Implementation of a Collaborative Observatory for Natural Environments

Andrew Barton Christian Dahl

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-71
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-71.html

May 19, 2007

Copyright © 2007, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Implementation of a Collaborative Observatory for Natural Environments

Andrew Dahl
UC Berkeley, Spring 2007

TABLE OF CONTENTS

1 INTRODUCTION

The scientific study of animals in their natural habitat can be a difficult task requiring weeks or even months of vigilant observation. It may additionally be costly, dangerous, and a lonely experience for scientists. The CONE (Collaborative Observatory for Natural Environments) project attempts to solve these problems by providing a new class of teleoperated/autonomous robotic "observatories". Using emerging advances in networked robotic cameras and related technologies, these observatories give animal enthusiasts and scientists the ability via the Internet to remotely access and observe animals from a safe and convenient location. In addition to being collaborative in nature, the observatories include the ability to record and index animal activity for scientific study. The CONE project was proposed by Ken Goldberg, professor of engineering at UC Berkeley, and Dez Song, professor of computer science at Texas A&M in 2005 [1][10]. It is a 3-year collaborative effort by computer scientists and engineers from both schools funded by the National Science Foundation.

This paper describes the implementation of the 4th and most recent prototype system produced by the CONE project called CONE Sutro Forest (CONE-SF). The purpose of this system is to allow bird enthusiasts and scientists to observe birds in their natural habitat. CONE-SF was installed in Spring 2007 at the private residence of craigslist founder Craig Newmarks in San Francisco overlooking the Sutro Forest. It is a complete revamp of previous CONE systems, which were plagued by issues of instability and complexity making them hard to use and modify. CONE-SF was designed using an entirely new code-base to create a simpler and more robust system. Focus was also put on improving the collaborative experience and CONE-SF was therefore set up as game in which players earn points by taking live photos and classifying birds.

Creating the CONE-SF system was a team effort that involved 11 students and professors and required many components to be designed, built, and connected together. In this paper I choose to focus on my contributions to the project, but still describe how the system works as a whole. The next section will give an overview of the CONE system. Experiences gained from previous CONE prototypes will then be discussed together with a list of resulting goals for CONE-SF. A brief discussion of the design & implementation choices made for the main parts of the system will then be presented. The rest of the paper will then focus on the parts of the system that I worked on and related work. Finally concluding remarks are made and future work is discussed.

2 SYSTEM OVERVIEW

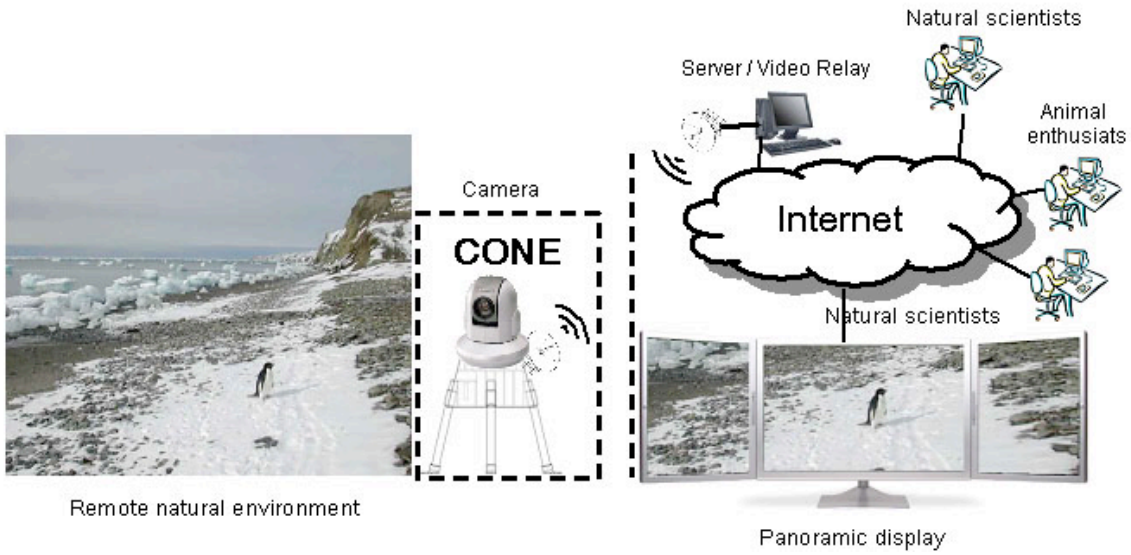


Figure 1. CONE system overview

The key component of the CONE system is the networked robotic camera, which is set up in a remote location to make it possible to observe animals in their natural environment. The camera is connected via the Internet to a server that is responsible for controlling the camera. Scientists and casual viewers then use a web browser to access a Java-based client that connects either directly to the camera or through a video relay server to receive a live video feed from the camera. Clients also connect to the server, which allows them to request that the camera change its viewpoint. To be able to make these requests the client interface contains a panorama that represents the viewable area of the robotic camera. Viewers draw frames with their mouse on this panorama, which are then turned into requests for the camera to move to and display the area covered by the frame. Upon getting requests from clients to move the camera, the server calculates and issues a command to the camera to move to the viewpoint that will maximize total viewer satisfaction. This is what makes the CONE system collaborative. Instead of using a simple queue to move the camera to requested frames in turn, the system groups requests and asks: “how can I make the largest number of users happy”. This creates better utilization of camera resources and scales with the number of viewers, since multiple viewers can be satisfied with a single movement of the camera. An example of a set of user frame request and the corresponding camera frame is given below in figure 2. The original algorithm used to calculate the optimal camera frame was developed by Dezhen Song and Ken Goldberg [3]. In the CONE-SF implementation a slight variation on this algorithm is used, which will be discussed in section on CONE-SF design and implementation. In addition to making frame requests, the client also allows viewers to see who else logged into the system and to take snapshots of the current scene displayed. These snapshots are presented on a web site for the project together with the Java-based client.

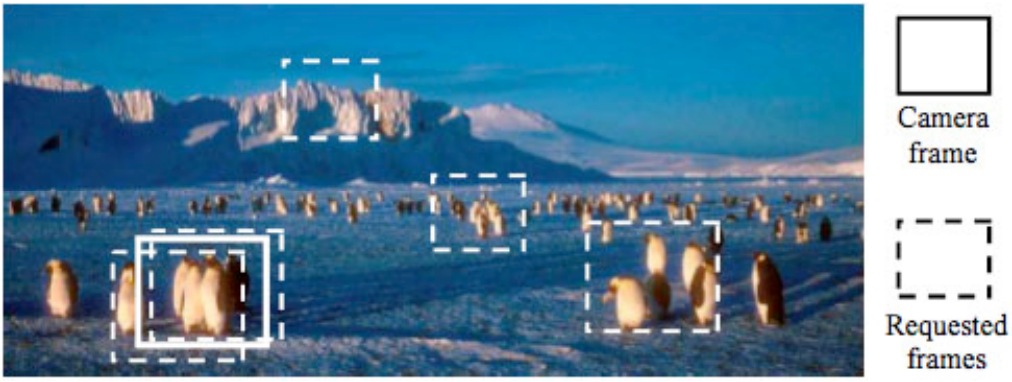


Figure 2. Example of frame requests and resulting camera frame

3 PREVIOUS EXPERIENCE

The CONE project has generated several prototype systems in the past. These prototypes were deployed in different settings and were all used by a varied group of scientists and casual viewers, including CONE project team members. Serious problems were encountered with these prototypes, in large part due to the technology driven nature of the systems. Rather than provide a reliable, intuitive system to viewers, the focus had been on testing the capabilities of emerging robotic camera technology and algorithms. Some of the problems encountered included the system being unresponsive or crashing when experiencing moderate load and complaints about a lack of a collaborative feeling. From a software development perspective the previous prototypes were also difficult to modify and debug due to the systems requiring a host of interrelated programs to run.

The problems of previous prototypes persuaded us to start from scratch and completely revamp the CONE system in designing and implementing CONE-SF. To aid us in this we gathered feedback from people who had experienced the problems of previous deployments. From this feedback we developed a list of possible requirements for a new system. We broke this list down by intended user group, as this would help our team to frame our observations and provide specific solutions for each group. Furthermore, for each group the problem found with the system or desired feature, was listed together with the possible resulting requirements for a new CONE system.

Table 1. Feedback from previous CONE deployments

Intended User	Problem found / Desired Feature	Requirement
Scientists & Casual users	System would crash when under moderate load (> 5 users)	CONE scales to many users (20+) gracefully
	System did not always respond immediately to user requests	CONE provides some notification that a request has been entered and will be satisfied
	Users should be able to see their requested frame even after the camera has moved on	CONE allows users to capture still images on demand
	System only allowed for short captions on images by the capturer of	CONE allows users to provide detailed text to accompany images,

	the image	even ones they did not capture themselves
	Seeing other people's requests would help disambiguate camera movement	CONE clients show all the user requests
	System must feel collaborative rather than competitive	CONE provides several ways to collaborate observation
	System must feel collaborative rather than competitive	CONE provides several ways to collaborate observation
	Captured images would be improved with detailed observational comments	CONE provides a space for users to review images and store comments and notes
Observatory Administrators	System required 3-4 separate programs to be running in order to function	CONE runs in a single process
	System relied on Apache modules which have to be compiled in to the Apache server	CONE does not require compilation at install
	System must be quick to install for users unfamiliar to the system	CONE can be set up in <5 hours
	System panorama had to be custom-generated manually	CONE automatically generates its own panorama image
	System camera had to be manually calibrated by blind guess-and-test	CONE automatically calibrates the system to the camera
Ourselves	System must be easy to incrementally add modules to	CONE has a fully modular design
	System must allow the swapping of components	CONE makes liberal use of code abstraction layers to help code swapping

4 CONE-SF GOALS

Taking the feedback from previous experience into consideration it was clear that we needed to create a system that was focused more on the user experience, while still allowing for easy modification and setup. The old prototypes had essentially been a way for multiple viewers to control a single camera. In a new system we wanted to move away from this paradigm and instead create a space for "online collaborative observation". To accomplish this, any new system would have to extend beyond the basic camera control and video display and make the user feel more engaged. We decided that the key to the new system would therefore be to create a compelling user interface and to add the ability to collaborate in text and images. This collaboration would augment that already created by shared camera control and be set up as a game in which players could earn points by taking snapshots of animals and classifying them. Allowing for autonomous tracking of interesting objects was originally a goal of the first CONE prototype, but was never implemented. We wanted to revisit this problem by creating an autonomous client that could scan and capture animal activity. Finally, because the National Science Foundation sponsors the CONE project we wanted to make the code-base for any new CONE system available as open source. This would allow other

research groups and the public to contribute to the project and the ability to create their own deployments.

5 CONE-SF DESIGN AND IMPLEMENTATION

In implementing CONE-SF we decided to split the design and development into three distinct parts: Server, Client, and Website. The server would primarily be responsible for controlling the camera, but would also contain the autonomous client and a video relay server. The client would present the graphical user interface allowing users to make requests to move the camera, and the website would provide the environment in which users could collaborate using both client, text, and images. In addition to being logical boundaries in the CONE system, this division would allow team members to work independently, except when needing to discuss the contract specification of an interface between the parts.

In working on the CONE-SF project I was responsible for the design and development of the client, as well as the video relay server and autonomous client. In the next sections a brief discussion of the design and implementation choices for the main parts of the CONE-SF system will be given together with an overview of system architecture. The rest of the paper will then focus on the details of the parts that I was responsible for.

5.1 Server

To accommodate the requirements set forth for a new CONE system and to meet our goals, we decided to implement the CONE server as a Java web-service that utilizes a standard webAPI protocol. Such a service would present various advantages over previous CONE prototypes. First the server would be contained in a single component and would be easy to install by simply dropping it into a web-services container, such as Apache Tomcat. It would be centrally configurable making it easy to set up and could be expanded by adding additional Java classes (modules) to the service. Finally the server would be client-agnostic in that any client adhering to the interface exported by the web service could connect and access the CONE server. This means that new custom clients can be easily written and can be web-based, as in the past, desktop-based, or even written as applications that run mobile phones. To make CONE-SF scale better to many users and overcome bandwidth limitations to the camera, we decided that a video relay server module would be written as part of the CONE server. The goal of creating an autonomous client to scan and capture interesting animal activity would be accomplished by building it as a server module as well. Finally a panorama-generation and calibration tool would be built as part of the server to be able to automatically generate a panorama and calibrate the system to the camera.

5.2 Client

Previous CONE clients had been implemented as a Java Applet able to run in any Java supported browser. It was decided to continue with this model, which would make it easier to re-implement. To meet the requirements for the new CONE system the client

would need to be rewritten to be more efficient and the interface design would need to be redone. In creating the interface design we looked at how data was being visualized on the web. One concept we came up with was based off of a Google Maps style view, with the navigation tools superimposed on a large viewing screen. After sketching some prototype specs (see Appendix A) we weighed the pros and cons of our candidate designs and settled on reworking our old design, including a new color scheme and features from our list of requirements.

5.3 Website

The website we had used for previous CONE prototypes served merely as a means to access the Java Applet client. As discussed in the goals for CONE-SF we wanted to make the website a space for online collaborative observation. This required a web application with a completely new interface design, a database to store images, and the ability for viewers to add comments. In addition, we wanted to incorporate a game into the system, in which players can earn points by taking capturing images of animals and classifying them. Ruby-on-Rails was decided upon as the best web application framework to use to meet these requirements, since it provides skeleton code frameworks (scaffolding) for rapid development of database driven websites. In designing the interface for the website we went through the same process as the client, generating and sketching ideas until we settled on an initial design (See Appendix B). Once an implementation of the final design had been made, it was then further reworked through focus group sessions, in which users were asked to use and comment on the functionality of the site.

5.4 Architecture

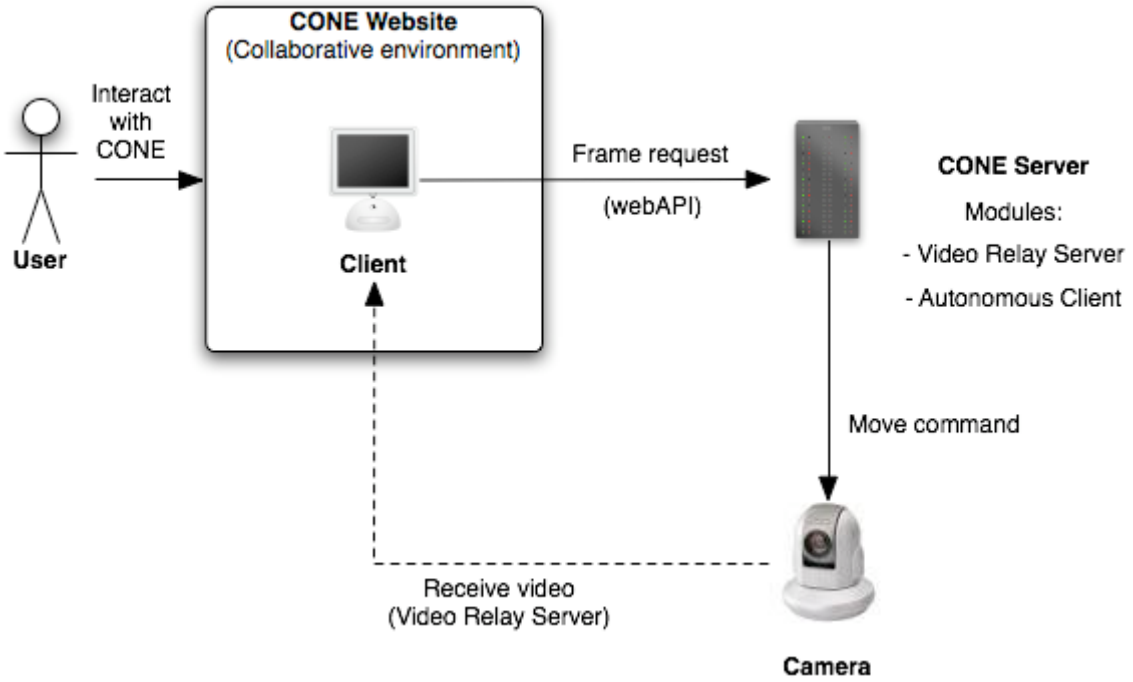


Figure 3. Architectural overview of the CONE-SF system

To interact with the CONE-SF system viewers start by accessing the CONE-SF website. The main page of the website is the camera page in which the viewer is presented with the CONE-SF client and can participate in viewing the live video feed and controlling the camera. In addition, the client allows users to capture images that are then stored in a central database and displayed on the website. A gallery page was created in which viewers can see, comment on, and classify images taken by others. Viewers also have a personal gallery page in which their point score is displayed together with the snapshots they have captured using the client. The point system assigns points to players when they take snapshots and when they correctly classify images as defined by a majority. An example of a viewer's personal gallery and the classification interface is show below in figure 4 and 5.

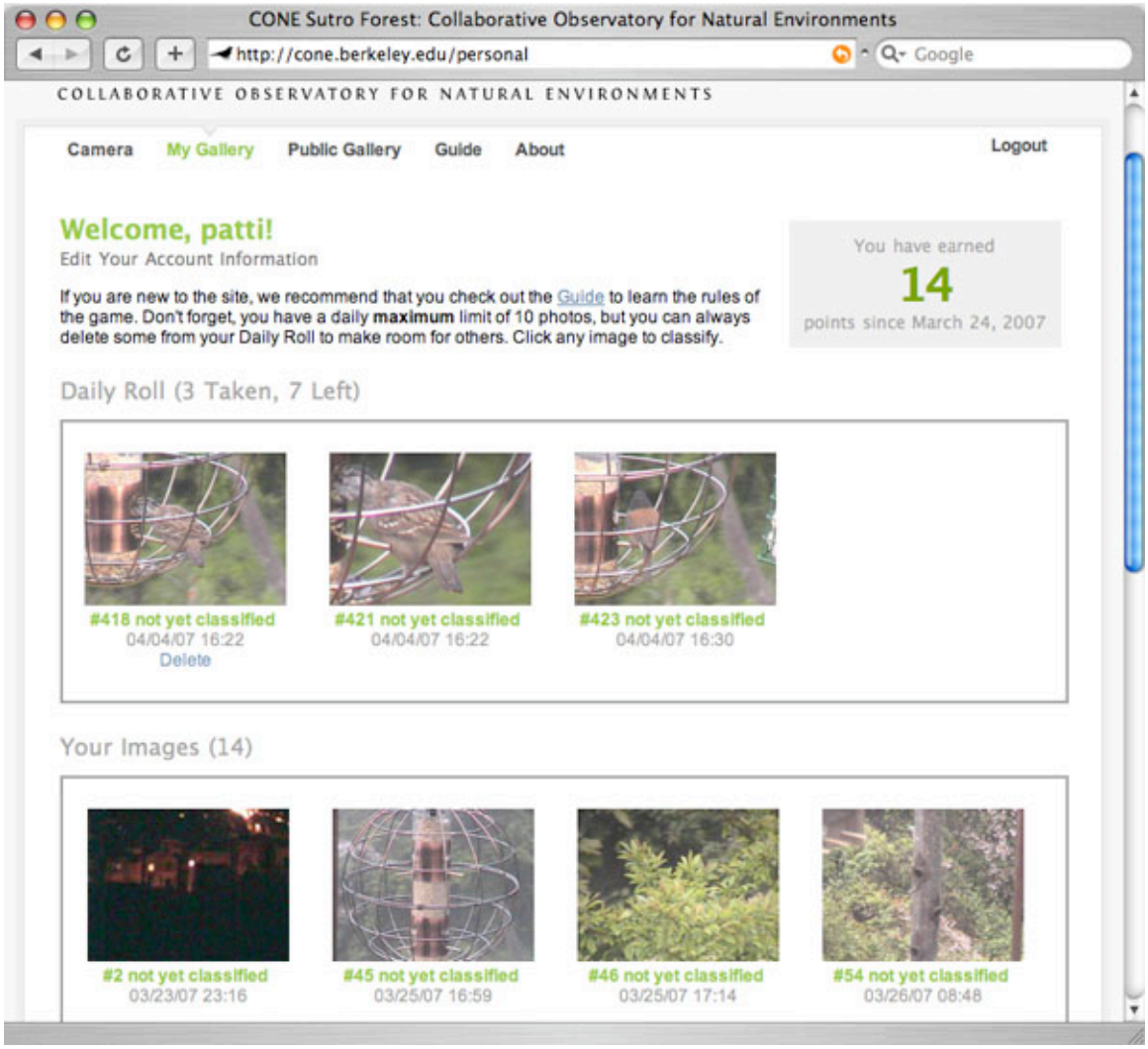


Figure 4. Example of personal gallery page

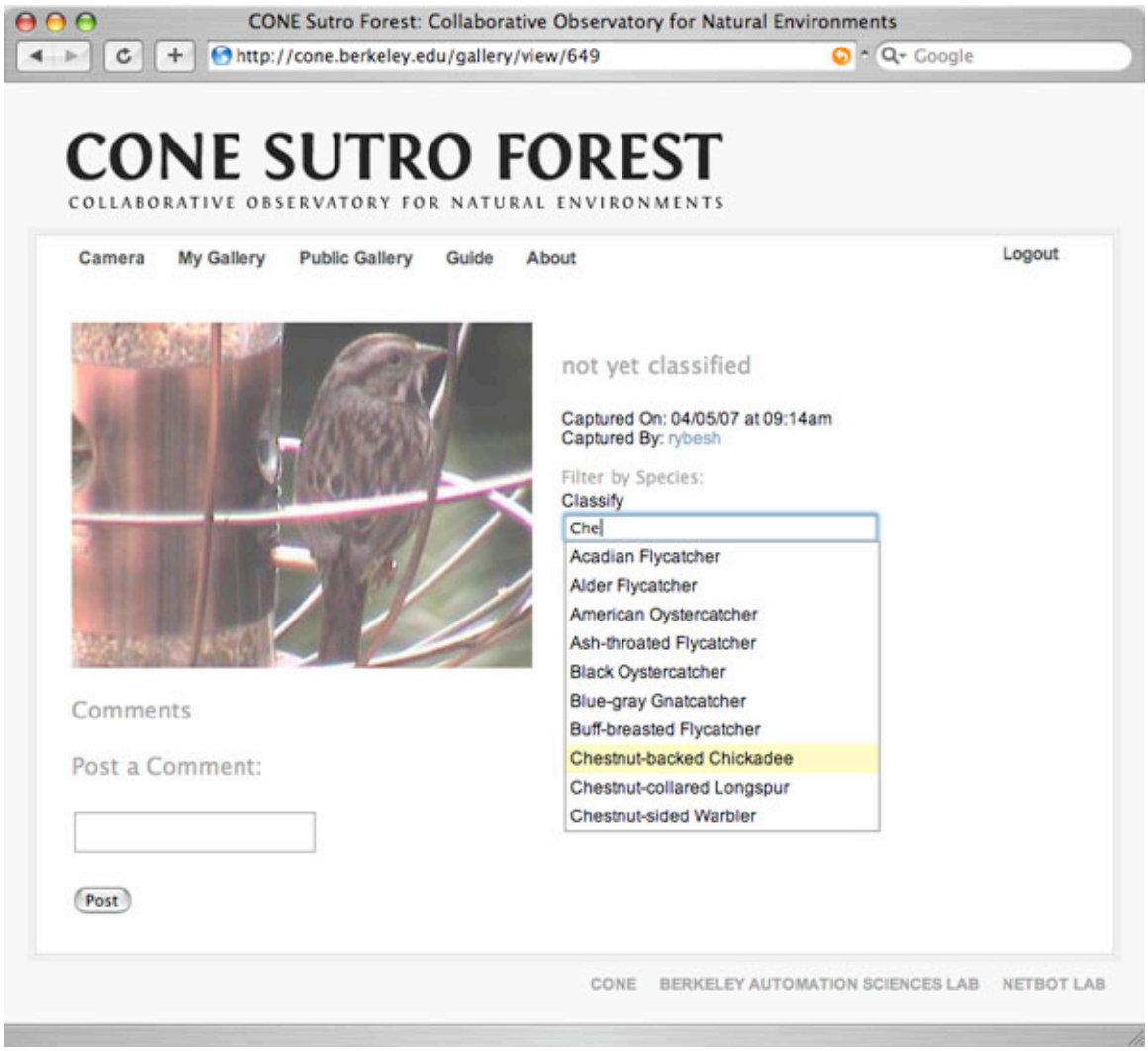


Figure 5. Example of classification interface

When viewers make frame requests using the client they are sent to the CONE-SF server using the web-services webAPI. The server gathers requests into a list of current requests. Every 5 seconds an algorithm is run on the server to calculate the optimal frame for the camera to move to. This frame is translated into a pan, tilt, and zoom command, which is sent to the camera that then moves to the specified location. The algorithm used in previous CONE prototypes calculated the optimal frame by attempting maximize total viewer satisfaction, but did not take into account the amount of time a viewer had been waiting to be satisfied [3]. This could lead to starvation because viewers' requests would only be removed from the current requests queue when having been at least partially satisfied. To overcome this problem a new frame selection algorithm was developed that adds the notion of weight to a frame request. The weight of a request is increased in proportion to the time the request has not been satisfied. The goal with this new algorithm was enhance the collaborative experience when controlling the camera.

To receive video from the camera the client is connected to the video relay server, which was created as a module for the CONE-SF server. Having the relay server as a part of

CONE-SF server makes the system simpler and more robust, since clients only have to connect to one service. Another module that is part of the server is the autonomous client. It is activated either when no one is logged into the system or when no requests to move the camera have been made for a certain period of time. Once activated the autonomous client will scan the viewable area in attempt to capture interesting animal activity.

6 VIDEO RELAY SERVER

A key concept behind the CONE project is to place robotic cameras in locations that allow for the observation of animals in their natural environment. Such locations generally do not offer an Internet connection with adequate bandwidth to stream video to more than one or two clients. In the case of CONE-SF where the camera is installed at a private home in San Francisco, the camera is connected to a Comcast cable Internet connection with a maximum upstream bandwidth of 768kbps. To overcome bandwidth limitations and be able to support more users we designed and implemented a video relay server in Java as a module for the CONE server. This server maintains a single connection to the camera through which is receives video and is placed on a high-speed Internet connection, like the one available at UC Berkeley. Clients then connect to the relay server and stream video as if they were connected to the camera directly. This allows us to support a number of users proportional to the Internet connection of the relay server, rather than be limited by the camera connection (see figure 6 below).

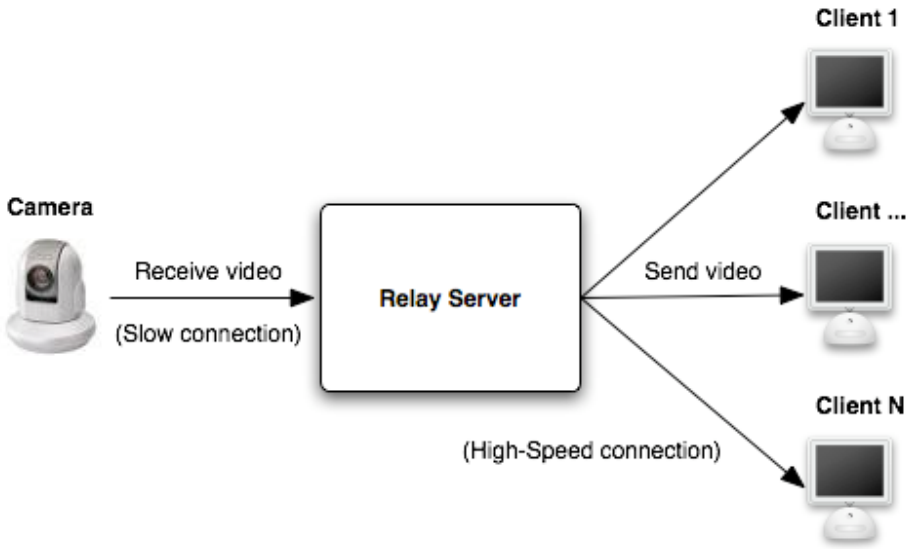


Figure 6. Video received through video relay server

6.1 Design

An initial relay server was created as a proof of concept and to learn from. This application was a single threaded and used synchronous blocking I/O. It would run a continuous loop that would receive a chunk of data from the camera and then send it to each client connected in turn. This worked well for relaying video to a couple of clients but performance degenerated as the number of clients increased further. The reason was

that blocking I/O required each client be sent a complete data chunk before the next client could receive anything. The more clients that connected the longer all clients had to wait to receive data, resulting in reduced frame rate for everyone.

To overcome the problem of blocking I/O we went to a multithreaded (thread-per-connection) model instead. In this model each client connected to the relay server is associated with its own thread of execution that handles sending data to that client. This removes the interdependence between clients and allows each client to receive data as fast as the camera connection will allow. A distributor thread is responsible for receiving data from the camera, which is collected into entire images and stored in a circular buffer of image chunks. We store entire images in a buffer to deal with clients who cannot keep up with the rate at which the relay server receives data from the camera. Instead of falling farther and farther behind, clients are sent the newest image from the image buffer. This ensures clients see live video, even if at a lower frame rate than what the relay server and camera are capable of providing.

6.2 Implementation

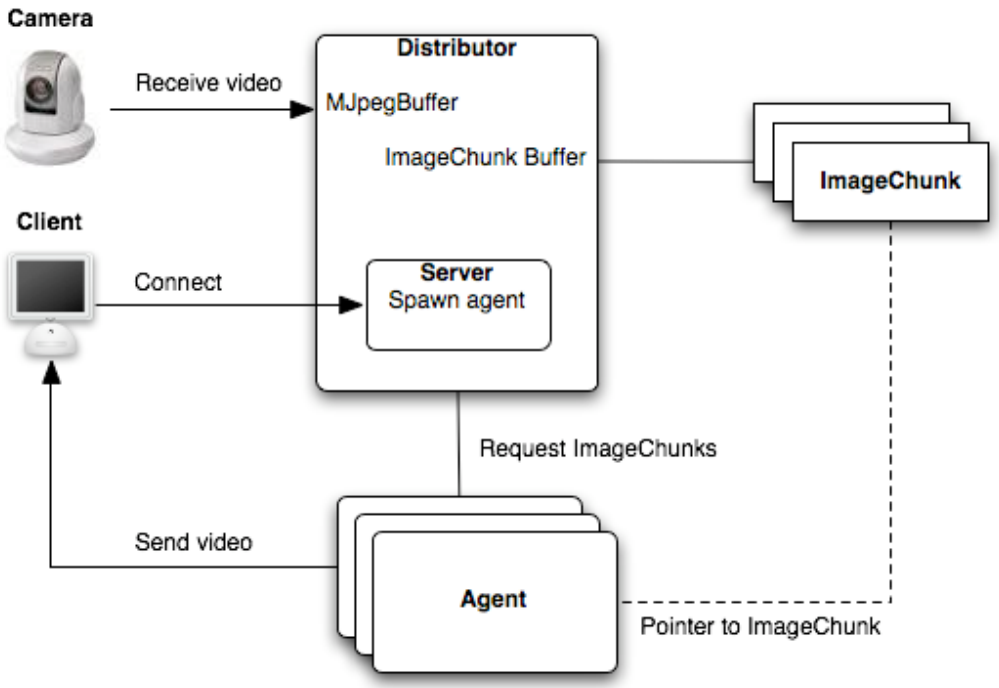


Figure 7. Video relay server architecture

An overview of the video relay server architecture can be seen in figure 7. The *Distributor* class is the main thread of the relay server application. It is responsible for maintaining a connection to the camera from which it receives video data in the form of separately compressed JPEG images known as motion JPEG (M-JPEG). Bytes of image data are continually retrieved and gathered in an *MJpegBuffer* that is used to decode these bytes into JPEG images. Each decoded image is stored in a circular image buffer as an *ImageChunk*. The *Distributor* also contains a *Server* thread that listens for client connections on a predefined port. When a client connects, the *Server* thread obtains a

socket for the client and passes it to a newly created *Agent* thread that will handle the connection with the client from this point on.

An *Agent* thread loops indefinitely sending the newest *ImageChunk* from the *Distributors* buffer of images to the client. It does this by calling the getImage() method in the *Distributor*, which will either return a pointer to the latest *ImageChunk* or block the *Agent* if it has already sent this image to the client. In the later case the *Distributor* will notify the *Agent* when a new image is available and sending will resume. Pointers to *ImageChunks* are used to increase efficiency by eliminating the need to copy image data. This does however raise the possibility of an *ImageChunk* being overwritten while an *Agent* is reading it. In our experience this can be avoided by making the buffer of images large enough (100 in our implementation).

After running the relay server for a short period of time with the CONE-SF camera we noticed that the connection to the camera would often be lost or blocked indefinitely. This was a major problem since it rendered the entire system useless. One of the issues seemed to be with the cable Internet connection, which would cause our *Distributor* to block indefinitely when trying to read video data from the camera.

To solve the problem we added a read timeout that attempts to re-establish connection with the camera should the timeout be reached or should the connection be lost for any other reason. Re-connection attempts are implemented with exponential back-off to prevent a flood of reconnection attempts. In addition, we added a *Beacon* thread to the relay server that monitors the *Distributor* and sends an email to system administrators should anything go wrong.

6.3 Results and Discussion

To determine the performance of the video relay server and how well it would scale we put it under varying degrees of load in a lab environment. Frames/second (FPS) were used as a measure in these experiments, since FPS is a standard measure for the quality of a video stream. First the relay server was connected to the camera. On this connection we measured a steady video stream from the camera being received at about 6 FPS. Then multiple machines connected to the relay server via 100mbit switched Ethernet were set up to each spawn clients that would access the relay server video stream. The total number of clients spawned by all machines was varied from 1 to 100 clients over a set of runs lasting one minute each. The 100mbit connection between relay server and clients would theoretically be able to support about 200 simultaneous clients at 10 FPS and therefore ensured that connection speed would not be a bottleneck in our experiments. After each one minute run the average FPS for each client was averaged with the other clients in the run to get the mean FPS for all clients. Figure 8 below shows the results for the experiments with the mean FPS graphed as a function of the number of clients connected to the relay server.

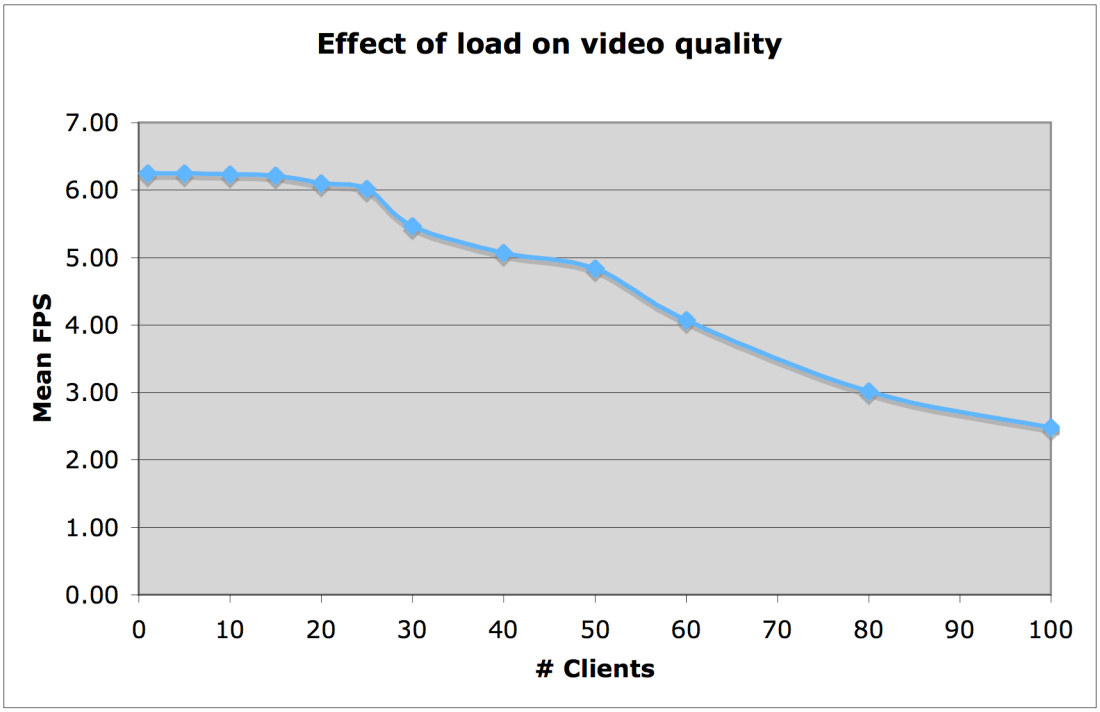


Figure 8. The effect of load on video quality

The results show that clients experience practically no reduction in video quality up to about 25 clients. From 25 to 50 clients the FPS received by clients is still only slightly less than that of a single client connected directly to the camera. At 50 clients the quality starts to taper off and after 80 clients the FPS received is at a level where there would be a noticeable reduction in quality resulting in choppy video being displayed. Being able to support up to 80 clients reasonably well is a huge gain over the limitations of slow Internet connections, such as the one used by CONE-SF camera. To support even more clients, additional video relay servers could be set up on different High-Speed connections and connected to the camera in series.

In setting up CONE-SF we used a single relay server, but limited the number of simultaneous connections to 20. This conservative choice was made to ensure maximum video quality for all viewers in testing the system under real world conditions. The system has now been running for about a month and there has been no marked decline in video quality, even when the system has been at the 20 client limit.

7 CONE CLIENT

In starting work on CONE-SF it was decided to completely revamp the architecture of the system and therefore also the client. A major reason for this revamp, as discussed, was due to system instability causing frequent crashes. The system also did not scale well and therefore could only support a very limited number of users. Finally the graphical user interface (GUI) needed to be redesigned to create a more collaborative feel and meet the requirements and goals set forth for a new CONE system.

To solve these problems the CONE client was rebuilt from scratch to be as lean and efficient as possible. As an example, custom code was written for many parts of the client instead of having to rely on third-party libraries. This reduced the footprint of the client to just less than 50 kilobytes, small enough for most Internet connections to download in seconds. The size of the video stream processed by the client was reduced from 640x480 to 320x240 to limit bandwidth requirements. A new design for the client was created and features of the graphical user interface were changed, as discussed in the section on the design and implementation of CONE-SF. The resulting client interface and a description how it works is detailed next.

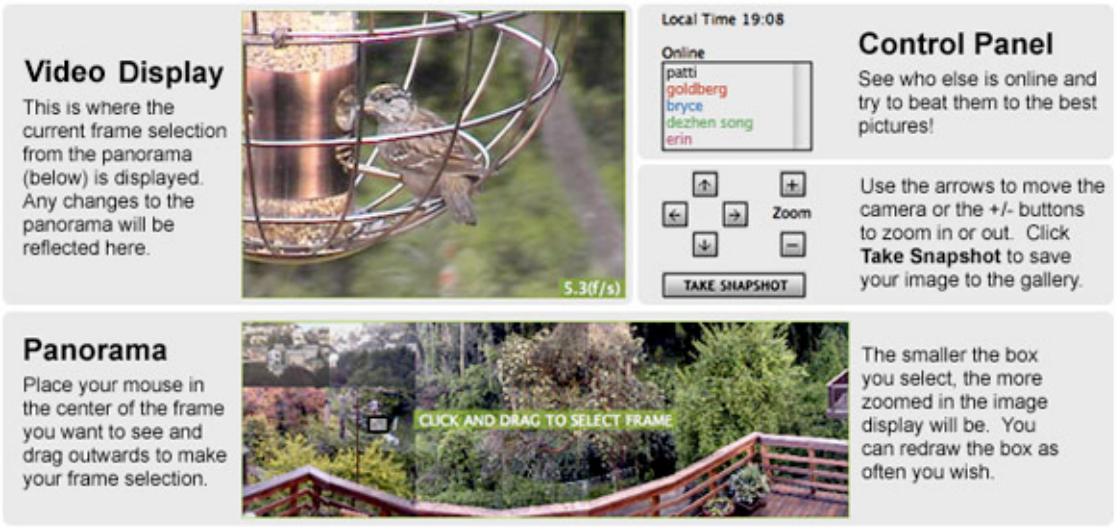


Figure 9. Screenshot and description of CONE-SF client

7.1 Client GUI

The CONE client as seen above in figure 9 has three main components that viewers can use to interact with the CONE system: Video Display, Panorama, and Control Panel. The video display is where the user can see live video from the camera. The video shown is a reflection of the current frame selections made by viewers. To allow users to gauge the quality of video they are viewing the current number of frames per second received by the client per second (fps) is overlaid onto the live video. The panorama component displays an image that represents the field of view of the robotic camera. On this panorama users can draw frames that are then turned into requests for the camera to move to and display the frame selected. To draw a frame viewers simply click with their mouse on the panorama and drag to adjust the size of the frame. The location initially clicked on the panorama becomes the center of the frame. This is in contrast to most user interfaces where frames are drawn from the corner and not the center. We made this choice because we thought that people would naturally click the location where they want the camera to move to. The center of the frame drawn therefore determines where to move the camera and the size establishes the zoom level. Both the current frame displayed by the camera and all the selections made by users are drawn on top of the panorama. In the control panel users can see who else is logged into the CONE system and the local time. Each user is displayed in their own color, which is also the color used to draw their current

frame selection on the panorama. In addition to the list of users and time, the control panel contains buttons that allow for fine-grained control of the current frame being displayed by the camera. Clicking and optionally holding down any of the arrow control buttons move the frame in the direction corresponding to the arrow. The plus and minus buttons adjust the zoom level of the camera and finally a "take snapshot" button makes it possible for viewers to capture a photograph of the current scene.

7.2 Implementation

The CONE client was implemented as a Java Applet using version 1.4 of the Java Development Kit (JDK), which includes the Swing GUI toolkit, which we used to create the graphical user interface. The choice of Java was made due to the power of the language derived in part from the abundance of built in libraries, platform independence, and the ability of Java Applets to run in a browser. We wanted the CONE system to be available to as many people as possible and using a Java Applet practically anyone with a browser and the Java Runtime Environment (JRE) installed can use CONE.

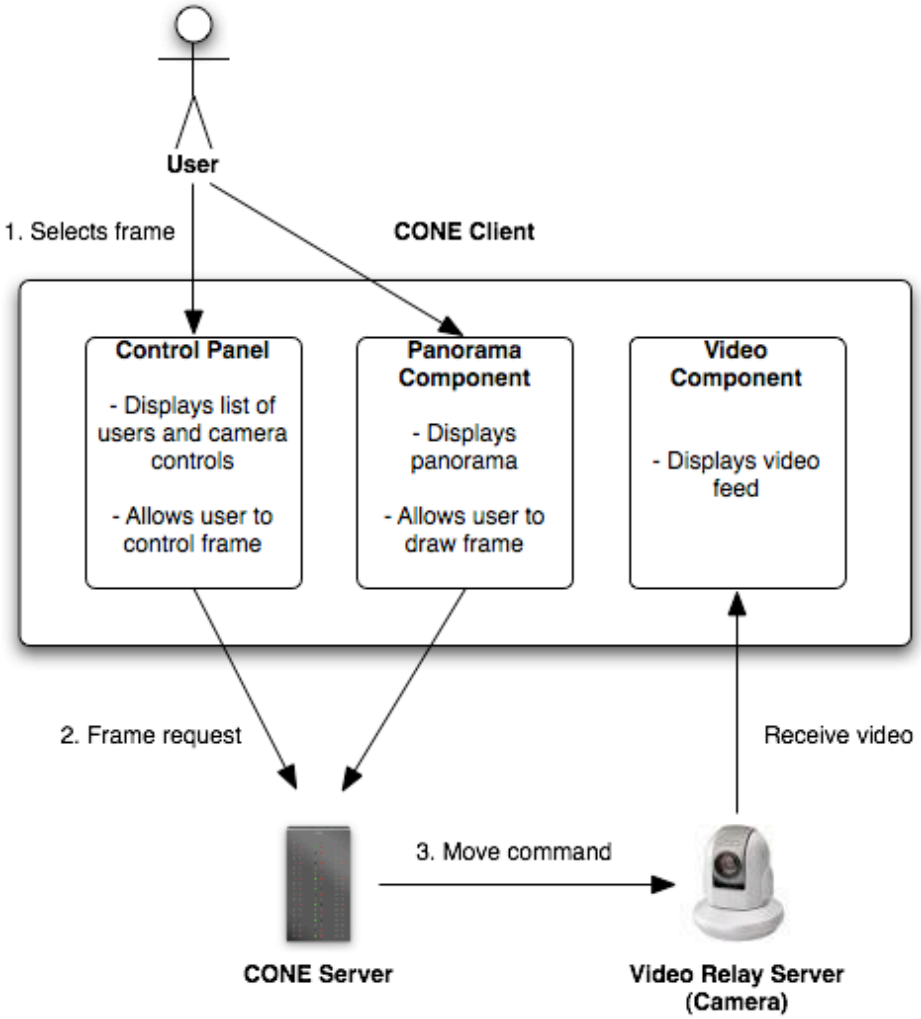


Figure 10. Architecture of the CONE-SF client

The architecture of the client implementation, which can been seen in figure 10 above, is very similar to its graphical layout. A class called *Client* is the starting point of the client applet. On initialization it loads parameters supplied to the applet, such as the username of the viewer and the address of the CONE server. These parameters are then used to connect to the CONE server and register the viewer. Finally, it loads and displays the three graphical components of the client each contained in their own classes: *VideoComponent*, *PanoramaComponent*, *ControlPanel*. When the applet is stopped, the *Client* class is also responsible for un-registering the viewer from the CONE server and shutting down the applet.

The *VideoComponent* class is responsible for retrieving video from the camera and displaying it in the video display frame of the client. Video from the camera is available as a sequence of separately compressed JPEG images known as motion JPEG (M-JPEG). To display this video the *VideoComponent* creates a connection to the camera (or video relay server) and continually retrieves bytes of image data. These bytes are gathered in an *MJpegBuffer* that is used to decode these bytes into JPEG images that are then displayed. The number of frames decoded and displayed per second is continually calculated and overlaid onto the video.

The *PanoramaComponent* class displays a panorama representing the field of view of the camera, which is retrieved from the CONE server when the client applet is initialized. A user can click-and-drag on the panorama to draw a frame that will be sent as a frame request to the CONE server. To make this possible the *PanoramaComponent* implements Java's built in mouse event listener interfaces *MouseListener* and *MouseMotionListener* and registers itself as a listener object. As a result, when a mouse event occurs such as a user pressing or releasing a mouse button, a method corresponding to the event is called in the *PanoramaComponent*. The most important of these methods being: *mousePressed()*, *mouseDragged()*, and mouseReleased(). *MousePressed*() is called when a user first presses down on a mouse button while the mouse cursor is over the panorama. It sets the location of the mouse cursor as the center point of the selected frame. Subsequently when the mouse is dragged (still with the mouse button pressed), the *mouseDragged*() method is continually called, recalculating and redrawing the selected frame on the panorama based on the position of the mouse cursor. At last when the user releases the mouse button, *mouseReleased()* sends the parameters of the selected frame as a frame request to the CONE server. The frame drawn by a user and in turn a frame request, is determined by the following three parameters:

($frameX$, $frameY$) - Frame center
$frameZoom$ - Frame zoom level

The width and height of the frame can easily be derived from the zoom level (see the discussion on frame selection below). In addition to the above, the *PanoramaComponent* spawns a separate thread to draw the frame requests made by other users and a frame representing the current camera view.

The *ControlPanel* class displays who else is logged into the system by having a separate thread poll the CONE server for this information on a fixed 5-second interval. To allow for fine-grained control of the current camera frame the *ControlPanel* presents a set arrow buttons to move the frame in a specific direction and plus and minus buttons to adjust the zoom level. The longer one of these buttons is pressed without releasing, the more the current frame is altered. This is implemented by creating a separate timing thread each time a button is pressed, which continually issues the desired action until the button is released. This behavior was used since it is analogous to that of directional buttons on a PC keyboard. For the directional controls the current frame is moved by multiplying the current zoom level by a fixed step gain and adding or subtracting it from the X or Y component of the frame, depending on the direction desired. Moving the current frame left for example, would be calculated as follows:

$$frameX = frameY - (frameZoom * STEP\_GAIN)$$

This makes the frame move in proportion to the current zoom level. In contrast, the zoom level controls simply increment or decrement the current zoom level to calculate the new zoom level. In addition to the camera control buttons, the *ControlPanel* class also displays a button called "take snapshot" that lets users take a photograph of the current scene. This is implemented by making a call to CONE server, which handles capturing the image and storing it for later viewing. The reason it was choosing to handle this on the server side rather than on the client, was to save having to send the image to the server. The tradeoff is that there might be minor delay between when the "take snapshot" button is clicked and when the snapshot is actually taken. Below is an overview of the CONE client architecture.

7.2.1 Client/Server Communication

For communication between the CONE server and client, the web-services webAPI protocol SOAP is used. SOAP uses XML-based messages to communicate between heterogeneous systems over a network, most commonly using HTTP. The CONE server adheres to and is deployed using the open source web services framework Apache Axis, which consists of an implementation of the SOAP server. As a result the entire CONE server API is exposed as a web service described by a dynamically generated document adhering to the Web Services Description Language (WSDL). This WSDL document is in XML format and describes the input and outputs, types and request protocols, required by each API function. To make is easy to create clients using a SOAP enabled web service there are libraries available that can automatically generate SOAP client stubs based on a WSDL document.

The reason for choosing SOAP as the protocol for client/server communication is that any SOAP compliant client can access and use the CONE system. Due to the XML-based nature of SOAP such clients can be written on almost any combination of platforms and languages. In addition, the ability to automatically generate client stubs simplifies the matter even more.

The CONE client we wrote implements the CONE server webAPI using a custom coded SOAP client called *SoapProxy*. The reason for not using a library to automatically generate a SOAP client was the space requirement of such a library, which would have brought the footprint of the CONE client from less than 50 kilobytes to over a megabyte.

7.2.2 Frame Selection

The frame drawn by a user on the CONE client panorama must be mapped to a frame request and ultimately to pan, tilt and zoom actions taken by the camera. To have the frame selection made by a user be displayable by the camera, it must maintain the same aspect ratio as the camera. In the CONE-SF implementation this aspect ratio is 4:3. Therefore, anytime a user presses and drags the mouse on the panorama, the *PanoramaComponent* maintains the correct aspect ratio by calculating the frame zoom level in the following way:

$$zoomX = |frameX - x| / ASPECT\_WIDTH$$
$$zoomY = |frameY - y| / ASPECT\_HEIGHT$$
$$frameZoom = MAX(zoomX, zoomY)$$

Where (x, y) is the current location of the mouse cursor on the panorama and in the case of CONE-SF, the $ASPECT\_WIDTH$ and $ASPECT\_HEIGHT$ are 4 and 3 respectively. From this zoom level, the frame width and height used to draw the frame on the panorama is calculated as follows:

$$frameWidth = frameZoom * ASPECT\_WIDTH$$
$$frameHeight = frameZoom * ASPECT\_HEIGHT$$

This ensures that the frame width and height are always proportional to the aspect ratio of the camera and thus translate into a frame displayable by the camera.

8 AUTONOMOUS CONE CLIENT

It was originally a goal of the first CONE prototype (CONE 1.0) to create a client that would allow for autonomous tracking of interesting objects, such as animals. At the time however it presented more hurdles than expected and was never implemented. Since then, the CONE project prototype ACONE (Automated Collaborative Observatory for Natural Environments) was installed that attempts to capture images of the thought to be extinct Ivory Billed Woodpecker [11]. To do this, the ACONE system records video sequences, which are then later analyzed to determine if the Ivory Woodpecker was present. In a similar fashion in CONE-SF we wanted to be able to detect birds or other animals and take pictures of them. The idea was to create an autonomous client that would patrol looking for activity when nobody else was using the camera as to not interfere with regular use. This autonomous client would potentially provide snapshots of birds and other animals that people would otherwise not capture. This could be due to viewers not

being able to collaborate effectively to get the animal in sight, not being quick enough to take the picture, or simply not seeing it because it was never within view of the camera.

8.1 Design

To create an autonomous client for CONE-SF as described we needed a way to scan around the camera's viewable area and detect movement that would then trigger the camera to zoom in on the activity and take a snapshot. This naturally broke the problem into two parts, an algorithm for scanning and a motion detection algorithm. These would then be combined with the frame request and snapshot features already implemented to create an autonomous client. We came up with a scanning algorithm based on an activity map and experimented with two different types of motion detection algorithms discussed in the literature. To be useful we also decided that the autonomous client should be restricted to only run during daylight hours.

8.2 Scanning algorithm

The idea behind the scanning algorithm we came up with was to create a probability distribution (activity map) of where motion had been detected in the past. This probability distribution could then be sampled to determine where the autonomous client should request the camera be moved. In addition, some randomness would be induced into the sampling to insure the autonomous client would discover new areas of activity. To accomplish this we used the panorama generated by the CONE server, which represents the area viewable by the camera. The panorama was broken in to square sections and each section associated with a value in the probability distribution (See figure 11 below for a panorama broken in to sections). The autonomous client will then periodically sample the distribution to select a section, which it will then request the camera move to by making a frame request matching the square section chosen. Initially all sections will have uniform probability of being chosen. However, as the autonomous client samples the distribution and moves to a section and motion is detected, a counter is incremented for the given section. Periodically this count of detection events for each section is then used to update the probability distribution. An extension to the algorithm that might be interesting to pursue, but which we did not have time to, would replace the single activity map with an array of activity maps corresponding to different periods of time during the day. This would possibly more accurately account for where to find activity throughout the day.

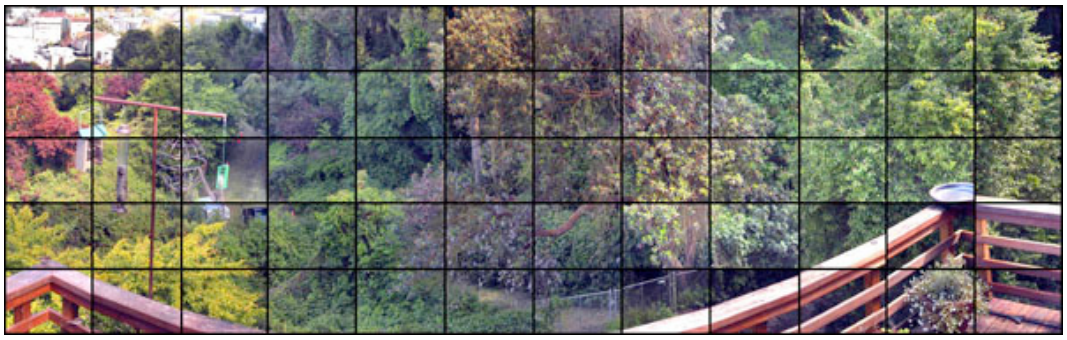


Figure 11. Panorama as an activity map

8.3 Motion detection

Once the autonomous client has used the activity map to choose a section of the panorama and the camera has moved to it, the client needs to look for activity. It does this by invoking a motion detection algorithm that will attempt to segment the current view of the camera into foreground (motion) and background. If motion is detected and it is above a certain threshold the autonomous client will then zoom in on the motion and take a snapshot, hopefully capturing the activity detected. The Sutro Forest in San Francisco, which is the scene the CONE-SF camera overlooks, is a challenging environment to detect legitimate activity in due to all the motion in the scene. As can be seen in figure 11 above, it contains a lot of trees and bushes that move in the wind. We therefore decided to take two approaches to use try two types of motion segmentation algorithms from the literature to see which would work best for our type of environment.

8.3.1 Temporal differencing

The first approach we used to motion segmentation used a three-frame temporal differencing algorithm. Temporal differencing uses the pixel-wise difference between two or three consecutive frames in a video sequence to extract regions of motion. It is very adaptive to dynamic environments and simple to implement, making it a good starting point for our purposes. In three-frame differencing a pixel is foreground if its intensity has changed significantly between both the current image and the last frame and the current image and the next-to-last image. Let $I_n(x)$ represent the intensity value at pixel position x, at time t = n. Then a pixel is a foreground if

$$(|I_n(x) - I_{n-1}(x)| > T_n(x)) \text{ and } (|I_n(x) - I_{n-2}(x)| > T_n(x))$$

where $I_n(x)$ is a threshold describing a significant intensity change at pixel position x. Using this formulation a motion image can be created consisting of pixels labeled as foreground according to the above criteria and the rest as background.

8.3.2 Adaptive background subtraction

Background subtraction is a popular method for motion segmentation, but generally only works well with a relatively static background. It detects moving regions in an image by thresholding the difference (or error) between the current image and a reference background image in a pixel-by-pixel fashion. A common approach to creating a background model is to determine the average value of each pixel over a number of frames in a video sequence. This is simple, but very sensitive to changes in dynamic scenes. This is because pixels in a scene that has a lot of background motion, such as that in CONE-SF, might need to represent multiple surfaces. An example would be a pixel that at times is a rock and at others a leaf that is moved over the rock by the wind. Creating the background model as described would likely result in this pixel being classified as foreground when in fact it's part of the background.

To overcome the problem of creating a background model that can adequately represent a dynamic environment adaptive background models were suggested. One such model uses a mixture of Gaussians for each pixel that is able to represent multiple surfaces and can be used to calculate the likelihood that a sampled pixel is one of them. A Gaussian mixture model for background subtraction has been implemented in numerous ways in the literature, but here we will describe our specific implementation for CONE-SF.

In contrast to the temporal differencing algorithm, background subtraction requires that a set of samples be initially collected to create the background model. Once these samples are collected in our implementation we create a mixture of Gaussians for each pixel. Each Gaussian was represented by variance (σ^2) and a mean (μ). The mean in effect represents the color of a surface and the variance, how much samples of that surface differ. To determine the mean for each Gaussian the K-means clustering algorithm was used. For each of the groups determined by the K-means algorithm the variance was also calculated and the group size used to determine its prior (ρ). This constituted the model-building phase where after subsequent frames could be evaluated against the model to determine a motion image as follows

$$M(x) = \sum_{K}^{1} \rho_K * f(x;\mu_K,\sigma_K) > T\ ,\ f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where T is a threshold describing the confidence required for a pixel to be labeled as background. K is an adjustable parameter, which determines the number of Gaussians used for each pixel in the model and therefore surfaces that it can represent. There is an inherent tradeoff between performance and being able to adequately model the background when determining the value for K. We found setting K=3 would allow us to run in real-time, while still creating satisfactory background model.

In our implementation of a Gaussian mixture model we made some simplifications to the algorithm, both to make implementation easier and due to the nature of the problem we were trying to solve. The first was to convert the RGB images supplied by the camera into black and white images. Having each pixel be represented by one value instead of three would allow us to use single dimensional Gaussians. This would greatly speed up the creation and evaluation of the model. The second was to not update the background model as new frames were acquired and evaluated. Formulations of the Gaussian mixture model allow for this, but because we were going to move the camera frequently and therefore having to build a new background model, this seemed unnecessary and would severely hurt performance.

8.3.3 Algorithm comparison

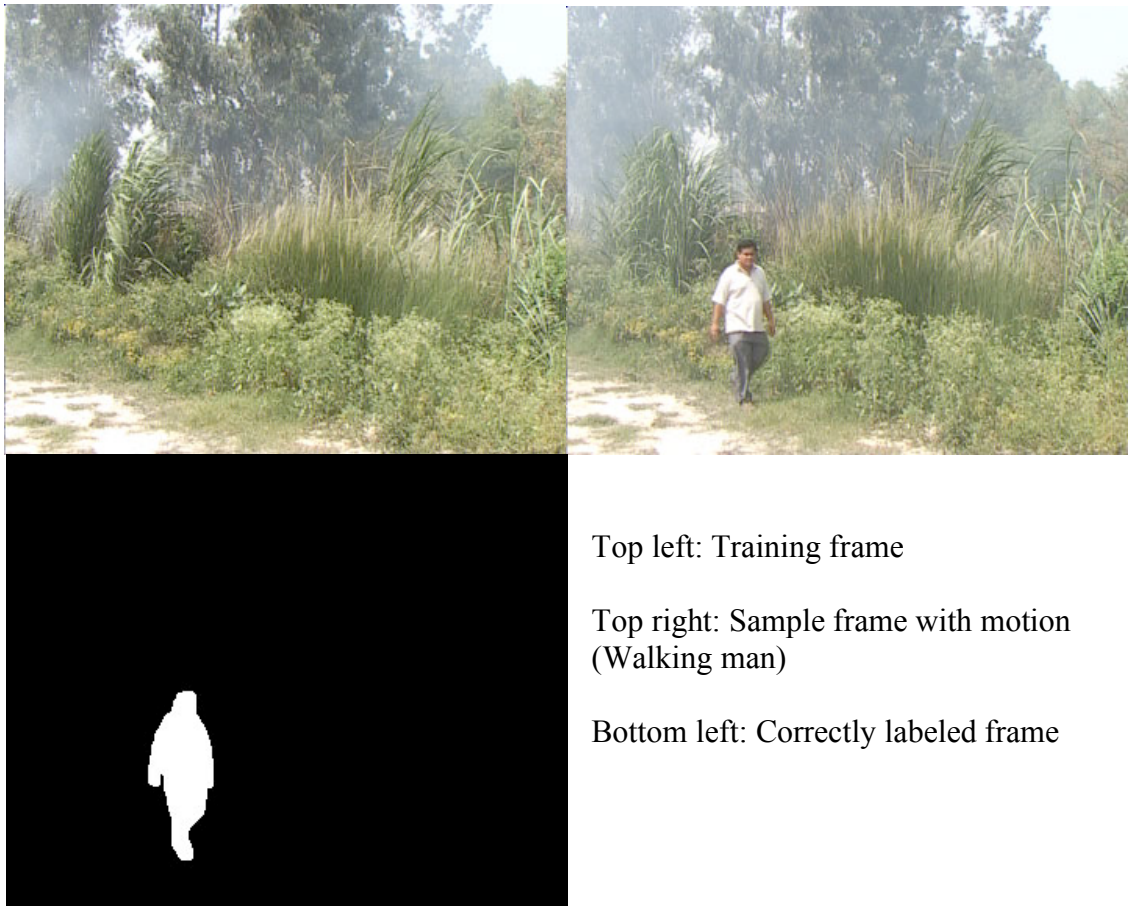
To compare the performance of the two motion detection algorithms we used the common measures of *precision* and *recall*. Precision represents the percentage of positive predictions that are correct. In our case: the percentage of pixels classified as foreground by the algorithm that are actually foreground. Recall on the other hand, is the percentage

of foreground pixels in the image that are classified by the algorithm as foreground. The formal definition for precision and recall are:

$$precision = \frac{True\_Positive}{(Tue\_Positive + False\_Positive)}, \quad recall = \frac{True\_Positive}{(True\_Positive + False\_Negative)}$$

There is an inherent tradeoff between these two measures. The more pixels classified as foreground and thus greater recall, the less the likelihood that these are all correct classifications, and hence a lower precision. Should precision be high, it is likely that not all foreground pixels were detected and therefore a lower recall results. To compare two segmentation algorithms it is therefore helpful to graph precision versus recall curve for each algorithm.

To create the precision versus recall curves for the two implemented algorithms we obtained a video sequence that was broken into three sets of frames. A set of training frames from which to create a background model, a number of sample frames with motion, and a set of corresponding frames labeled correctly with foreground and background. Examples of these frames are given below in figure 12.



Top left: Training frame

Top right: Sample frame with motion (Walking man)

Bottom left: Correctly labeled frame

Figure 12. Frames created from video sequence

Using the frames from the video sequence each algorithm was run and the precision and recall calculated for different values of the given algorithms threshold parameter. The resulting graph can be seen below in figure 13.

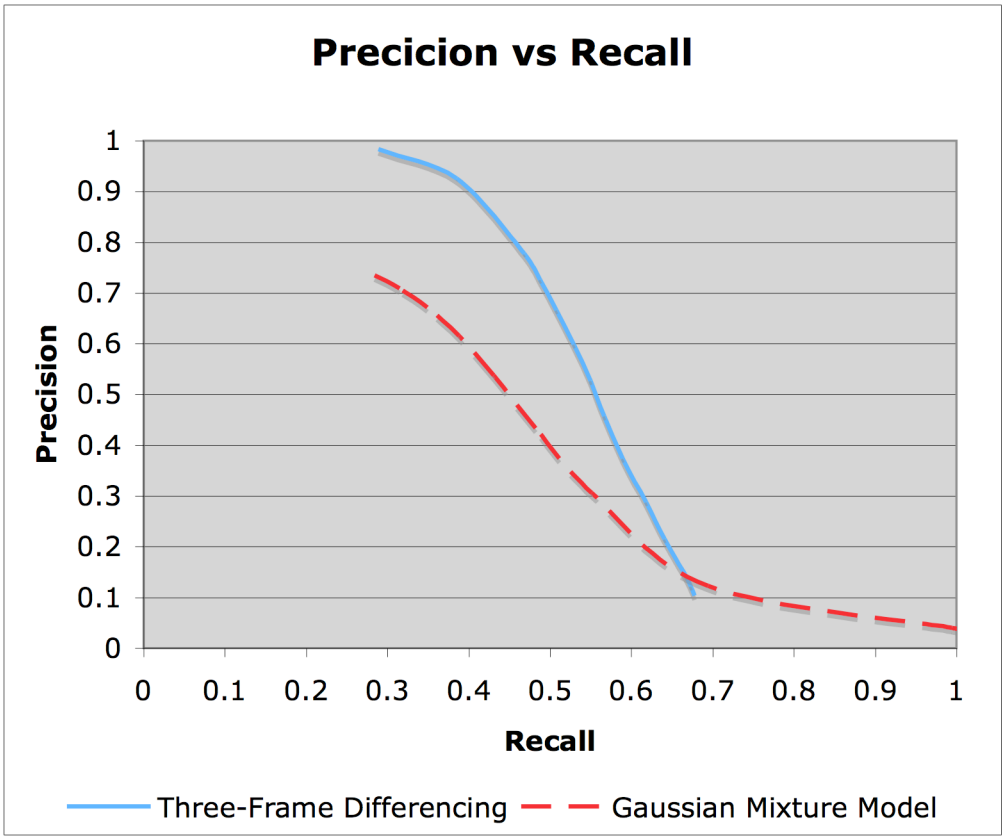


Figure 13. Comparison of motion detection algorithms

As we can see from the graph, the temporal differencing algorithm seems to both detect more motion and be correct in classifying it as such. This was a surprise to us, as we would have expected the more complex mixture model to outperform the simpler frame differencing approach. The reasons we have identified for the Gaussian mixture model being outperformed ironically relate to the simplifications we made in implementation, namely using black and white images and not updating the background model.

The video sequence contains a lot of slow moving smoke. When converted to black and white the intensity of pixels containing the white smoke change drastically. This combined with the smoke moving across the scene results in the outdated background model failing to classify the smoke correctly. But since the smoke is moving slowly, the temporal differencing approach correctly classifies the smoke as background. The problems with the Gaussian mixture model could potentially be solved either by continually updating the background model or by using RGB images and multi-dimensional Gaussians.

8.4 Status

The autonomous client has still not been tested as a complete system, since we believe we need to improve upon our motion detection capabilities before such a test will be useful. In particular we want to further explore the Gaussian mixture model for motion detection and mechanisms for object classification. The scene that CONE-SF overlooks is very dynamic and contains a lot of moving objects, such as bird feeders. The ability to classify objects seems necessary in such an environment to be able to discern animals from other moving objects. If good enough, such an object classification mechanism could even be used to automatically label and index images taken of different types or species of animals.

9 RELATED WORK

The CONE-SF client is a graphical user interface that allows many internet-based viewers to share simultaneous control of a networked robotic camera. The idea for such an interface originated with the ShareCam project at UC Berkeley [2][3]. In the ShareCam implementation of the interface viewers could specify a desired viewing frame drawn on a panorama of the camera's field of view. This interface was developed further in the Demonstrate project, which allowed online participants to zoom in to frame and photograph activity in the Sproul Plaza at UC Berkeley [12]. In addition, control buttons were added to the interface to allow for fine-grained control of the selected viewing frame. The interface used for Demonstrate was adapted for use in the first iterations of the CONE project that have been the inspiration for the CONE-SF client [10].

Autonomous systems that can track and record activity are widespread in the field of video surveillance and many such systems have been proposed [4][5]. In applying this concept to wildlife Steward et. al describe a system for video surveillance and monitoring of mammals in their study of the European badger [6]. Another prototype of the CONE project, ACONE (Automated Collaborative Observatory for Natural Environments), attempts to capture images of the thought to be extinct Ivory Billed Woodpecker by recording and analyzing video sequences [11]. All these autonomous systems rely on motion segmentation algorithms to detect interesting motion, which is currently one of the most active research topics in computer visions. As a result many types of such algorithms exist including segmentation using background subtraction and temporal differencing, which we decided to experiment with [7]. Collins et. al in their system for video surveillance and monitoring employ a temporal differencing algorithm similar to the one used for the CONE-SF autonomous client [4]. Stauffer and Grimson first suggested adaptive background subtraction, as a variation on classic background subtraction to cope with dynamic environments [8]. This algorithm has since been improved upon to reduce processing power requirements so as to be able to run in real-time [9].

10 CONCLUSION AND FUTURE WORK

The completed CONE-SF system was opened up to the public on April 23rd 2007 and has since been a huge success. On the day of the launce the system was garnered local TV and radio coverage and was featured in an article in Wired magazine. Since then, CONE-SF has received lots of attention both from the news media and public. With more than 2600 registered, 5500 images taken, and 650 comments made, it seems the goal of creating a space for "online collaborative observation" has succeeded. Bird enthusiasts and scientists alike have been using CONE-SF and have gone as far as creating Blogs dedicated to using and understanding the system, which can be accessed by anyone at: http://cone.berkeley.edu

The problems of previous CONE prototypes seem to have been fixed with the new implementation. There have been no crashes and the system seems to be scaling well. With up to 20 simultaneous viewers using the camera there has not been any noticeable reduction in performance. To be cautious and to provide the best video quality possible we choose limit the maximum allowed users 20 until the system had been tested with real users and load. We might increase this limit in the future as necessary and as the system allows. The point system that was created to enhance the collaborative feel of the system seems to really have engaged users. We are continually impressed with the increasing point scores that players are able to achieve. In some instances we have even worried people were cheating until they contacted us and turned out to just be really into bird watching and the game. In a user created Blog dedicated to CONE-SF, the author explains other users every detail of the client and how the algorithms used for frame selection make controlling the camera collaborative instead of competitive.

We have however had to make some minor changes to CONE-SF. Given the new and simpler architecture of the system this has not been a difficult task. Privacy concerns made it necessary to change the panorama and thus the viewable area of the camera. This has been a relatively simple task using the panorama-generation and calibration tools we developed. Also, because of the high zoom capabilities of the camera viewers were able to look into the windows of nearby houses. To prevent this from happening we modified the client to not allow viewers to zoom in on the given area.

Recently we completed the documentation for the CONE-SF system and released it together with the entire code-base as open-source [13]. Having released the code-base and documentation for the system will hopefully persuade other research groups or individuals to contribute to the project and create their own deployment.

In there future we want to experiment with scaling the system to even more users and create deployments in new and interesting locations. In feedback received for CONE-SF people are already suggesting places they would like to have an observatory installed. Having a stable power source over long periods and a reliable Internet connection are some challenges we still need to overcome before being able to install observatories in truly remote locations, which is a key of the goals of the CONE project. We also want to

complete the autonomous client as discussed and possibly extend it with ability to automatically classify animals.

BIBLIOGRAPHY

[1] Dezhen Song and Ken Goldberg, Networked Robotic Cameras for Collaborative Observation of Natural Environments, The 12th International Symposium of Robotics Research (ISRR 2005), October 12th-15th, 2005, San Francisco, CA, USA

[2] D. Song and K. Goldberg, ShareCam Part I: Interface, System Architecture, and Implementation of a Collaboratively Controlled Robotic Webcam, IEEE/RSJ International Conference on Intelligent Robots and Systems, Las Vegas, NV, Oct. 2003.

[3] D. Song, K. Goldberg, and A. Pashkevich, ShareCam Part II: Approximate and Distributed Algorithms for a Collaboratively Controlled Robotic Webcam, IEEE/RSJ International Conference on Intelligent Robots and Systems, Las Vegas, NV, Oct. 2003.

[4] R. Collins, A. Lipton and T. Kanade, "A System for Video Surveillance and Monitoring," Proc. Am. Nuclear Soc. (ANS) Eighth Int'l Topical Meeting Robotic and Remote Systems, Apr. 1999.

[5] I. Cohen and G. Medioni, "Detecting and Tracking Moving Objects for Video Surveillance," IEEE Proc. Computer Vision and Pattern Recognition, June 1999.

[6] P. Stewart, S. Ellwood, D. MacDonald, "Remote video-surveillance of wildlife - An introduction from experience with the European badger Meles meles," Mammal Review [MAMM. REV.]. Vol. 27, no. 4, pp. 185-204. Dec 1997.

[7] L. Wang, W. Hu, T. Tan, and S. Maybank. A survey on visual surveillance of object motion and behaviors. IEEE Trans. on Systems, Man and Cybernetics, 3:334–352, 2004.

[8] Chris Stauffer, W.E.L. Grimson, "Adaptive Background Mixture Models for Real-Time Tracking," cvpr, p. 2246, 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'99) - Volume 2, 1999.

[9] P. W. Power and J. A. Schoonees, "Understanding background mixture models for foreground segmentation," in Proceedings Image and Vision Computing New Zealand, pp. 267–271, (Auckland, New Zealand), Nov 2002.

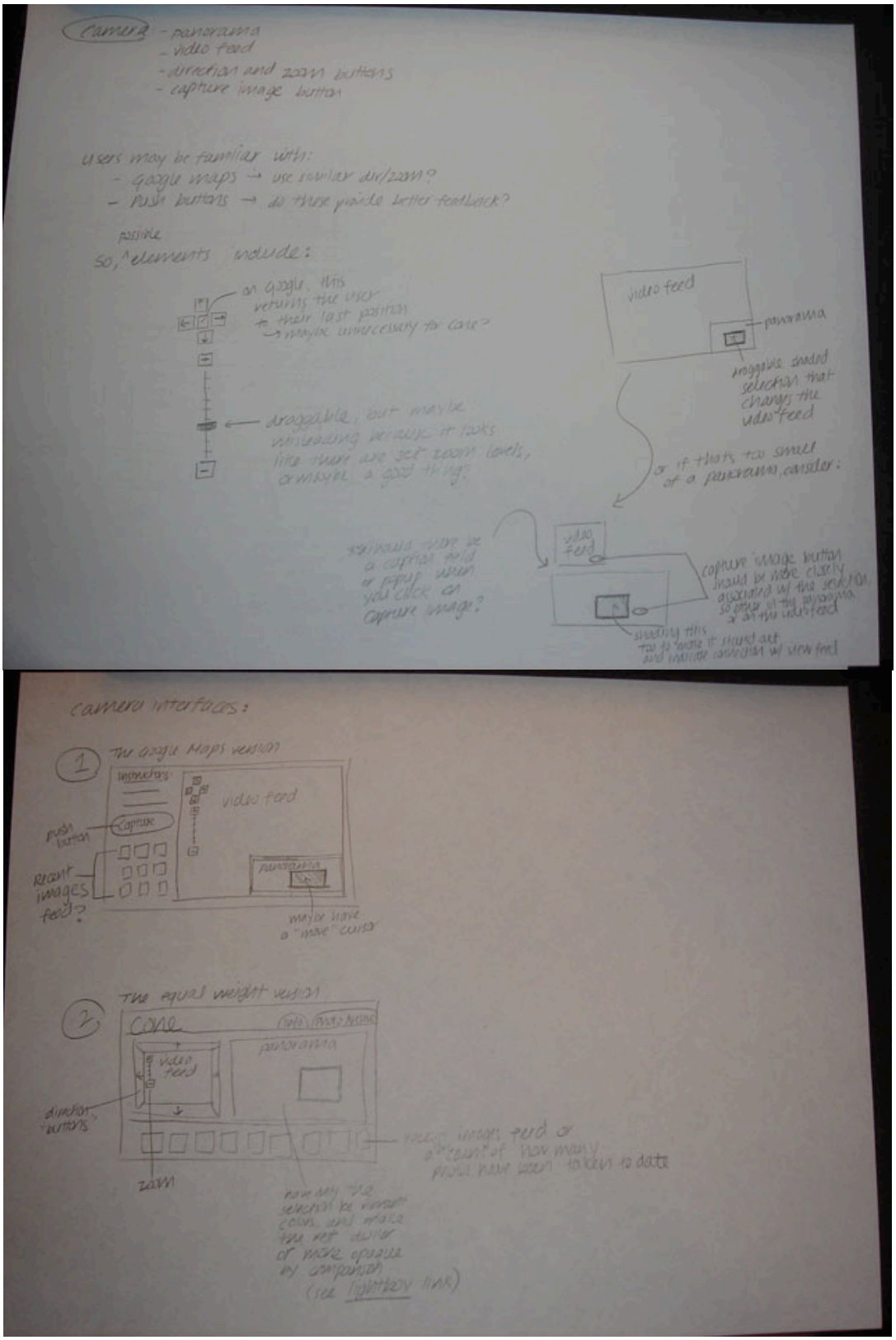
[10] Project CONE. Collaborative Observatories for Natural Environments. 20 April 2007 <http://www.c-o-n-e.org>

[11] Project ACONE: Automated Birdwatching. Assisting the Search for the Ivory-Billed Woodpecker. 20 April 2007 <http://www.c-o-n-e.org/acone>

[12] Demonstrate. <http://demonstrate.berkeley.edu>

[13] CONE Source Code and Documentation. <http://teleactor.berkeley.edu/cone/doc>

APPENDIX A – CLIENT DESIGN



APPENDIX B – WEBSITE DESIGN