Swank

The Fault-Tolerant, Multithreaded, High-Performance Surveillance Server for Growing Businesses

An Integral Component of Surveillance@Home

Sean Leonard

6.033 Computer Systems Engineering

Spring 2004, 3/17/2004

Design Project 1

Recitation #4, TR11am

Recitation Instructor: Prof. Michael Ernst

Teaching Assistant: Kathryn Chen

Abstract

Corporate properties require wide surveillance, but traditional surveillance systems are expensive. Swank provides a robust, scalable, fault-tolerant, and low-cost alternative for wide-area surveillance. Swank's three modules continue to process under widely varying camera loads, unstable algorithms, and inconsistent spotters.

Table of Contents

List of Figures

Word Count

Executive Summary: 1,190 words
Main Text: 3,745 words
Total: 4,935 words
These counts exclude figures, captions, headings, headers, and footers.

Executive Summary

Problem, Constraints, and Solution Overview

Surveillance@Home requires a web-surveillance platform that provides fault-tolerance, load-balancing, and reasonable performance in its delivery of camera data to spotters around the world. The Swank system meets and exceeds these criteria with its multithreading kernel and robust three-module design. Swank's kernel enforces hard-modularity and fault isolation between modules, while providing safe connections such as pipes between them. Swank's three core modules, *ReadCameras*, *Transcode/Detect*, and *SwankWeb*, respectively read camera data, transcode data and detect anomalies, and serve resulting images to spotters over the Internet. A fourth module serves two purposes: it initializes the three core modules, and allows Swank to recover from serious and unusual errors by periodically monitoring and restarting problematic processes.

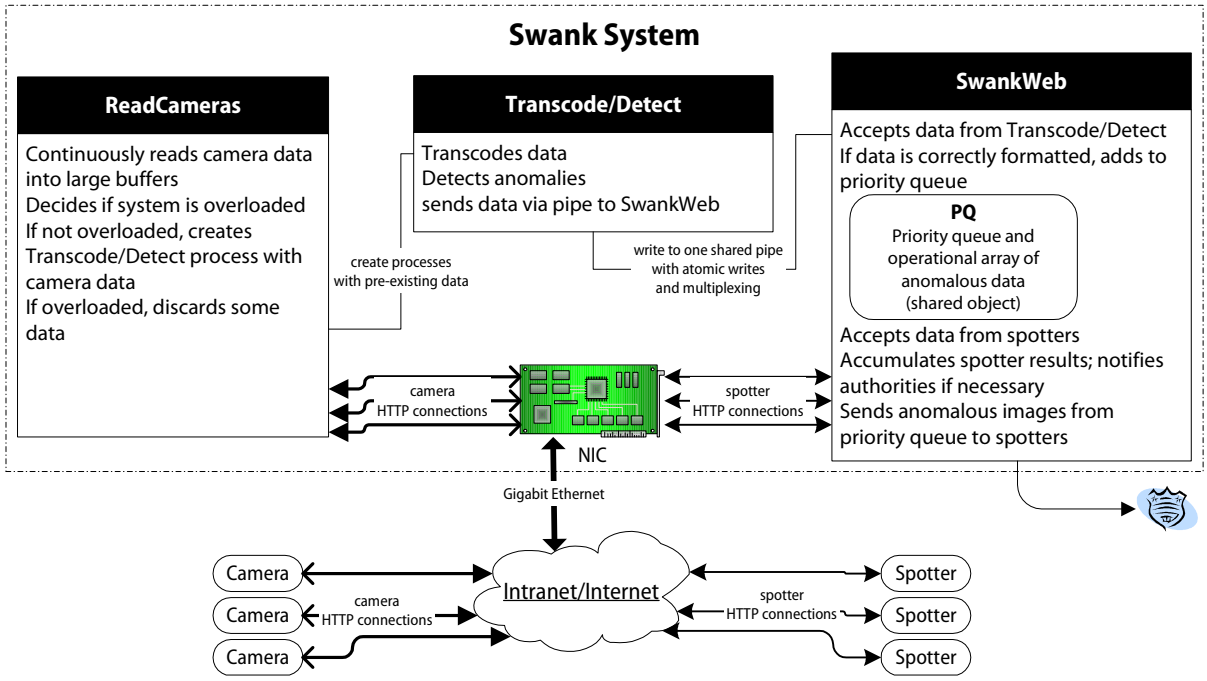

Figure 1: A complete Swank system diagram, with purposes of and connections between ReadCameras, Transcode/Detect, SwankWeb, and the system hardware. Each module pictured occupies a separate address space. The kernel and guardian process are not explicitly pictured; see Section 2.4 on page 10 for details.

These user modules and system components are illustrated in Figure 1 above, and are described in the following sections. ES.1 summarizes Swank's kernel and hardware, and then describes a general evolution of processes from system boot-up to steady-state. Subsequent sections disclose the

features and design rationales of Swank's three modules. Finally, ES.5 illustrates Swank's performance by measuring how much of the processor a camera sequence would require in the steady-state.

System Design, Reliability, and Performance

ES.1. Kernel and Hardware Specifications

Swank operates on the commodity PC outlined in Design Project 1, including a 2.4GHz Intel® Pentium® 4 processor with Hyper-Threading Technology [1], a clock chip programmed to send 25ms "heartbeats," and a system bus arbiter for TSL support [2]. The Gigabit NIC implements an 8MB buffer, a size large enough to hold 50ms minimum of camera data with 1.75MB to spare. Swank's kernel follows Design Project 1's specification, with the following enhancements and clarifications in Figure 2.

```
// Swank Kernel additions: multithreading, streams, interrogation

// pre-emptive multithreading scheduler with TSL
yield();            // see 6.033 Notes 2.D [3]
acquire(L);         // see 6.033 Notes 2.E [2]
release(L);         // see 6.033 Notes 2.E [2]

const _POSIX_PIPE_BUF ← 256KB;              // atomic writes
n ← read_stream(stream_id, buffer, size);   // nonblocking
n ← write_stream(stream_id, buffer, size);  // nonblocking

howmanychildren ← countpschildren(); // number of forked processes
                                     // from caller in O(1) time
memory ← howmuchmemoryfree(); // number of bytes of unallocated
                              // memory in system, O(1) time
```

Figure 2: Additional and clarified functions in Swank kernel.

The multithreader supports both pre-emptive and cooperative thread switching, as well as thread coordination for shared variables. Both `read_stream` and `write_stream` are nonblocking. They return negative values if the stream is closed or an error has occurred, otherwise reporting 0 or more bytes read or written (for details, see Section 3.1 on page 11). `Write_stream` guarantees to write up to `_POSIX_PIPE_BUF` bytes atomically per call [4]: Swank assumes a value of 256KB, which is several standard deviations larger than an average JPEG sent through the system's pipes. Finally, ReadCameras relies on the system interrogators when it determines whether or not to drop camera data.

Swank's computer boots up and loads the kernel as outlined in the 6.033 notes [5]. A general process evolution is provided in Figure 3 below. Note that Swank, the main user process, never terminates. After creating a shared pipe and forking ReadCameras and SwankWeb with the

`pipe_id` of the pipe, Swank serves as a "guardian angel" that restarts those processes with a fresh pipe, if they terminate abnormally.

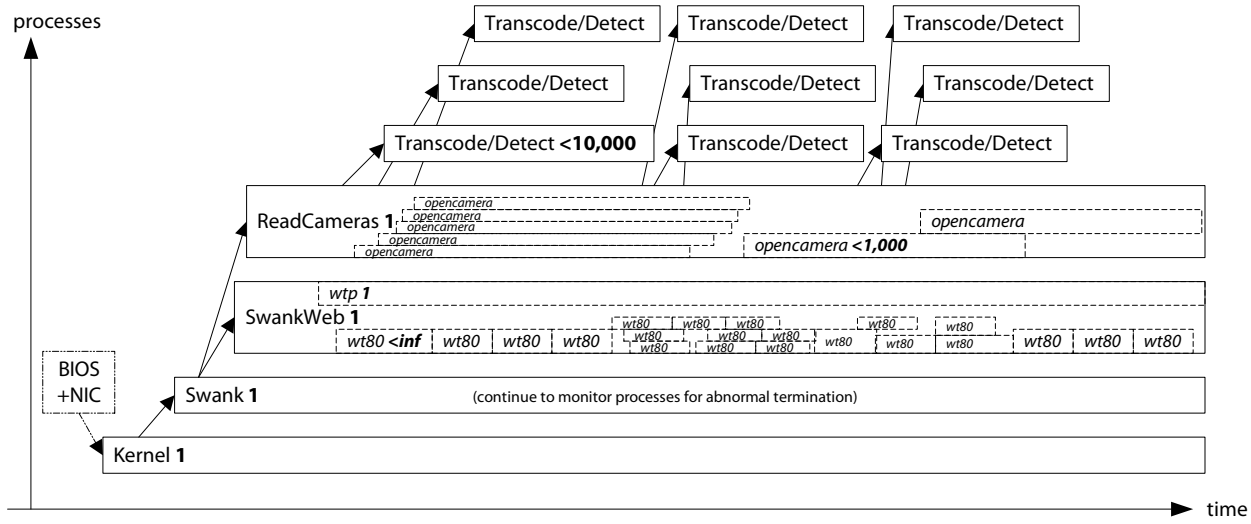

Figure 3: Process evolution diagram. Bold values indicate the maximum concurrent instances of each process or thread; italicized names indicate threads in a process. Each process occupies a separate memory space, illustrated as a solid line.

ES.2. ReadCameras

After initialization by Swank, ReadCameras establishes connections to the cameras, retrieves camera data, and decides whether or not to send camera data to Transcode/Detect. If a camera connection is unavailable, ReadCameras creates an asynchronous *opencamera* thread to establish the HTTP request/response chain, since `open_stream` may block for several seconds. ReadCameras copies all available data in camera streams in a round-robin fashion, well before the NIC's small buffer overflows. Connections are permanently held open; however, if a connection fails, ReadCameras will spawn a new *opencamera* thread to reopen a connection. If the system is not busy, as defined by `countpschildren` and `howmuchmemoryfree`, and if the camera buffer has accumulated enough data, ReadCameras will fork a new Transcode/Detect process. When it forks, ReadCameras sets the `copy_on_write` bit to analyze a "chunk" of camera data without copying a large buffer; it also passes pointers to the correct data "chunk," and supplies metadata such as the camera ID. In the worst case, ReadCameras takes 400μs to pass through the loop and fork 1,000 processes. At the end of each loop, ReadCameras calls `yield` to allow other processes to execute before emptying the NIC buffer again.

ES.3. Transcode/Detect

Transcode/Detect calls `transcode` and `detect_anomaly`, gets the result of these routines, marshals the result into the format `(size of all data)`, `(camera ID)`, `(time)`, `(threat)`, `(frame_size)`, `(JPEG frame)`, sends the data through a shared pipe if the total size is no greater than `_POSIX_PIPE_BUF`, and terminates. `Transcode` and `detect_anomaly` are guaranteed to fail at some time. Placing them in separate address space will allow them to fail independently of all other processes, at which point the kernel will teardown the crashed process' memory space. Performance-wise, process setup and teardown time is negligible compared to the time to run `transcode` and `detect_anomaly`.

ES.4. SwankWeb

SwankWeb perpetually maintains one shared priority queue, *pq*, and one additional thread called *wtp* that reads on the shared pipe. *Wtp* continuously reads and unmarshals data. If data is invalid or out of sync, *wtp* will close the pipe and call `exit` to terminate SwankWeb; the main Swank process will recognize this condition and create fresh SwankWeb and ReadCameras processes. While drastic, it is conceivable that a spurious Transcode/Detect process could actually rewrite part of its marshaling code without failing, thus sending corrupt data through the pipe. Restarting SwankWeb and ReadCameras is highly inefficient, but this case is expected to be so rare that a restart will almost never be required.

If values in the message appear valid, *wtp* calls a public method `pq.Add` to add data to the queue. *Pq*– implemented in an object-oriented programming language—achieves soft-modularity, information hiding, and acceptable performance: the web server must read data from *pq* concurrently if it is to serve thousands of spotters at once. Internally, *pq* contains a priority queue implemented as a heap and a dynamically-resizable "operational array." The "operational array" allows the web server to send JPEGs to spotters concurrently: *pq* replaces anomalous data in the array from the top of the heap. If the only "operational data" for the web server were at the top of the heap, the data would bottleneck the whole process for at least ten seconds per image, the delay from sending out a JPEG to receiving a spotter's response.

For every request that SwankWeb receives, it creates a *wt80* thread designed like a "single-instance" MT in the Flash paper [6]. *Wt80* writes the preliminary HTTP header and HTML start to notify the client that is has started to service the request. Following this action, *wt80* calls

`pq.Update` to update metadata for anomalous images based on the request's HTTP POST. Finally, *wt80* calls `pq.Output` to output a new image to the spotter, closes the stream, and terminates.

ES.5. Performance

Under a few, well-justified assumptions, Swank spends an average of 0.31% of processor resources on each request. For a 10-second stream of video, the system cannot spend over 10ms per request. Swank spends 31µs, plus an estimated average of 6ms on `transcode` and `detect_anomaly`. Section 4.4, "Performance for a data sequence," in the main text provides details on these assumptions.

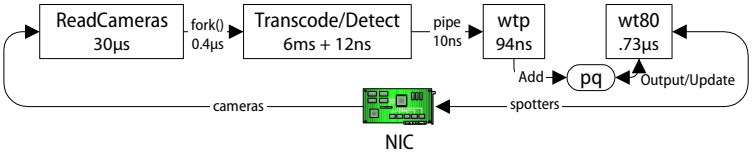

Figure 4: A simplified processor-consumption diagram for each module and connection, including wasted processor time due to memory accesses. For details, see Figure 12 on page 21.

Conclusion

Swank splits the design for a web-surveillance server into three multithreaded, fault-isolated modules, with additional components monitoring and supporting the base design. These modules are connected with several different mechanisms, including shared pipes and acquire/release semaphores. Although the Swank design cannot eradicate long-term overload, it can handle sudden spikes by eliminating data early in the data pipeline. Swank can also recover from broken connections using the *opencamera* thread and the Swank guardian process. These features make Swank an ideal choice for a web-surveillance system.

1. Introduction

Swank provides fault-tolerance, load-balancing, and excellent performance in its delivery of camera data to spotters around the world. Swank accomplishes these objectives with a multithreading kernel and a robust three-module design, breaking tasks up into camera-reading, signal processing, and web-serving stages. Each module executes independently of the others, achieving fault isolation through the hard-modularity that the kernel and hardware enforce. Within each module, software contracts ensure soft-modularity and fault tolerance in spite of potential threading hazards and corrupt data. Each module successively filters data to improve performance across the whole system. A fourth module adds recoverability to Swank, periodically monitoring and restarting problematic processes.

2. Design Overview

2.1. Modularity in Design

Swank distributes its workload among three different modules, each separated by hard-modularity contracts established by the kernel and the network. Although Swank's design depends upon this three-module abstraction, Swank further specifies methods for interprocess communication, a guardian process for error recovery, and a kernel for multithreading and memory-mapping. These critical pieces grew naturally out of the three-module abstraction, and are described below.

2.2. Three Components

The three core functions of Swank are to read camera data, to transcode and analyze data, and to serve data to and from spotters. These functions correspond to Swank's three modules: *ReadCameras*, *Transcode/Detect*, and *SwankWeb*, which are explored in this section. Swank's three-module abstraction appears in Figure 5 below, sorted in the order of camera dataflow.

2.3. Multiple Connections

Figure 5 illustrates the many interconnections between components in the Swank system: each connection is tailored for fault-tolerance, performance, and interoperability as each module demands. ReadCameras communicates to Transcode/Detect by forking new Transcode/Detect processes from itself: by forking with the `copy_on_write` bit set, both processes isolate faults without incurring a buffer-copy penalty. Transcode/Detect processes send marshaled data to SwankWeb via a shared

pipe, which guarantees that properly-marshaled data will be written to the shared resource. The HTTP connections follow semantics defined in the HTTP/1.0 standard [7], permitting seamless communication with other network applications. Intra-module connections, such as those between helper threads in ReadCameras, are explored in detail in section 3, "Design Description."

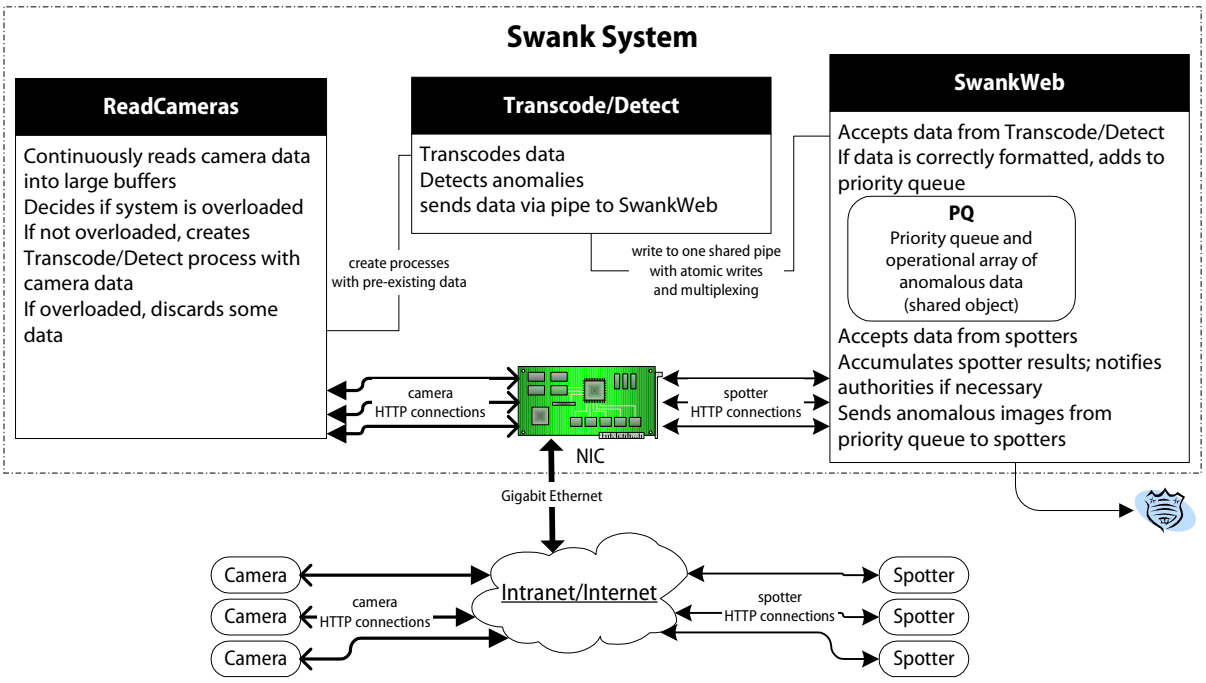

Figure 5: A complete Swank system diagram, with purposes of and connections between ReadCameras, Transcode/Detect, SwankWeb, and the system hardware.

2.4. Monitor Above, Kernel Below

In addition to the three modules and multiple connections above, the Swank system contains two additional, critical components: the monitor process *Swank* and the *Swank kernel*. Swank serves as a "mother" and a "guardian angel." The process contains all executable user-mode code for the Swank system, but forks off child processes to run the code. After launching its child processes, Swank perpetually monitors the health of ReadCameras and SwankWeb. If either process terminates, Swank will restart them and will provide them with fresh, uncorrupted resources.

The Swank kernel holds the system together: all user processes rely on it to establish interprocess and network communication, to report status of the system, to map, expand, and revoke memory as needed, and to rapidly switch between threads of execution.

3. Design Description

3.1. Details of the Kernel and Protocol Stack

Swank implements the Unix-like kernel described in Design Project 1, with several enhancements to support multithreading, process interrogation, and nonblocking streams.

The kernel supports pre-emptive and cooperative multithreading with kernel-threads and locks, as outlined in the 6.033 notes [5]. The pre-emptive scheduler relies on the computer's clock chip, which sends a regular "heartbeat" of interrupts every 25ms. While Swank's internal modules are built to follow cooperative multitasking principles, `transcode()` and `detect_anomaly()` may not return for long periods of time. Therefore, a single Transcode/Detect process cannot consume the system for more than 25ms. The scheduler favors executing lower-order processes—that is, parent processes of the interrupted process—when a 25ms interrupt occurs. `Yield()` is supported: when a program calls `yield()`, the scheduler will transfer execution to another process at random. If `fork()` or `fork_thread()` are called, the parent process or thread resumes execution before the new process or thread. The Swank kernel also implements `acquire(L)` and `release(L)` for thread-locking as outlined in [2]; the system's bus includes an arbiter to support these calls. `Release(L)` has no effect if the thread did not acquire the lock.

The protocol stack of the Swank kernel implements the same functionality described in Chapter 4 of the 6.033 notes [8]. The TCP portion of the stack in the end-to-end layer has an identical buffer size as the Gigabit NIC; both buffers are 8MB, a size large enough to hold 50ms (twice the pre-emptive interrupt period) minimum of camera data with 1.75MB to spare.

To access the NIC and memory, the kernel implements nonblocking `read_stream(stream_id, buffer, size)` and `write_stream(stream_id, buffer, size)` calls. `Read_stream()` attempts to read `size` bytes from the stream into the buffer provided; the function returns `n`, the number of bytes actually read into the buffer. If less than `size` bytes are immediately available, `n` will be less than size. If `read_stream()` returns `0`, the stream is open and available, but no bytes are waiting in the stream. If the stream is not available or an error occurs, `read_stream()` returns a negative value.

`Write_stream()` writes bytes to the specified stream, returning the number of bytes that the underlying stream believes will be written successfully in finite time. If `stream_id` refers to a TCP socket [9], `write_stream()` returns asynchronously with the TCP guarantee that `n` bytes

will eventually be sent to the sender in order. If `stream_id` refers to a kernel pipe created by `pipe()`, `write_stream()` guarantees immediate writing to memory before returning `n`. As with `read_stream()`, `write_stream()` failures return negative values. Pipes do not close unless the process that created the pipe crashes. According to the POSIX standard [4], `write_stream()` further guarantees that up to a certain `_POSIX_PIPE_BUF` of bytes will be written atomically: if multiple processes each attempt to write q_i bytes to the same stream, and if q_i is less than or equal to `_POSIX_PIPE_BUF`, then q_i bytes will remain in order in the stream. If q_i is greater than `_POSIX_PIPE_BUF`, then each `_POSIX_PIPE_BUF` chunk of bytes are guaranteed to be in order. Swank's kernel specifies the `_POSIX_PIPE_BUF`) as 256KB, or 2^{18} bytes: this value corresponds to several standard deviations larger than the average JPEG output from `transcode()`, plus the size of some metadata associated with the image.

The kernel implements two additional functions: `howmanychildren ← countpschildren()` and `memory ← howmuchmemoryfree()`. `Countpschildren()` returns the number of running child processes created with `fork()` by the current process. The function returns in constant time: the kernel keeps a total of running child processes of every process. `Howmuchmemoryfree()` returns the number of free bytes of *total memory*. *Total memory* includes addressable physical memory and memory swapped to disk: since the Swank system has no disk, total memory is physical memory. ReadCameras will use these functions to selectively discard camera data during periods of high stress.

3.2. Initial Process Evolution

3.2.1. Swank Initialization

The computer boots up and loads the kernel as outlined in the 6.033 notes [5]. The kernel loads the Swank process as the first user program in memory. Swank creates three variables: a restart counter for SwankWeb, a restart counter for ReadCameras, and a pipe. The pipe structure will eventually be shared by the three modules in the swank system. Swank then acts a stub, starting up the SwankWeb and ReadCameras processes using `fork()`. Swank enters an infinite loop, usually calling `yield()` to yield control to other processes. Periodically, however, Swank checks the system to see if processes have terminated abnormally, in which case Swank will spawn fresh shared pipes and replacement processes as necessary.

3.2.2. SwankWeb Initialization

After Swank forks the SwankWeb process, the code branches off in its own function, `SwankWeb()`, which takes the shared pipe and SwankWeb restart count as arguments. SwankWeb creates the priority queue data structure illustrated in Figure 5 on page 10. SwankWeb calls `fork_thread(wtp, pipe_id)` to create the workhorse thread for reading on the shared pipe (*wtp*); the *wtp* thread takes the shared pipe as an initial argument in the same fashion as `clone()` [10]. SwankWeb assumes the role of the external web server, waiting for requests to come in on port 80 after calling `listen(80)` and `accept_stream(80)`. SwankWeb calls `fork_thread(wt80, http_stream_id)` for every request received. Each *wt80* processes data in the MT-style described in the Flash paper [6], except that *wt80* starts with an open stream, and terminates after servicing its request.

3.2.3. ReadCameras Initialization

The ReadCameras process accepts the shared pipe and restart count as initial arguments. ReadCameras creates 1,000 large buffers (`camera_buffer[1000]`), pointers to current buffer "chunk" positions and current read positions (`cb_chunk[1000]` and `cb_pos[1000]`), remaining size of the buffer (`cb_rs[1000]`), failure counters (`camera_failures[1000]`), and `stream_ids` (`camera_streams[1000]`) corresponding to the 1,000 cameras from which it reads. Each camera buffer is 1.5MB long (1.5GB for 1,000 cameras), corresponding to 12 to 240 seconds of raw camera data (at 1Mbps peak rate, or 50Kbps minimum rate). The process has camera IP addresses in its initial boot image (`camera_IPs[1000]`), which are loaded into memory. As ReadCameras receives camera data, it will continuously `fork()` new Transcode/Detect processes to analyze the data.

3.3. Details of ReadCameras

After initialization by Swank, ReadCameras establishes connections to the cameras, retrieves camera data, and decides whether or not to send camera data to Transcode/Detect (see Figure 6). ReadCameras processes each camera in a round-robin fashion, essentially implementing a cooperative user-level multithreader without context-switching overhead. If `camera_streams[i]` is 0, ReadCameras creates a new thread *opencamera* to open an HTTP stream to the i^{th} camera and sends the request "GET /video HTTP/1.0," then saves the `stream_id` to the i^{th} position in the `camera_streams` array (see Figure 7). `Open_stream()` may block if the camera takes a long

time to respond, necessitating a separate thread. No race condition can occur, since only *opencamera* writes to `camera_streams`, and ReadCameras critically depends on `camera_streams` at only one point per cycle. ReadCameras calls `read_stream()`, passing data from the protocol stack's small buffers to ReadCamera's sizeable buffers well before the protocol stack's buffers overflow.

If a camera does not respond, ReadCameras resets the buffer, increments the failure counter, and attempts to reconnect to the camera. The failure counter is periodically decremented after successful connections. If ReadCameras cannot connect to a camera after many repeated attempts, it reports an error message directly to SwankWeb in the same format as the JPEG/metadata struct `vision_result`. Problems with the shared pipe are more serious: as a precautionary measure, such problems cause ReadCameras to terminate, which has the side-effect of closing all camera streams.

```
// in initial boot image
const camera_IPs[1000]; const CAMERAS ← 1000;
// available to both functions in process
camera_streams[1000];

void ReadCameras(pipe_id, restart_count) {
   // initialize camera_buffer[1000], cb_chunk[1000],
   // cb_pos[1000], camera_failures[1000], and
   // cb_rs[1000] using resize_heap() as necessary
   if (write_stream(pipe_id, 0, 0) < 0) exit(1);
   for (int i; i < CAMERAS; i++) {
      // NOTE: at any point, if the return from a syscall
      // is negative, the failure count is incremented and
      // camera_streams[i] is set to 0; excessive problems
      // induce an error message to be written to the shared pipe
      // open stream: initialize camera connections
      if (camera_streams[i] <= 0) {
         fork_thread(opencameras, i);
      } else {
         // read stream
         sizeread ← read_stream(camera_streams[i], cb_pos[i],
            cb_rs[i]);
         cb_rs[i] ← cb_rs[i] - sizeread;

         // drop data, or fork Transcode/Detect process
   }  } // end if camera_streams, end loop over all cameras
   yield();
}
```

Figure 6: General pseudocode for ReadCameras process. Note that italicized comments are placeholders for more pseudocode.

After reading, ReadCameras calls `countpschildren()` and `howmuchmemoryfree()` to determine how busy the server is. If `countpschildren()` is greater than 10,000 (assuming the average latency of each Transcode/Detect process is about 10ms in a multithreaded environment), or if `howmuchmemoryfree()` is less than 250MB, ReadCameras

will discard bytes from its camera stream after reading them, and will use inverse exponential backoff to discard existing bytes in the camera's buffer.

```
void opencamera(camera_id) {
   activestream ← open_stream(camera_IPs[camera_id], 80);
   write_stream(camera_streams[camera_id], "GET /video HTTP/1.0\n",
      stringsize);
   // read_stream() until the null line separating the entity body
   // is encountered
   // avoid race conditions assigning shared variable just-in-time
   camera_streams[camera_id] ← activestream;
}
```

Figure 7: Pseudocode for *opencamera* thread, which "primes" the HTTP request/response chain for raw reading.

If ReadCameras determines that the data in a buffer is "ready" (400KB long for 3-30 seconds of data, or the buffer has reached its 1.5MB limit), `fork()` is called with the `copy_on_write` bit set [11], creating a new Transcode/Detect process to operate on the "chunk" of data and its companion camera and time-completed information. By using `copy_on_write`, the massive camera buffer available to the Transcode/Detect becomes available without the penalty of copying that buffer. The kernel enforces memory isolation, preventing Transcode/Detect from modifying the data associated with ReadCameras. ReadCameras increments the "chunk" after forking, so that the remainder of the buffer can be used for the next bytes of the stream without forcing the kernel to copy memory pages for Transcode/Detect. This drop/fork process is illustrated in Figure 8 below.

```
// drop data, or fork Transcode/Detect process
if (countpschildren()>10K || howmuchmemoryfree()<250MB) {
   // binary exponential backoff: go back on buffer
} else if (cb_pos[i] - cb_chunk[i] >= 400KB) {
   switch (fork()) {
   0:  // Transcode/Detect
      TranscodeDetect(pipe_id, camera_IPs[i], time(),
         cb_chunk[i], cb_pos[i] - cb_chunk[i]);
      break;
   default:  // still in ReadCameras
      // reset or increment as necessary arrays[i]
      break;
}  } // end switch on new process, end if server busy
```

Figure 8: Drop/fork pseudocode. This code expands the pseudocode in ReadCameras.

3.4. Details of Transcode/Detect

After forking from ReadCameras, each Transcode/Detect process begins with `TranscodeDetect(pipe_id, camera, time, bufferpointer, size)`, as illustrated in Figure 9. Transcode/Detect calls `transcode()` and `detect_anomaly()`, gets the result of those routines, and if successful, sends the result to SwankWeb via the shared pipe. The Swank design places these two routines in their own module for fault isolation and efficient

multithreading. The routines are guaranteed to crash at some time. Placing them in separate address spaces will allow a routine to fail independently of all other processes, at which point the kernel will teardown the crashed process's memory space. The kernel consumes a negligible amount of time to setup and teardown a Transcode/Detect process relative to the lifetime of that process, so there is virtually no performance hit associated with creating separate processes.

```
// about 30KB/jpeg * 10 seconds * 5 jpegs/second
MAX_JPEGBUFFER_SIZE ← 1.5MB;
MAX_JPEG_SIZE = _POSIX_PIPE_BUF - sizeof(float) - sizeof(long)*4;

void TranscodeDetect(pipe_id, camera, time, bufferpointer, size) {
       jpegbuffer ← new jpegbuffer(MAX_JPEGBUFFER_SIZE);
   transcode(bufferpointer, size, jpegbuffer,
       MAX_JPEGBUFFER_SIZE);
   yield();
   vision_result vr;
       vr ← detect_anomaly(jpegbuffer, MAX_JPEGBUFFER_SIZE);
   // see page 18 for definition of THREAT_MINIMUM
   if (THREAT_MINIMUM <= vr.threat <= 1.0 && 0 < vr.frame_size <
       MAX_JPEG_SIZE && vr.frame_pointer is valid) {
       sizeofeverything ← vr.frame_size + float and 4 longs
       streambuffer ← new streambuffer();
       // put (size of everything), (camera), (time), (threat),
       // (frame_size), and JPEG into streambuffer
       write_stream(pipe_id, streambuffer, sizeofeverything);
   }  exit(0);
}
```

Figure 9: Pseudocode for Transcode/Detect. Note the marshaled format of the data.

If values in `vision_result` are invalid (e.g., JPEG has no size, or `threat_level` is outside [0.0, 1.0]), or if the pipe has been closed, Transcode/Detect terminates silently as if it had failed to determine a result. Otherwise, Transcode/Detect marshals the `vision_result` data into the format in Figure 9 when sending it through the pipe to *wtp*.

3.5. Details of SwankWeb

SwankWeb perpetually maintains one shared data structure, *pq* for priority queue, and two threads of execution (see 3.1 above); this analysis considers the pipe-reading thread *wtp* first, since it follows in the data path from Transcode/Detect. Discussions of *pq* and SwankWeb follow.

When *Wtp* initializes, it attempts to read one `long` value from the pipe, corresponding to the length of the rest of the message. *Wtp* continually calls `read_stream(pipe_id, readbuffer, (remaining size))` and `yield()` until it reads an entire message. The message is unmarshaled and its component values are examined; finally, *wtp* calls a public method `pq.Add(camera, time, threat, JPEG, size)`. *Wtp* repeats this process until no

more data can be read, at which point it calls `yield()`. If any data in this sequence is invalid, e.g., the first `long` is longer than `_POSIX_PIPE_BUF - 1`, *wtp* will attempt to resynchronize with the pipe's data, although the specifics of resynchronization are beyond the scope of this document. For now, *wtp* closes the shared pipe and forces SwankWeb to terminate with `exit(1)`, signaling to Swank that SwankWeb and ReadCameras need to be restarted.

For system stability and modularity, *pq* is implemented as an object in an object-oriented programming language such as C++. This design choice lets *pq* hide its private data—a priority queue implemented as a heap [12] and an "operational array" of initial length 6,000, much larger than `MAX_CAMERAS`—and expose public methods to add, update, and output the data. The operational array ensures that SwankWeb can serve many highly-suspicious images concurrently, rather than blocking on the top of *pq*'s heap for the full latency between Swank and the distant spotter.

Swank's design makes a tradeoff for the sake of performance: *pq* could be in a completely separate process, thus achieving hard-modularity between the three SwankWeb components. Specifying *pq* as an object enforces only soft-modularity through contracts. As a shared object, however, *pq* can be read by multiple *wt80* threads at once, rather than serially. With potentially thousands of spotters accessing the server at once, optimizing parallel reads is much more important than parallelizing writes, which is the province of the serialized *wtp* design. Figure 10 below illustrates the complete class specification, with especial attention to locking of shared data.

```
class PQ {
   // in addition to Transcode/Detect data, stores net
   // suspiciousness of image, and how many views image has had
   struct {camera, time, threat, JPEG, size, suspiciousness,
      views} pqseq;
   PQ() { // initialize heap and servarray }
   ~PQ() { // lock all heap, all servarray, delete all items }

   Add(camera, time, threat, JPEG, size) {
      if (other criteria met) {
         if (threat > heap[0].threat) acquire(toplock);
         acquire(bottomlock);
         // create new pqseq and add to heap in O(log n)
         release(bottomlock);
         release(toplock);
   }  }

   Update(sa_i, camera, time, threat, size, suspiciousness);
   boolean Output(sa_i, stream_id);
   Resize_Op_Array() { // dynamically resize high-priority array }
   long SizeOfHeap() { // size of heap }
   long SizeofOpArray() { // size of operational array }
private:
   pqseq** heap;       // the internal heap
   pqseq** servarray;  // high-priority array, currently serving
   otherdata cameradata[MAX_CAMERAS]; // data per camera
};
```

Figure 10: Priority Queue with information hiding and thread-safe methods.

Wtp calls `pq.Add()`, which adds data to the priority queue if it matches certain criteria. One criterion is that the threat-level exceeds a certain minimum, which would be found by inspection at each Surveillance@Home site. To eliminate data as early as possible in the pipeline, Swank tests this criterion in Transcode/Detect (see Figure 9 on page 16), since the criterion is not dependent on the state of user responses. Other criteria may be specified depending on per-camera or per-time data: for example, if one camera has a scarecrow in front of it, and if spotters report that the data is clearly not suspicious, *other criteria* can dynamically set a higher minimum threat-level for that camera for a period of time.

In contrast, *wt80* calls `write_stream(http_stream_id, HTTP response & HTML beginning, data size)`, then `pq.Update()` and `pq.Output()` as shown in Figure 5 below. Before making these calls, *wt80* unmarshals values found in the "POST *url* HTTP/1.0" request from the client, if such a request exists. *Wt80* will only call `Update()` if the unmarshaled values are valid, and `Update()` will only update the item if the spotter is submitting data for the identical item. If a spotter submits data after the item in the "operating server array" is removed or replaced, the spotter's data is discarded.

After `Update()`, `sa_i` is incremented modulo the size of the operational array. In the absence of an HTTP POST, `sa_i` takes a random value in the interval [0, oparraysize). `Output((sa_i + 1) mod pq.SizeOfOpArray(), http_stream_id)` sends the next item's identifying information along with the embedded JPEG image [13] and HTML end-text, so that `Update()` can validate data received from "POST *url* HTTP/1.0" requests. When `output()` terminates, *wt80* may choose to dynamically resize the operational array to handle greater or fewer concurrent requests. *Wt80* finally calls `close_stream(http_stream_id)` to complete the HTTP request/response chain.

```
const MAX_VIEWS_BEFORE_DECISION ← 5   // five witnesses is enough

PQ::Update(sa_i, camera, time, threat, size, suspiciousness) {
       acquire(servarraylock(sa_i));
       if (servarray[sa_i] = this data) {
           servarray[sa_i].suspiciousness ←
               servarray[sa_i].suspiciousness + suspiciousness;
           servarray[sa_i].views ← servarray[sa_i].views + 1;
           if (servarray[sa_i].views > MAX_VIEWS_BEFORE_DECISION) {
               // alert authorities if pqseq is extremely suspicious
               // store some per-camera data if desired
               // delete pqseq from memory
               servarray[sa_i] = NULL;    // nullify pointer
       }  }
       release(servarraylock(sa_i));
   }

boolean PQ::Output(sa_i, stream_id) {
       acquire(servarraylock(sa_i));
       // if OpArray item is null, repopulate with top of heap
       if (servarray[sa_i] = NULL) {
              acquire(toplock); acquire(bottomlock);
              servarray[sa_i] = heap[0];
              // remove top of heap, sort heap in O(log n)
              release(bottomlock); release(toplock);
       }
       // if still no data, write_stream a "no data" JPEG
       write_stream(stream_id, servarray[sa_i] & HTML end, size);
       release(servarraylock(sa_i));
       return (servarray[sa_i] != NULL);
   }
```

Figure 11: Update and Output methods called by *wt80*: these public methods protect *pq*'s internal data from race conditions and arbitrary manipulations.

4. Design Tradeoffs and Performance

Swank's design includes certain tradeoffs. This section considers three alternatives that were rejected because they did not meet performance criteria, or could have left the system unstable. The last subsection illustrates expected performance of a single data sequence through Swank, demonstrating that Swank exceeds performance criteria. In all cases, the Swank system is assumed to

include a 2.4GHz Intel® Pentium® 4 processor with Hyper-Threading Technology [1], a 533MHz front-side bus [14], and a bus-mastering 64-bit PCI bus: the NIC transfers data to and from main memory with minimal CPU intervention.

4.1. ReadCameras round-robin scheduling and camera connections

An alternative to one ReadCameras process is 1,000 threads that read on all 1,000 cameras. Running 1,000 threads requires 1,000 extra context switches, however, and there is no guarantee that one thread will not switch to another expensive Transcode/Detect process. Furthermore, all 1,000 threads read from a shared resource, the NIC. In contrast, the single ReadCameras process requires ten to five hundred instructions per camera per loop-cycle. Assuming the worst-case for the `fork()` operation, each loop must execute a maximum of 500K instructions. At 2.4GHz, the loop latency is a mere 400μs, or 26.4ms if a Transcode/Detect process stalls; both values are far below the 50ms it would take for the NIC's internal buffer to overflow.

ReadCameras also never drops camera connections, although it can recover from dropped ones. This optimization prevents the overhead of renegotiation on every cycle.

4.2. Shared pipe reset

Although explicitly disallowed by Swank's design, mysterious errors in Transcode/Detect—for instance, overwriting of very specific executable code in memory—could cause corrupt data to be sent through the shared pipe to SwankWeb. For this reason, Swank has a way to reset the shared pipe by restarting both SwankWeb and ReadCameras, and by recreating the shared pipe. ReadCameras will terminate when it realizes that its shared pipe is invalid. However, even if a concurrent ReadCameras process continues to execute, the Transcode/Detect processes it forks will silently terminate after attempting to write to the closed pipe. The reset method described is extremely inefficient, but the circumstances for its execution are also extremely rare.

4.3. Priority queue, lock-blocking, and wt80

Although SwankWeb creates *wt80* threads for every request, it seems that most *wt80* threads will eventually be serialized waiting for the `toplock` lock. While serialization is expected in most circumstances, Swank must be engineered to continue accepting new web requests, even in periods of high stress. In most cases, *wt80* will complete before returning control to SwankWeb. However, in cases where *wt80* takes exceptionally long to complete, such as when it is waiting on many queued

`Add()` operations from *wtp*, SwankWeb will still be able to handle additional requests. *wt80*'s first `write_stream()` ensures that the client gets some static data, so it knows that the server has not timed out.

4.4. Performance for a data sequence

Given a few well-justified assumptions, this subsection predicts the average time a piece of data takes to travel through Swank, and determines that this time exceeds performance requirements. One camera sends data at the rate of 50Kbps-1Mbps, which enters the NIC and is quickly copied into the `camera_buffer[i]` of ReadCameras. At average data rates, 400KB (3.2Mb) of data from about 10 seconds is sent to Transcode/Detect, meaning each data sequence must consume a maximum of 10ms of processor time. ReadCameras consumes 30µs for this operation: 4ns for 10 instructions and 2ns for memory copy (over the 533MHz front-side bus) [15], executed 5,000 times over loops in ReadCameras in the worst case, plus 1,000 instructions to fork a Transcode/Detect process. Guidance from Design Project 1 suggests that an average (not peak) 10-second data stream would take 2ms to transcode, and that the resulting JPEGs would take about 4ms to detect anomalies. Writing to the shared pipe may also take some time, perhaps about 25 instructions or 10ns-worth plus a memory write hit of 2ns.

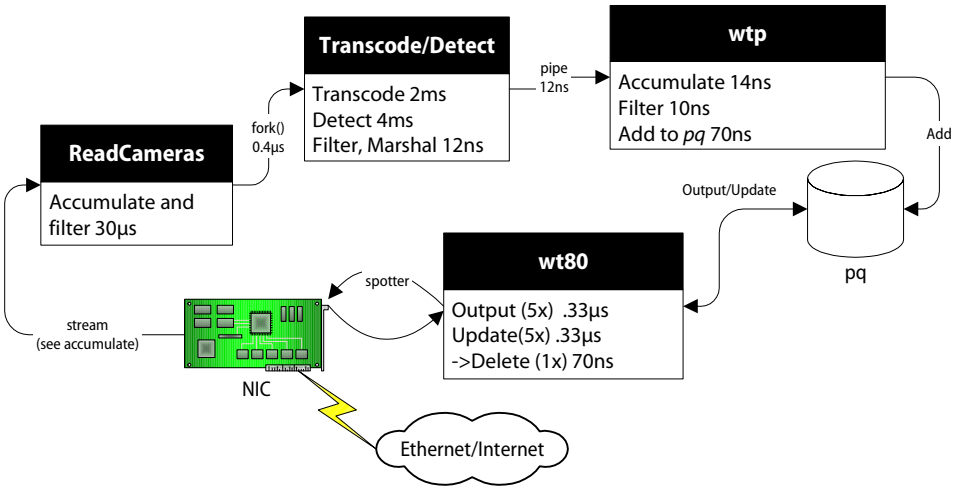

Figure 12: The round-trip processor time for the Swank system. Without Transcode/Detect, the system consumes 0.31% of processor resources.

Next, *wtp* unmarshals the data, adds it to the priority queue, outputs the data to spotters, accumulates the results, and then deletes the data or notifies the authorities. Reading and unmarshaling from the shared pipe takes 14ns of dedicated processor time in the worst case. Although the data rate through the pipe is only 3MB/sec (30KB/10 seconds per camera × 1,000 cameras), the

processor spends considerably less time waiting on the pipe once *wtp* calls `yield()`. Therefore, on every pass *wtp* will unmarshal many messages. `Add()` spends only 70ns in the worst-case scenario. If a full *pq* occupies 500MB of memory (4GB - 1.5GB - 1.5GB with 500MB slack), and if the average JPEG occupies 30KB, then there may be as many as 16,667 entries in *pq*. `Add()` takes O(log *n*) time, so log 16,667 = 4.22 × 25 instructions/operation = 50ns, plus 20ns for memory access. `Output()` and `Update()` without run in finite time of about (100 instructions + 24ns memory reads) × 5 times = 0.33μs each; the recursive delete operation, like `Add()`, requires an additional 70ns.

Summing the loop, each 10-second camera "chunk" requires 6.031ms of dedicated processor time, or 0.31% without `transcode()` and `detect_anomaly()`. The Transcode/Detect process contributes the vast majority of this delay, suggesting that Swank is indeed a light, fast, and efficient architecture.

5. Conclusion

Swank splits the design for a web-surveillance server into three multithreaded, fault-isolated modules, with additional components monitoring and supporting the base design. The three core modules execute in their own address spaces, preventing failures in one module from immediately propagating to others. Swank components are connected with several different mechanisms, including shared pipes for interprocess communication and acquire/release semaphores for single-process multithreading. If these inter-thread or interprocess channels fail, Swank can also restore the system through a variety of means. ReadCameras can spawn new *opencamera* threads to restore network connections, while Swank can reinitialize all three core modules if the shared pipe fails.

Swank also addressed performance in its module-centric design. Although this design cannot eradicate long-term overload, it manages to degrade gracefully under periods of high stress. Swank further reports these chronic overloads to users. Armed with this knowledge, an informed operator can expand the system's resources as the system serves wider areas and processes more complicated camera feeds.

6. References

6.1. Acknowledgements

I discussed the general design with Michael Lin early in the development process. Jesse Smithnosky suggested the use of a priority queue implemented as a heap. Prof. Michael Ernst provided assistance with the specifics of the shared pipe, and motivated me to think about performance bottlenecks in the design. Mary Caulfield assisted me several times with checking and revising the main text.

6.2. References

[1] Intel, "Intel® Pentium® 4 Processor Product Information," [online document], 2004, [cited 2004 Mar 17], Available HTTP: http://www.intel.com/products/desktop/processors/pentium4/

[2] J. H. Saltzer, "Coordinating references to shared writable variables," in Topics in the Engineering of Computer Systems. MIT, Cambridge, MA, 2004, sec. 2-E.

[3] J. H. Saltzer, "Enforcing modularity with threads," in Topics in the Engineering of Computer Systems. MIT, Cambridge, MA, 2004, sec. 2-D.

[4] Linux Documentation Project, The, "6.2.4 Atomic Operations with Pipes," [online document], 1996 Mar 29, [cited 2004 Mar 17], Available HTTP: http://www.tldp.org/LDP/lpg/node13.html

[5] J. H. Saltzer, "Enforcing modularity with virtual memory," in Topics in the Engineering of Computer Systems. MIT, Cambridge, MA, 2004, sec. 2-C.

[6] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: an efficient and portable Web server," presented at Annual Usenix Technical Conference, Monterey, CA, 1999 Jun, [cited 2004 Mar 17], Available HTTP: http://web.mit.edu/6.033/www/papers/pai99flash.pdf

[7] T. Berners-Lee, R. Fielding, and H. Frystyk, "RFC 1945: Hyptertext Transfer Protocol -- HTTP/1.0," [online document], 1996 May, [cited 2004 Mar 17], Available HTTP: http://www.freesoft.org/CIE/RFC/1945/

[8] J. H. Saltzer, "The Network as a System and as a System Component," in Topics in the Engineering of Computer Systems. MIT, Cambridge, MA, 2004, sec. 2-E.

[9] J. Postel, "RFC 793 - Transmission Control Protocol," [online document], 1981 Sep, [cited 2004 Mar 17], Available HTTP: http://www.faqs.org/rfcs/rfc793.html

[10] Linux Programmer's Manual, "Unix man pages: clone (2)," [online document], 1998 Apr 25, [cited 2004 Mar 17], Available HTTP: http://www.rt.com/man/clone.2.html

[11] Linux Programmer's Manual, "Unix man pages: fork (2)," [online document], 1995 Jun 10, [cited 2004 Mar 17], Available HTTP: http://www.rt.com/man/fork.2.html

[12] J. Morris, "Data Structures and Algorithms: Heaps," [online document], 1998, [cited 2004 Mar 17], Available HTTP: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/heaps.html

[13] R. E. Critchlow Jr, "Inline Images on Web Pages," [online document], 2003 May 16, [cited 2004 Mar 17], Available HTTP: http://www.elf.org/essay/inline-image.html

[14] IBM, "IBM US IBM - ThinkCentre A - United States," [online document], 2004, [cited 2004 Mar 17], Available HTTP: http://www-132.ibm.com/webapp/wcs/stores/servlet/CategoryDisplay?catalogId=-840&storeId=1&langId=-1&dualCurrId=73&categoryId=2580521

[15] M. Schulte, "Lecture 13: Main Memory," [online document], [cited 2004 Mar 17], Available HTTP: http://www.eecs.lehigh.edu/~mschulte/ece401-01/lect/my-lec13-p2.pdf