Dan Ports A

A nice job that shows a full understanding of the SEDA architecure and its proper application to this design project. Among the few nits, I would point out:

* failure to handle "uncertain" responses from spotters (DP askes for second opinion)

* failure to count "in handler queue" events as part of "queue length" (which is actually "buffer length")

* failure to handle infinite-loop crashes in buggy modules

* I'm concerned about your HTTP accumulator. Each camera connection is kept open indefinitely. This essentially means that your system will be receiving all data seen by all cameras. We certainly have the bandwidth for this in the steady state, but if you get overloaded transcoding etc, isn't it possible to build up an arbitrarily large backlog? We haven't really defined what the camera does at that point, but it obviously doesn't have infinite buffers. It would be plausible for the camera to close the tcp connection at some point. Or to crash! So perhaps you should close the connection yourself if the backlog builds up. Indeed, there are numerous other reasons the connection might close unexpectedly, so for robustness I would like to see some mechanism for reopening a connection to the camera if the old one breaks.

* Your design hands off file descriptors between processes. This doesn't work; an fd is only "meaningful" to the process that created them (and to any that were forked from the creating process _after_ creation.

A Concurrently-Staged Design for Surveillance Monitoring

Dan Ports

March 18, 2004

Executive Summary

1 Design Overview: The SEDA Model

We present the design for a Surveillance-At-Home system. The primary objectives of this design are, in order, fault isolation, graceful degradation during periods of unexpectedly heavy load, and high performance and scalability.

These objectives are met using a novel design based on the Staged Event-Driven Architecture (SEDA) model developed by Matt Welsh. This model decomposes the system's processing into a pipeline of individual *stages* that perform one segment of the processing, as in our system's dataflow diagram in Figure 1. Each stage is executed concurrently, using one or more processes per stage.

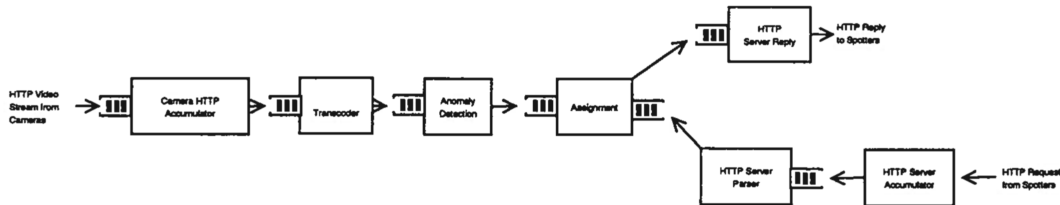

Figure 1: System data flow diagram

The input to each stage is an *event*: a data structure that contains all the information necessary for the stage's processing. In this system, an event represents either a partially-processed segment of video data from one camera, or an incoming HTTP request from one of the spotters' web browsers.

Each stage consists of a *stage controller* that maintains a queue of events to be processed, and *event handler* subprocesses that may be created by the controller to process individual events. Each stage controller uses a socket-based interface to listen for incoming events. Other stages submit incoming events by connecting to the destination stage controller and transmitting the event in a marshalled form. The destination stage controller then places the event in its queue. It either processes the event itself, or creates an event handler process to do so. Enqueuing is the only means for stages to communicate with each other, which helps enforce modularity and provide fault isolation.

1.1 The Stages

The following stages are used in this design:

Camera HTTP Accumulator This interface stage opens the HTTP connections to the cameras. It then operates as a loop (without event handler subprocesses) that receives data from the

cameras via the HTTP connection It stores incoming data from each camera in a buffer, and divides it into one-second segments that are formatted as events and passed to the Transcoder stage.

Transcoder This stage uses event handlers as wrappers for the TRANSCODE procedure. It converts events from the Camera HTTP Accumulator stage into events that can be processed by the Anomaly Detection stage.

Anomaly Detection Similarly, this stage uses event handlers as wrappers for the DETECT-ANOMALY procedure. It takes in a transcoded video stream, and outputs a threat index and the single most suspicious frame.

Assignment This stage handles the assignment of video frames to spotters. It operates by using a loop that queries two input queues: one receiving scored frames from the Anomaly Detection stage, and one receiving HTTP requests from the HTTP Server Parser. Received frames are stored in a state table. An incoming HTTP request causes a frame to be selected from the state table according to a weight function that ensures that the highest-threat frames are selected first, without allowing any camera to be starved of attention. The pairing of the HTTP request and selected frame is enqueued for the HTTP Server Reply stage.

HTTP Server Accumulator The HTTP Server Accumulator is an interface stage that listens on TCP port 80 for HTTP connections from the spotters, It buffers incoming data, passing HTTP requests to the HTTP Server Parser stage.

HTTP Server Parser This stage parses an incoming HTTP request from a spotter's web browser, and passes it to the Assignment stage.

HTTP Server Reply This stage generates and sends a reply to the spotter, based on an event containing a HTTP connection ID and a frame.

1.2 Adaptive Feedback Algorithms

In addition to the stage controller and event handler processes, the system also has a *master controller* process that communicates with the stage controller process via a shared memory segment, synchronized with mutual-exclusion locks. It monitors the number of events in each stage's queue, and the average rate at which events are being processed by each stage.

The master controller controls the number of event handler processes that each stage is allowed to use. It allocates these processes to each stage proportionally to the amount of work in the queue and the length of processing time required per event. Since the kernel schedules processes in a round-robin fashion, this allocates more processing time to the stages that need it most. This minimizes the effect of bottlenecks, when one stage runs considerably slower than the others.

The master controller also implements load-shedding. When the measured rate at which data is being processed by the camera-processing pipeline falls below the rate at which data becomes available, the master controller instructs the Camera HTTP Accumulator stage to begin shedding load at a certain rate. The Camera HTTP Accumulator stage then begins dropping video segments from each camera, in a fair manner that ensures that the same proportion of frames from each camera are processed.

2 Analysis

The SEDA-based design presented in this paper achieves the design goals of fault isolation, graceful degradation, and high performance.

Fault isolation is achieved in two ways. First, event handler processes operate in their own address spaces, so bugs in individual event handlers (including the known-buggy TRANSCODE and DETECT-ANOMALY functions) cannot interfere with other parts of the system. Furthermore, communication between stages is restricted to the narrow interface for inserting events into another stage's queue. This effectively isolates each stage from the others.

The adaptive feedback algorithms described above ensure that this design achieves graceful degradation under load. As the system becomes overloaded, the master controller will shift the allocation of system resources to bottleneck stages. The load-shedding algorithm causes video segments to be dropped when necessary. This load shedding is performed at the beginning of the pipeline, minimizing wasted work, and is performed in a fair manner, dropping segments from each camera equally so that all cameras continue to be monitored regularly.

High performance is achieved through the use of the SEDA architecture, with its event-driven concurrency and adaptive algorithms. This architecture has proven effective in implementing other high-performance applications, such as a web server. Furthermore, the architecture scales easily to a multi-CPU system, or a network of multiple systems.

A Concurrently-Staged Design for Surveillance Monitoring

Dan Ports
March 18, 2004

6.033 Design Project 1

Recitation #9: Karger/Lesniewski-Laas

Contents

1 Introduction **1**

2 Design Overview **1**
2.1 The SEDA Model . . . 1
2.2 Stages . . . 2

3 Design Description **2**
3.1 Operating System Requirements . . . 2
3.2 Master Controller . . . 3
3.3 Common Procedures for Queue Management . . . 4
3.3.1 The Protocol . . . 4
3.3.2 Transmitting an Event . . . 4
3.3.3 Receiving Events . . . 5
3.4 Camera HTTP Accumulator . . . 6
3.5 Transcoder . . . 6
3.6 Anomaly Detection . . . 6
3.7 Assignment . . . 8
3.8 HTTP Server Accumulator . . . 8
3.9 HTTP Server Parser . . . 9
3.10 HTTP Server Reply . . . 9

4 Analysis **9**
4.1 Fault Isolation . . . 10
4.2 Graceful Degradation . . . 10
4.3 Performance and Scalability . . . 11

5 Conclusion **11**

List of Figures

1 Data flow between stages . . . 1
2 Master controller pseudocode . . . 5
3 Transcoder stage pseudocode . . . 7
4 Anomaly Detection stage pseudocode . . . 7

1 Introduction

This design implements a Surveillance-At-Home system that achieves the goals of fault isolation, graceful degradation under load, and performance. We accomplish these goals using the Scalable Event-Driven Architecture, dividing the processing into a series of stages that operate concurrently. By using address space separation and strictly limiting interaction between stages, the system achieves fault isolation. A controller monitors system performance uses a feedback algorithm to dynamically reallocate system resources and implement load shedding, providing the system with high performance and graceful degradation.

2 Design Overview

This report presents a design for the central computer of a video surveillance system that uses both image processing and human spotters to identify suspicious activity. The system receives video streams from many cameras simultaneously, and divides them into short segments. Each segment is transcoded into a standard image format, and then passed through a vision module that identifies frames with potential suspicious activity. Images that have received a high threat index are stored in a queue. A web server responds to HTTP requests from human spotters by sending an image from the queue. If the human spotters agree that the image is suspicious, an appropriate action is taken, such as notifying the operator.

2.1 The SEDA Model

This design is based on the *Staged Event-Driven Architecture* (SEDA) model introduced by Welsh [1]. This architecture divides the processing of requests into *stages*. Each stage consists of a *stage controller* that maintains a queue of requests ("*events*") to be processed, and *event handler* processes that are invoked by the controller to handle each event. Multiple handler processes for the same stage can operate concurrently, up to a maximum number for the stage that is determined dynamically. Individual stages only communicate with each other by inserting events into each other's event queue through a well-defined internal interface.

In this system, an event represents either a partially-processed segment of video data from one camera, or an incoming HTTP request from one of the spotters' web browsers. The processing sequence can be viewed as a pipeline: events are processed, then transmitted to the next stage. The data flow of events through stages is shown in Figure 1. Because incoming network traffic does not naturally arrive conveniently partitioned into SEDA events, we create *interface stages* that receive data from the network, partition it into appropriately-sized blocks (either one second of video data, or a single HTTP request), then formats it as an event and passes it to the next stage.

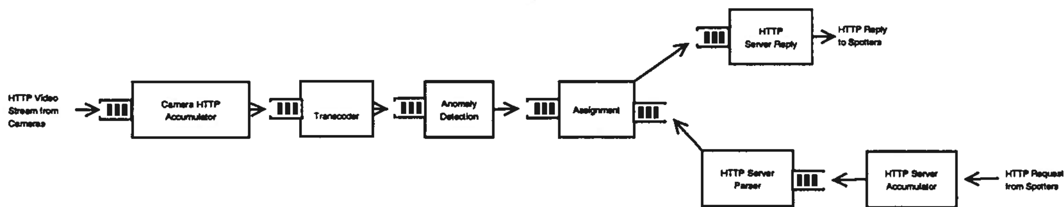

Figure 1: Data flow between stages

Stages communicate with each other by passing events. Once an event handler process has completed its processing for a particular event, it passes the result to the next stage's controller. As the stages operate in different address spaces, inter-process communication must be achieved through a sockets-based interface. Each stage controller process listens on an internal port. To

enqueue an event, event handler processes establish a connection to the next stage's controller via this port, and send the event data in the proper marshalled form for the next stage.

In addition to the stage controllers that manage queues for each stage and start event handler processes, a master controller process oversees the stage controllers. The master controller monitors the number of event handlers currently running at each stage and the queue sizes. It adaptively changes the maximum number of processes at each stage to reflect the demands placed on the system by that stage's processing: by starting more processes at a given stage, the kernel's preemptive scheduler allocates more time to that stage. By dynamically reallocating processing power, the master controller eliminates bottlenecks as they arrive. In addition, the master controller monitors the system load, and orders the stages to begin shedding load when the system is overloaded. These feedback-based algorithms make graceful degradation possible.

2.2 Stages

Seven discrete stages are used in the processing pipeline. These stages are described in detail in Section 3, but we present a brief overview here:

Camera HTTP Accumulator This interface stage receives data from the cameras via HTTP, and puts it in event form. It stores incoming data in a buffer associated with each camera. When one second of video data has been received, the contents of the buffer are passed as an event to the transcoder stage. This stage also opens the HTTP connections to the cameras.

Transcoder This stage converts the stream chunks received from the cameras into the standard JPEG format. It is a wrapper for the TRANSCODE procedure.

Anomaly Detection The Anomaly Detection stage is a wrapper for the DETECT-ANOMALY procedure. It takes in a transcoded video stream, and outputs a threat index and the single most suspicious frame.

Assignment This stage handles the assignment of video frames to spotters. It has two input queues: one receiving scored frames from the Anomaly Detection stage, and one receiving HTTP requests from the HTTP Server Parser. It maintains an internal state table of recently arrived frames, and uses this to assign frames to spotters based on their threat index. The pairing of the HTTP request and selected frame is enqueued for the HTTP Server Reply stage.

HTTP Server Accumulator The HTTP Server Accumulator is an interface stage that listens on TCP port 80 for HTTP connections from the spotters, and buffers incoming data. Once a full HTTP request is received, it encodes it as an event and passes it to the HTTP Server Parser stage.

HTTP Server Parser This stage parses an incoming HTTP request from a spotter's web browser. It identifies whether the request is for a new image, or a ranking for a previous image, and passes it to the Assignment stage.

HTTP Server Reply This stage takes in an event containing a HTTP connection ID and a frame, and generates and sends a reply to the spotter's web browser.

3 Design Description

3.1 Operating System Requirements

This system design is intended to operate using standard PC hardware and an operating system similar to Unix. We describe procedures using Unix system call notation. Some of the features that we require of the kernel include:

- Preemptive scheduling with a round-robin scheduler is required to ensure that processor time is divided equitably among all threads.
- A sockets-based communication API is used. In particular, we require a non-blocking interface and the SELECT system call.
- We must be able to use sockets to communicate between processes on the same machine.
- The ability to create a shared memory segment between selected processes is required. Mutual-exclusion locks ("mutexes") are used for synchronizing access to shared memory.

3.2 Master Controller

The master controller module has two primary functions. First, it creates the seven stage controller processes when the system is starting up. Second, when the system is operating, it monitors actual system performance and uses a feedback-based algorithm to adaptively reallocate processor time in order to maintain optimum performance.

The master controller and stage controller processes communicate via a memory segment they share (but the event handlers do not). This shared memory segment is used for exchanging measurements and parameters. Access to each shared variable is controlled by a mutual-exclusion lock ("mutex"), which ensures that updates are synchronized between processes. The following variables are included in the shared memory segment:

- *load-shedding-fraction*: the fraction of incoming data to be dropped, set by the master controller
- For each stage, *max-event-handlers*[*stage*]: the maximum number of event handler processes that can be invoked by each stage, set by the master controller
- For each stage, *queue-length*[*stage*]: the number of events currently in the stage's queue, set by the stage controller
- For each stage, *events-processed*[*stage*]: a counter variable for the number of events processed by the stage, incremented by the stage controller, and measured and reset every second by the master controller.

Creating the stage controller processes is a straightforward procedure. When the master controller is started, it first allocates the shared memory segment and mutexes described above. Then, for each of the seven stages, it creates a new process with the FORK system call. Each of the child processes then executes the appropriate stage controller code. The master process then proceeds to its monitoring loop.

The procedure for monitoring system performance is more complex. The goal of this procedure is to adjust two parameters based on measurement of system performance: the maximum number of event handler processes to be used by each stage, and the fraction of incoming data to drop.

The master controller monitors the performance only of the stages in the *camera pipeline*: the Transcoder, Anomaly Detection, and Assignment stages, and these are the only stages it adjusts the maximum number of event handlers for. The stages in the HTTP server operate with a fixed maximum number of event handlers, because we predict that the camera stream processing will be responsible for the majority of the load, not the HTTP server. Hence, when we refer to "all stages" being monitored and adjusted, we mean "all stages in the camera pipeline".

Since processor time is scheduled preemptively using a round-robin scheduler, the percentage of processor time devoted to each stage is proportional to the number of event handler processes executing it. This allows bottlenecks to be avoided. Normally, if the system is under load and one stage is operating especially slowly, it will act as a bottleneck, and its queue will fill, causing the

pipeline to back up at this stage. Allocating more processes to this stage does not increase the total processing power of the system; however, it does ensure that more processing time is spent on the bottleneck stage *relative to the other stages*. This reallocation ensures that all stages are processing data at approximately the same rate (in terms of video segments per second), so no stage remains a bottleneck.

The algorithm used to determine the number of processes allocated to each stage is based on computing the *queued workload* for each stage, which we define to be the product of the number of events currently in that stage's queue and the average amount of time required to process each event. The average amount of processing time per event is computed by measuring the number of events processed by the stage each second from the shared memory buffer, and calculating a moving average over the last 15 seconds. The stage with the lowest queued workload is assigned one event handler process, and larger number of event handler processes are assigned to the other processes in proportion to their relative queued workloads.

Although reallocating processor time by adjusting the per-stage process count allows bottlenecks to be avoided, it does not deal with the case when the system is overloaded: when the system simply cannot process the video data fast enough to keep up with the incoming streams. In this case, there is no other option but to start shedding load by dropping segments of the incoming streams. Since it would be wasteful to spend time processing a segment only to discard it later, load shedding is performed at the earliest stage of the pipeline: by the Camera HTTP Accumulator.

Load shedding is accomplished by automatically adjusting a parameter that instructs the Camera HTTP Accumulator stage as to how much load should be discarded. The critical insight here is that the amount of data to be processed (i.e. not discarded) should be kept as close as possible to the amount of data that the system is actually able to process. Thus, we define the *relative rate ratio*, which is equal to the number of video segments that are processed each second divided by the number of video segments that arrive each second (which is equal to the number of cameras, since one segment is one second of data from one camera). The number of segments processed per second can be measured by using a moving average of the number of events handled by the Assignment process each second. The proportion of video frames to be dropped will be 1 minus the relative rate ratio.

3.3 Common Procedures for Queue Management

Communication between stages in this architecture takes place only through events in queues: one stage generates an event containing the data for the next stage to process, and then transmits the event to the next stage's queue. Since this design pattern is ubiquitous in this system, we define a standard protocol for placing events in another stage's queue, and a pair of common procedures that implement this protocol. These procedures are used in nearly every module.

3.3.1 The Protocol

We define a protocol that makes possible a single operation: one stage can place an event in another stage's queue. To perform this operation, the stages communicate via sockets-based connections within the system. A datagram-based protocol such as UDP is used; since the connections are only internal to the system, we do not need to worry about packet loss or corruption. Each stage controller process listens on a fixed internal port, which we refer to as PORTS[*stage*]. To enqueue an event, the stage creating the event connects to the destination stage on port PORTS[*destination-stage*]. It then transmits the event in a marshalled form: each field in the event is transmitted in sequence.

3.3.2 Transmitting an Event

We define the common procedure SEND-EVENT-TO-STAGE for placing an event in another stage's queue, according to the protocol above:

```
MASTER-CONTROLLER()
 1  allocate shared memory segment and mutexes
 2  for stage-controller in all-stage-controllers
 3       do if FORK() = 0
 4            then EXECUTE stage-controller
 5
 6  while (1)
 7       do SLEEP(1 second)
 8          for stage in camera-pipeline-stages
 9               do ACQUIRE-MUTEX(lock[stage])
10                  events-processed-average[stage] ← UPDATE-MOVING-AVERAGE(events-processed[stage])
11                  events-processed[stage] ← 0
12                  queued-workload[stage] ← queue-length[stage] / events-processed-average[stage]
13                  RELEASE-MUTEX(lock[stage])
14          load-shedding-fraction ← 1 − (events-processed-average[ASSIGNMENT]/MAX-CAMERAS)
15          for stage in all-stages
16               do
17                  max-event-handlers[stage] ← queued-workload[stage] / min{queued-workload}
```

so each stage controller is a different process / addr space

Figure 2: Master controller pseudocode

```
SEND-EVENT-TO-STAGE(event, destination-stage)
1  port ← PORTS[destination-stage]
2  buffer ← MARSHAL(event)
3  SEND(port, buffer)
4  ACQUIRE-MUTEX(lock[destination-stage])
5  queue-length[destination-stage] ← queue-length[destination-stage] + 1
6  RELEASE-MUTEX(lock[destination-stage])
```

3.3.3 Receiving Events

We define the common procedure RECEIVE-EVENTS-INTO-QUEUE for receiving events from other stages and placing them into a queue. It takes as arguments the file descriptor for the socket on which the events are to be received, and the queue to which the events are to be added. This procedure is designed to be called from a stage controller process's event loop.

```
RECEIVE-EVENTS-INTO-QUEUE(fd, queue)
 1  new-queue ← queue
 2  while (DATA-AVAILABLE-ON-FD(fd))
 3       do buffer ← RECV(fd)
 4          if (IS-VALID-EVENT(buffer))
 5            then event ← UNMARSHAL(buffer)
 6                 new-queue ← ADD-TO-QUEUE(new-queue, event)
 7          ACQUIRE-MUTEX(lock[stage])
 8          queue-length[stage] ← queue-length[stage] − 1
 9          RELEASE-MUTEX(lock[stage])
10  return new-queue
```

with this approach, queue-length variable only measures # events "in-flight" in socket; doesn't count items in queue variable itself. Why is this inaccuracy OK?

3.4 Camera HTTP Accumulator

The Camera HTTP Accumulator stage is the interface stage that handles the HTTP connections to the cameras. Upon startup, it opens a connection to each camera, sends the HTTP GET request to initiate the video stream, and allocates a buffer for each connection.

Unlike most of the other stages, it does not use event handler processes. Instead, after startup, it operates as a loop. During each iteration of the loop, it (waits for one second,) then reads from each HTTP connection into its associated buffer. The *non-blocking* variant of the READ system call is used.

[Handwritten note: there's a good reason for this, but worth explaining,]

Once data is read from each connection, the stage determines what to do with the data. If the system is under normal load, all data is sent to the next stage. If the system is overloaded, data from n connections is discarded, where

$$n = \textit{load-shedding-fraction} \times \text{NUM-CAMERAS}$$

The stages chooses which data to discard by maintaining a table of when each connection's data was last processed. It then chooses the (NUM-CAMERAS $- n$) least-recently-processed buffers to process, sends their data to the next stage, and discards the rest. The last-processed-time table is updated, and all buffers are cleared in preparation for the next read sequence.

Data from each buffer that is not discarded is marshalled into event form: the raw data is associated with the camera ID number, and encapsulated in an event data structure. This event is then passed to the Transcoder stage using the standard SEND-EVENT-TO-STAGE procedure.

[Handwritten note: Not terrible, but if load suddenly drops ~~after handling~~ (fully handle NUMCAMS-n segment in << 1 second) you may be left twiddling your thumbs for one second. You have enough RAM to hold a few seconds of data in reserve in case you end up with time to process it.]

3.5 Transcoder

The Transcoder stage receives raw data from the Camera HTTP Accumulator stage, and converts it into a sequence of JPEG images, a standard format. The conversion takes place using the provided TRANSCODE function.

This stage consists of a stage controller that receives events in its queue, and event handler processes that essentially wrap the TRANSCODE function. The stage controller operates as a loop. During each iteration of the loop, it uses the standard RECEIVE-EVENTS-INTO-QUEUE function to accept any events that may have been sent to it on its port and place them into the queue. If the queue contains an event and the number of processes currently running is less than the maximum number of processes allowed for this stage, a new event handler process is created in a new address space.

The event handler process takes the raw data from the event, and calls the TRANSCODE function to transcode it. Once this transcoding has completed, it then creates a new event containing the transcoded images and the camera ID, and sends it to the Anomaly Detection stage using the standard SEND-EVENT-TO-STAGE procedure.

3.6 Anomaly Detection

The Anomaly Detection stage uses a provided vision procedure (DETECT-ANOMALY) to identify suspicious frames in the data supplied from the transcoder stage, and determine a threat index between 0 and 1 that indicates how suspicious the frame is.

The stage operates essentially like the Transcoder stage: it is a simple wrapper for the DETECT-ANOMALY function. The output from this function is encoded into an event and sent to the Assignment stage.

We omit the pseudocode for the ANOMALY-DETECTION-STAGE-CONTROLLER procedure, as it is virtually identical to the analogous procedure for the Transcoder module.

```
TRANSCODER-STAGE-CONTROLLER
1  while (1)
2       do queue ← RECEIVE-EVENTS-INTO-QUEUE
3          while (queue is not empty and event-handlers-running < max-event-handlers[TRANSCODER])
4               do ACQUIRE-MUTEX(lock[TRANSCODER])
5                  events-processed[TRANSCODER] ← events-processed[TRANSCODER] + 1
6                  RELEASE-MUTEX(lock[TRANSCODER])
7                  event ← REMOVE-FROM-QUEUE(queue)
8                  if FORK() = 0
9                    then EXEC(TRANSCODER-EVENT-HANDLER)
```

```
TRANSCODER-EVENT-HANDLER
1  output-buffer ← TRANSCODE(event.data)
2  new-event.transcoded-data ← output-buffer
3  new-event.camera-id ← event.camera-id
4  SEND-EVENT-TO-STAGE(new-event, ANOMALY-DETECTION)
```

Figure 3: Transcoder stage pseudocode

What if these "crash" by infinite looping? Would some timer that kills them be desirable?

```
ANOMALY-DETECTION-EVENT-HANDLER
1  vision-result ← DETECT-ANOMALY(event.transcoded-data)
2  new-event.index ← vision-result.threat
3  new-event.frame ← event.transcoded-data[vision-result.frame-pointer]
4  new-event.camera-id ← event.camera-id
5  SEND-EVENT-TO-STAGE(new-event, ASSIGNMENT)
```

Figure 4: Anomaly Detection stage pseudocode

3.7 Assignment

The Assignment stage assigns video frames to spotters. It receives video frames as events from the Anomaly Detection module, and requests for images from the HTTP Server Parser stage. Hence, unlike the other stages, it must maintain two queues for receiving a different type of events from each of these two stages. This stage also differs from the other stages in that it does not use event handler threads; all processing takes place in the stage controller process. This design choice was made because the Assignment stage must maintain extensive state: it must keep a table of recently-received suspicious frames that are waiting to be assigned to spotters.

We naturally want to ensure that the most suspicious frames are the first ones directed to the spotters. However, it is also desirable that suspicious frames from all cameras are processed; even if one camera is generating many very suspicious frames, suspicious frames from other cameras should also be processed. Hence, we maintain another table associating each camera ID with the time a frame from it was last processed. We generate a ranking for each frame in the queue of how long it has been since a frame from that camera was last processed, and then compute a *weight function*

$$w(\textit{frame}) = \alpha(\textit{frame}\ .\text{threat-index}) + \beta(\textit{frame}\ .\text{ranking})$$

The constants α and β indicate how heavily we wish to weight the threat index relative to the time-since-last-processed ranking. We leave the exact numeric values of these parameters unspecified, since they depend on the range of variation of the values returned by the DETECT-ANOMALY function. We also define a constant ϵ that represents the minimum threat index required for a frame to be considered suspicious at all; frames whose threat index is less than ϵ will be simply discarded instead of being assigned to spotters. We leave the value of ϵ unspecified for the same reason.

The stage controller process operates as a loop. On each iteration of the loop, it uses the RECEIVE-EVENTS-INTO-QUEUE procedure *twice*, to receive events from the sockets corresponding to each of its input queues, and places the resulting events into separate queue variables.

For each event received from the Anomaly Detection stage, the frame, threat index, and camera ID are placed in the table of received frames, if the threat index is greater than ϵ. If the table grows beyond some pre-specified maximum size (which we anticipate to be at least one order of magnitude larger than the number of spotters), the table is truncated to the maximum by removing the frames whose weight function is lowest. isn't this a kind of late-dropping for congestion control? Does that matter?

For each event received from the HTTP Server Parser stage, we determine whether the event represents a request for a new frame, or a spotter's evaluation of a frame.

If the event is a request for a new frame, the Assignment stage controller selects the frame from the table that has the highest value of the weight function. It removes this frame from the table and places it in a new event. This new event will also contain the camera ID number from, and the HTTP stream ID from the event received from the HTTP Server Parser. The resulting event is sent to the HTTP Server Reply stage. The table of time-since-last-processed rankings for the cameras is then updated, and the weight functions for all frames recalculated.

If instead the event is an evaluation of a frame, the system checks whether the spotter evaluated the frame as suspicious. If so, the system takes the appropriate action, i.e. notifying the operator.

DP asked for second opinion if first evaluation uncertain

3.8 HTTP Server Accumulator

The HTTP Server Accumulator is an interface stage that performs a function analogous to the Camera HTTP Accumulator. It listens on port 80 for incoming connections, reads from each open connection into a buffer, and converts incoming requests to events. However, it differs from the Camera HTTP Accumulator in three main respects: it does not perform load-shedding, it does not read from a fixed set of camera connections but rather from whatever clients have connected to the server, and it converts incoming data into events not by dividing it into one-second segments but rather by detecting a HTTP request.

Like the Camera HTTP Accumulator, this stage operates as a loop rather than using event handler processes. During each iteration of the loop, the SELECT system call is used to determine which connections have data available. For each such connection, the non-blocking variant of the READ system call is used to read all available data into its associated buffer. This stage then checks the received data for a double-carriage-return sequence that indicates the end of a HTTP request; if one exists, it packages the connection file descriptor and the received data into an event and sends it to the HTTP Server Parser stage.

you can't pass fds between two distinct processes (the meaning of the integer is process dependent)

If the SELECT system call indicates that a new connection has been opened, the file descriptor for this new connection will be added to the set of file descriptors for open connections that is passed to SELECT. In addition, a new buffer is allocated to hold the data for this connection.

3.9 HTTP Server Parser

The HTTP Server Parser stage receives raw data from the HTTP Server Accumulator stage, and parses it as a HTTP request.

This stage operates using a stage controller and event handler threads, much like the Transcoder or Anomaly Detection stages. However, rather than dynamically determine the maximum number of event handler processes that can be operating concurrently, we use a fixed maximum limit which we anticipate to be on the order of 10. We use a fixed limit rather than a load-adaptive algorithm for simplicity, since we anticipate that the HTTP server portion of this system will not be responsible for nearly as much of the load as the camera-processing stages.

The stage controller loop receives events from the stage's socket and stores them in the queue, and invokes event handler processes when events are present in the queue and able to be processed.

The event handler processes parse the received data using the HTTP protocol. If the request is a valid HTTP request, the handler determines whether it is a request for a new frame, or a spotter's response to a previous frame. It then creates a new event consisting of this type of request, the connection's file descriptor. If the request is a response to a previous frame, the event also includes the frame ID and the spotter's evaluation of it (suspicious or not suspicious). This event is sent to the Assignment process. If instead the request is invalid, the event handler sends an error message on the HTTP connection.

3.10 HTTP Server Reply

The HTTP Server Reply stage sends replies to HTTP requests for new images. It receives an event from the Assignment stage that represents a pairing of a HTTP request with a frame that has been assigned to it, and sends that frame to that connection.

This stage also operates using a stage controller and event handler threads. The stage controller loop receives events from the stage's socket and stores them in the queue, and invokes event handler processes when events are present in the queue and able to be processed. We use the same fixed limit for the maximum number of event handler processes as in the HTTP Server Parser stage, for the same reasoning.

The event handler processes take the video frame contained in the event, and generate a HTTP reply encoding a HTML page containing this image. This reply is then sent to the spotter's web client via the connection ID included in the event.

4 Analysis

The SEDA-based design presented in this paper achieves the three major design objectives for this system: *fault isolation*, *graceful degradation under load*, and *performance*.

4.1 Fault Isolation

The principal design objective for this system is fault isolation: no module should be able to cause the entire system to crash. In particular, the supplied TRANSCODE and DETECT-ANOMALY procedures are known to be unreliable.

The SEDA model naturally achieves fault isolation. The system is divided into discrete stages, and each stage can only communicate with the other stages through the narrowly-defined interface for placing events in queues. Since each event transmitted to a stage is validated before being enqueued, this prevents any failure of one stage from affecting any others.

Moreover, the stages that use event handler processes, including the Transcoder and Anomaly Detection stages, invoke these processes in their own address spaces. This not only prevents a malfunctioning event handler from causing a failure in a different stage, it prevents the failure from affecting other event handlers in the *same* stage. This means that if, for example, the DETECT-ANOMALY causes a spurious memory write, the event handler will simply be shut down, causing the event that triggered the error to be lost. Losing this event is the desired effect, because attempting to repeat the processing may cause the same error. All other parts of the system, including the other events in the Anomaly Detection stage's queue, and any other event handler processes running the DETECT-ANOMALY procedure will continue to operate as though nothing happened.

4.2 Graceful Degradation

Another critical design objective is graceful degradation under heavy load. Though the processor should be powerful enough to handle the normal load presented to the system, brief periods of unexpectedly high load may present a higher load than the system can handle. In this case, the system will be forced to discard some incoming data. It should continue to process data from each connection, ensuring that no camera is starved for attention.

Our SEDA design with adaptive process allocation and load shedding accomplishes this objective. By monitoring the system state and measuring current performance, the feedback algorithm in the master controller is able to tailor the system parameters in order to achieve optimal resource allocation.

The following scenario demonstrates how the system can respond to an unexpected increase in load:

Suppose the system is monitoring 1000 cameras when 100 of the cameras suddenly begin to observe much more motion than usual, causing the DETECT-ANOMALY procedure to run very slowly. Since the Transcoder and the other stages will still be running at the same rate, the Anomaly Detection stage's queue will grow as it fills up more rapidly than events can be processed.

When the master controller process takes its next set of measurements, it will observe that the queue of the Anomaly Detection stage has grown, and the average rate at which events are processed has also increased. Hence, the queue workload for this stage will be substantially higher than normal. In response, the master controller will raise the Anomaly Detection stage's maximum-event-handler allocation. This will allow the Anomaly Detection stage to start more event handler processes, causing the kernel scheduler to allocate more time to the stage. Thus, the queue will be filled more slowly because the Transcoder stage will process events at a slower rate, and emptied more rapidly because the Anomaly Detection stage will process events at a faster rate. The feedback algorithm will continue to adjust the maximum-event-handler allocations for each stage in order to achieve a dynamic equilibrium. This eliminates the bottleneck at the Anomaly Detection stage.

Though the process reallocation accounts for the bottleneck, it does not change the fact that the system does not have enough processing power to perform the processing for every camera and still keep up with the input streams. By monitoring the rate at which events are processed in the queue between the Anomaly Detection stage and the Assignment stage, the master controller observes that the camera pipeline is not processing segments as fast as they are arriving from the

network. In response, it instructs the Camera HTTP Accumulator stage to begin shedding load, proportionally to the difference between the rate at which data is arriving and the rate at which it is being processed. The Camera HTTP Accumulator begins dropping segments from some of the cameras. It does so in a manner that ensures that segments are dropped in equal proportion for each of the cameras. As a result, the system continues to process input from each camera, though some frames are being dropped.

When the load presented by the cameras returns to normal, the master controller will observe that the rate at which the system can process data is increasing. It will then direct the Camera HTTP Accumulator stage to decrease the load-shedding fraction until the full data stream is once again being processed.

4.3 Performance and Scalability

This design leverages the inherent performance advantages of the SEDA architecture. Through the use of massive event-driven concurrency and adaptive load shedding such as that which we have described here, applications developed with SEDA have been able to achieve impressive performance results. The designers of the SEDA architecture presented a high-performance web server, Haboob, that was able to outperform the industry-standard Apache web server and the high-performance Flash web server in performance tests, particularly under high load [2]. This accomplishment is particularly impressive in light of the fact that their web server was written in Java and contained no optimizations other than those inherent to SEDA. Though no performance testing has been done on our surveillance monitoring system, we anticipate similar results.

The design can also be easily adapted to a larger scale with only a few modifications. In particular, though we assume a single-CPU computer for this design, the highly-concurrent nature of SEDA makes it well-suited for a multiple-CPU system. In fact, its full performance advantages cannot be realized without using multiple CPUs. It is also possible to adapt the system to run stages on *separate* computer systems: since communication between processes takes place through a sockets-based interface, the other stage controllers may just as well be on a different computer connected via a network. Clearly some modifications would need to be made to the details of the design; for example, the master controller and stage controllers could no longer communicate via shared memory. However, most of the design architecture can be used without major modifications.

5 Conclusion

We have presented a design for a surveillance monitoring system that uses the SEDA architecture. The key features of our approach are a decomposition of the process into stages, concurrent event-driving processing of each stage, narrowly-defined interfaces for communications between stages, and adaptive algorithms for allocating resources and shedding load. These features allow our system to achieve the design objectives of fault isolation, high performance, graceful degradation under heavy load, and scalability.

Acknowledgements

The architecture for this implementation is based on the SEDA architecture developed by Matt Welsh as part of his Ph.D. thesis research; more information is available at `http://www.eecs.harvard.edu/ mdw/proj/seda/`.

Austin Clements provided many invaluable insights during the design process. Chris Lesniewski-Laas also provided helpful comments.

References

[1] M. Welsh, "An architecture for highly concurrent, well-conditioned internet services," Ph.D. dissertation, University of California, Berkeley, Aug. 2002.

[2] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, Oct. 2001.