a single computer, which, when initially turned on, will boot with the software’s program image in memory on a single process, which is what I will refer to as the *master process*.   The master process will create the web server process and a video analysis process for every camera, after which the surveillance system begins its operation.

2.1 Master Process

The master process creates the web server and video processes using the system call fork(). The child process from fork() calls the appropriate procedure to perform its assigned duties. The following is the pseudocode for launching the child processes.

```
int child_pid, camera, www_pid;
int camera_pid_table[NUM_CAMERA];

void main() {
  /* create web server process */
  child_pid = fork();
  switch (child_pid) {
    case 0:          /* inside www process  */
      run_www();
      exit(0);
    default:         /* inside master process */
      www_pid = child_pid;
      break;
  }

  /* create first set of video processes for every camera */
  child_pid = fork();

  for (camera = 0; camera < NUM_CAMERA; camera++) {
    switch (child_pid) {
      case 0:          /* inside video process  */
        process_video(camera);
        exit(0);
      default:         /* inside master process */
        camera_pid_table[camera] = child_pid;
        break;
    }
  }
}
```

Figure 1: Pseudocode for starting web server and video child processes in the master process

After creating the first set of video child processes, one for each camera, the master process must start a new process for a camera when its previous process terminates, regardless of whether the termination was due to a successful completion of video processing or an illegal operation in the child process. To accomplish this, I assume that it there is functionality in the kernel for the master process to be notified that a child is about to exit. In particular, I assume I have access to an interface like the signal interface in current versions of UNIX. With that interface, I can run the signal() procedure in the master process with a "signal child" argument and a location to the handler for this signal. Then, when a child is about to terminate, the master process will receive control and execute the specified handler.

The handler determines which camera's process terminated and restarts a new one for that camera. If the handler detects that the previous two or more consecutive processes for the camera ended from a "timeout" instead of normal termination or an illegal operation, the system will think that either the specific camera or the system as a whole is experiencing unusually high processing load, and it will attempt to alleviate the situation by using a simple exponential back-off algorithm, only resetting the wait time when a process on the camera no longer times out. This approach to handling processing overload is desirable in this system for two reasons. First, it detects the case in which a specific camera experiences trouble transmitting its video promptly and executes exponential back-off only when spawning new processes for that camera, while processes for other cameras are spawned without delay. Second, in the event that the main system server itself is experiencing processing overload, processes of all cameras will be affected and will presumably time-out, so exponential back-off will be applied to all cameras, and the system should eventually return to normal conditions.

Conveniently, the handler will also detect the unlikely scenario of an unexpected termination of the web server process and will restart a new one if that happens. The following code illustrates the details of the implementation.

```
/* this goes somewhere in main(), before any child process is created */
signal(SIGCHLD, catch_child);

/* this is an array that stores the number of consecutive times a child
process on each camera timed out.  The initial value is 0 for each element */
int consec_timeouts[NUM_CAMERA];

void catch_child(int sig_num)
{
  int dead_pid, new_pid, status;
  int cam;

  /* stores pid of exiting child in dead_pid and returns exit status */
  status = wait(&dead_pid);
  if (dead_pid == www_pid) {
    /* the child was the web server process.  restart a new one */
    -> fork new child, run run_www(), reset www_pid
  }
  else {
    /* the child was a video process. restart one on the same camera */
    cam = search_camera(camera_pid_table, dead_pid);  // looks up array

    if (status == PROCESS_TIMED_OUT) {
      consec_timeouts[cam]++;
    } else consec_timeouts[cam] = 0;
    -> if more than 1 consecutive timeouts, then exponential backoff
       (by sleeping) in proportion to the number of consecutive timeouts

    -> fork new child, run process_video(), reset camera_pid_table[cam]
  }
}
```

Figure 2: master process handler for child processes that are about to terminate

2.2 Video Processes

As mentioned previously, each camera is handled by its own sequence of child processes spawned by the master process. Each of these child processes, which I will refer to as a *video process*, analyzes a fixed amount of video data from the camera. By having one process per camera, as opposed to a handful of processes for all cameras, this design avoids the issue of "starving" a particular camera by not looking at its data because the system is busy working with other cameras' data for significant periods of time. Furthermore, it eliminates the need to have a central scheduling thread that manages the assignment of limited processes to contending cameras. My design is also highly scalable because, as long as there is sufficient processing power in the hardware, it is easy to accommodate additional cameras because the system simply needs to create one more process for each of them; on the other hand, in an alternative system that uses a manager to schedule a fixed number of processes to cameras, the scheduling algorithm that is used may need to be significantly changed to avoid starving cameras when the number of cameras increases by an order of magnitude, since the starving time will increase proportionally.

Besides having a separate process for each camera, this design has two other key features, which are (1) that each video process only gathers a fixed amount of data from a camera before processing, and finally terminating, and (2) that all of the data retrieval and analysis is done on the same process. It is important that each video process only works on a bounded amount of video because it helps the system achieve fault isolation. Given an unexpected termination in the transcoder or AI subroutines, we would like as few parts of the system to be affected as possible, and we would want to ensure that the system does not attempt to repeat the same inputs leading to that error. In this design, an unexpected failure in the transcoder or AI would only cause the system to lose the bounded amount of video data on which it was operating. Therefore, by setting this bound on the video data length per process to be sufficiently small (but not so small that the frequent re-spawning of processes becomes a significant performance or reliability cost), one can limit the adverse effects of a transcoder or AI failure to only a short amount of data loss from a single camera. Furthermore, the decision to do retrieval and analysis in the same process means that problem inputs will never be repeated, since the video is lost at the same time as the processing subroutine terminates; doing these tasks on the same process also allows us to avoid tricky issues with inter-process concurrency.

Once a video process receives the result of the AI module's analysis, it can decide what to do with it using the estimated threat level, which is measured on a decimal scale from 0.0 to 1.0. My design uses an adjustable threshold level (THREAT_THRESHOLD) that triggers a report to the web server process if the computed threat is higher than the threshold; in that case, the video process sends an HTTP POST request to the local system’s web server and transmits the suspicious frame, estimated threat level, source camera number, and frame capture time.

In the previous section, I mentioned that video threads can exit automatically if they take longer than a preset timeout period to run. Besides being used for coping with system overload as described previously, this feature also allows the system to terminate processes that may have entered into infinite loops in buggy transcoder or AI code. To implement timers, I assumed that the kernel has support for the UNIX alarm() system call that allows us to specify a procedure as a handler when an alarm times out. I also assumed that multiples alarms can be used concurrently across different processes without interfering with one another.

The following pseudocode summarizes the operation of a video process:

```
/* an array that is initialized with unique port numbers for each camera */
int camera_port_num[NUM_CAMERA];

void process_video(int cam) {
  signal(SIGALRM, catch_alarm); // to catch alarms
  alarm(VIDEO_PROCESS_TIMEOUT); // starts timer
  if (no_stream_on_port(camera_port_num[cam])) {
    /* stream was never opened or somehow got closed, so open it  */
    http_get(camera_host[cam], camera_port_num[cam], "/video");
  }

  listen(camera_port_num[cam]);
  stream_id = accept_stream(camera_port_num[cam]);

  /* get bytes from stream */
  read_stream(stream_id, camera_buffer, BYTES_PER_VIDEO_PROCESS);
  /* transcoder */
  transcode(camera_buffer, BYTES_PER_VIDEO_PROCESS,
        jpeg_buffer, &jpeg_buffer_size);
  /* AI analysis */
  vision_result = detect_anomaly(jpeg_stream, jpeg_stream_size);

  if (vision_result > THREAT_THRESHOLD) {
    http_post(localhost, "put_threatening frame",
          vision_result.frame, vision_result.threat, cam, time());
  }
  alarm(0); // resets timer
}

void catch_alarm(int sig_num) { exit(PROCESS_TIMED_OUT); // timed out }
```

Figure 3: Video process pseudocode, with alarm handler

2.3 Web Server Process Architecture

The web server uses a SPED architecture for its simplicity and performance. Because the system exclusively uses RAM for data storage, there was no need to worry about kernel support for asynchronous disk I/O or other disk-related drawbacks that SPED may have. On the other hand, a SPED design makes it easy to implement the web server functionality needed for this system. Each feature is just implemented as a state or sequence of states on the SPED server, which cycles and selects an awaiting request at every iteration (see Figure 4).

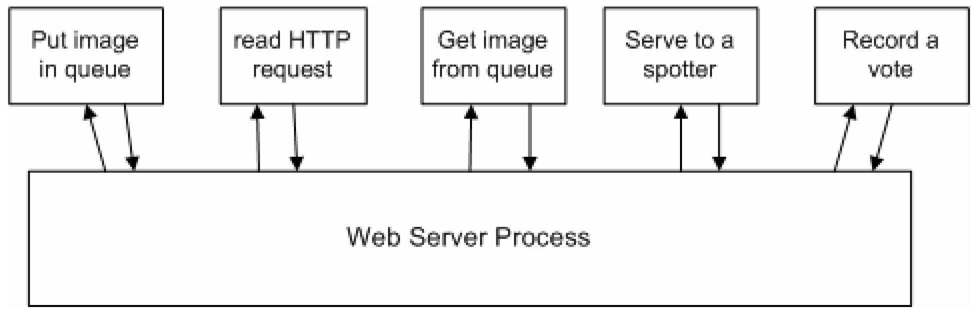

Figure 4: Web server architecture, with selected functionality displayed as tasks

The human spotter interacts with the web server process through a regular web browser using HTTP requests. When a spotter requests his or her first frame to evaluate, the server creates a unique spotter id and serves the id along with the frame. The spotter's browser is then responsible for keeping track of the spotter id and including it in subsequent requests in order to avoid receiving duplicate images. The exact interface for supported GET and PUT requests is presented in the following table:

Request name	Type	Fields	Description
get_frame	GET	spotter_id	Used by spotters to get a new frame to evaluate
put_vote	PUT	frame_id, vote	Used by spotters to respond
put_threatening_frame	PUT	frame, threat, camera, time	Used by video processes to post images

Table 1: Interface for GET, PUT requests to web server

For convenience, I created an "entry" data structure to encapsulate the useful information regarding a threatening frame. In particular, each entry holds the JPEG image, its unique "frame id," threat level, originating camera, capture time, a list of the spotters who have been served this image, and the total number of responses received thus far. Employing this data structure, procedures that interpret the HTTP requests described above can be written as follows:

```
void put_request(request req) {

   if (req = "put_threatening_frame") {
     /* initialize an entry's field with field values */
     entry.frame = req.frame;
     entry.threat = req.threat;
     entry.camera = req.camera;
     entry.time = req.time;

     entry.num_response = 0; // number of responses received for this frame
     entry.served_spotters = new linked list

     entry.id = generate_unique_id(); // generates a unique id for this frame
```

```
    <- request a "put in queue" operation for this entry
  }
  else if (req = "put_vote") {
    entry = lookup_entry(req.frame_id);
    vote = req.vote;
    if (entry != null) {
      <- request a "record vote" operation for entry and vote
    }
    else { // entry no longer in database, discard vote }
  }
}

void get_request(request req) {
  if (req = "get_frame") {
    spotter_id = req.spotter_id;
    if (is_invalid(spotter_id)) {
       spotter_id = generate_spotter_id();
    }

    <- request a "get image from queue" operation for spotter_id
  }
}
```

Figure 5: Pseudocode for procedures to interpret GET, PUT requests

2.4 Image Management and Voting

The database for holding entries is implemented as a straightforward linked-list queue. Thus, the aforementioned “put in queue” request merely attaches a new entry at the back of the queue. The “get image from queue” request, on the other hand, requires more work. It traverses the queue, from the beginning, until it encounters an entry whose served_spotters list does not contain spotter_id. When it finds such an entry, it adds the current spotter_id to the list and prepares to send the image in that entry to the spotter. The following code illustrates this implementation:

```
void get_image_from_queue(spotter_id) {
  entry = database.first;
  do {
    if (entry.served_spotters.contains(spotter_id)) break;
  } while ((entry = database.next) != null);

  if (entry == null) {
    <- serve spotter a page saying there is no new image currently available
  }
  else {
    <- serve spotter entry.image
  }
}
```

Figure 6: Pseudocode for getting images from the queue

The voting system also operates using a straightforward algorithm. For every response received for an entry, the threat level of the entry is changed by a fixed amount. It is increased if the spotter believes there is suspicious activity, decreased if the spotter believes there is no suspicious activity, and unchanged if the spotter is uncertain. If the modified threat level drops below a low threshold, the entry is deemed a false alert and removed from the database. If the modified threat level rises above a high threshold, the system raises its hardware alarm that is linked directly to the appropriate authorities. The following code provides the details of this procedure:

```
void put_vote(entry e, int vote) {
  switch (vote) {
    case YES :
      e.threat += YES_THREAT_INCREASE;
      break;
    case NO:
      e.threat -= NO_THREAT_DECREASE;
      break;
    default:
      break;
  }

  e.num_response++;

  if (e.threat > YES_THREAT_THRESHOLD) {
    <- raise alarm!
  }
  else if (e.threat < NO_THREAT_THRESHOLD) {
    <- remove e from database
  }
  else if (e.num_response > MAX_RESPONSE_NEEDED) {
    <- remove e from database, still undecided after enough responses
  }

}
```

Figure 7: pseudocode for voting procedure

Finally, the web server thread will periodically run a maintenance program that removes entries whose elapsed times are larger than a threshold ENTRY_LIFESPAN.

2.5 Assumptions and Feasibility

This design made a number of assumptions. First, I assumed that several system calls unspecified in the project handout were available to me; these calls include signal() and alarm() and were described as I used them. Second, I assumed that when the system leaves a TCP video stream open but does not read the stream, the camera will fill up its send buffer but continue to operate. I assume that it will overwrite its filled buffer with fresh video data and correctly transmits new data when the system reads it again. This is an important assumption because my design uses exponential back-off to shed load, and my video processes leave video streams open when exiting to reduce the overhead of closing and reopening the connection every video process cycle.

For the organization of processes in performing video retrieval and analysis, I evaluated various alternatives to my current design. I first considered separating the retrieval, transcoding, and AI tasks into three separate processes. This appeared to offer the advantage of “pipelining” (in the sense that the system is able to transcode a video segment and concurrently obtain newer data from the same camera, thus having video ready by the time the transcoder finishes with its current segment). On the contrary, in my final design, the system falls behind the video stream when a video process is running the transcoder or AI routines. However, if we pick a sufficiently large BYTES_PER_VIDEO_PROCESS value to read each time, then on average the system should catch up to the stream because it will be able to clear the read buffer at times when the video is streaming at a relatively low rate.

My design also has the additional benefit of avoiding much of the overhead processing required for most types of inter-process communication. By accumulating three tasks into one process, I can share variables and make up for the possible loss in performance for not having a pipelining scheme.

A design I considered that was similar to my final one was to again get and analyze video for a camera in a single process, but instead of having each process work for a short video segment and terminate, the alternative design had the video process loop back and get a new video segment on the same camera. The reason I chose to stay with having the short-lived processes despite the costs of terminating and creating processes is that it allows for the elegant solution of using timers to solve infinite loop problems and perform exponential backoff.

Interestingly, the system uses HTTP POST to communicate to its web server, as opposed to more traditional forms of inter-process communication, such as pipes. I chose this design because it takes full advantage of the HTTP protocol and the interface that is already in place in the SPED architecture used by the web server. Moreover, there should not be a significant performance tradeoff for this increased simplicity because connecting to a port on a local host should be comparable to connecting via pipes.

Finally, the system uses a straightforward queue and scoring system to collect votes from human spotters. Obviously, if all spotters fail to respond within a reasonable amount of time, the system is bound to fail, but my design has provisions against the occasional idle spotter because it does not use a direct “majority-rules” system, i.e. a system that serves an image to a fixed number of spotters and only raises the alarm if half that number responds with “yes.” Such a system would effectively count a lack of response as a “no,” but my system is able to treat it as a neutral answer.

2.6 Possible Design Extensions

A priority queue instead of a linked-list queue for the web server’s database of images would be an improvement to my current design because it will give images of higher estimated threat level priority when selecting images to serve spotters. However, due to time constraints, I did not fully design this feature, and I believe that if the system has enough human spotters, then the need for such an optimization may be diminished.

My design also ignored a wide array of security issues, which were not in the scope of this project’s immediate goals. However, in a real system that relies as heavily on the Internet as ours, it is important to have at least some form of encryption and/or session management to provide protection against malicious attackers.

3. Conclusion:

After considering several alternatives for the Surveillance@Home system, I arrived at one which I believe satisfies the initial goals of fault isolation, scalability, and acceptable performance degradation when overloaded. Fault isolation is achieved by having a separate process for each camera that obtains and analyzes a short video stream. The process sends information to the web server via HTTP POST if it decides that the video is suspicious, and exits. The master process then spawns a new video process for the same camera to retrieve and analyze another short video segment. The system detects system overload by keeping track of consecutive video process exits caused by “timeouts” and uses exponential backoff to reduce system load in that situation. Clearly, if the system is consistently overloaded, this solution will not work as the wait times will increase exponentially, and that is when we know we need faster hardware.

The design of the voting scheme for spotters provides fault isolation for the other type of “processors” that exist in our system, the human spotters. A mistake of one individual in selecting his choice will not significantly alter the final outcome of whether or not an alarm is triggered. Furthermore, as mentioned before, small numbers of individuals who take longer than expected to “process” or vote will not noticeably affect the correct operation of the system.

4. Acknowledgements

I discussed many of the ideas in this design with Jelani Nelson. I also owe much thanks to Simson Garfinkel for his office hours and insightful comments.