INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A Distributed Vision System Using Streaming Video Interconnects

Mark Sidloi
Department of Electrical and Computer Engineering
McGill University

July 2001

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Engineering

Abstract

I present a distributed vision system for testing and development of computer vision algorithms. The system's form follows a new approach to the construction of vision systems. Utilizing streaming video interconnects and distributed computer resources, the system provides researchers with sufficient processing power to run complex algorithms under real-time constraints. The system runs algorithms in a pipelined manner, with data passed between functional units using streaming video protocols. DirectShow forms the basis for the system's processing block objects. Researchers can configure the system using functional units from a library of source, rendering, and transform processing blocks available on the system. To aid in configuring the system to the specifications of a given task, I have developed a system management tool that allows the user to configure the system via interaction with a web-based Java applet. This presents the user with a simple, object-oriented framework that is robust and easy to manage.

Résumé

Je présent un system de vision artificielle distribuer pour le développent et la vérification d'algorithmes de vision artificielle. La forme du système s'accorde à une nouvelle méthode de construction pour les systèmes de vision artificielle. Utilisant des connections de "streaming video" et des ressources d'ordinateurs distribuer, le system et suffisamment puisant pour exploiter des algorithmes complexes sous des contraints de temps réel. Le système de vision artificielle contrôle des algorithmes dans une manière en série, avec l'information passer entre les unîtes fonctionnelles utilisant des protocoles de "streaming video". DirectShow est la base des unîtes fonctionnelles du système. Les chercheurs peuvent configurer le système utilisant les unîtes fonctionnelles qui viennent d'une bibliothèque d'unîtes de source, exposition et transformation, qui se trouve sur le system. Pour aider les usagers du system a configurer le system j'ai développé un outil de contrôle qui donne à l'usager la possibilité de configurer le system avec un applet sur l'Internet. Cet applet présent à l'usage un mode d'access simple et orienté d'object qui est robuste et facile a contrôler.

Acknowledgements

I would like to thank Professor James J. Clark, my supervising professor. Dr. Clark's insight and support were invaluable to me during the course of my work on this thesis. He did not spare any effort in ensuring that the resources I needed for design, implementation and testing of this distributed vision system were always available to me.

Professor Clark is blessed with a very competent and professional core of graduate and undergraduate students working on his projects. I would like to acknowledge these individuals, who are my colleagues, for their support and understanding, especially while making use of their computing resources. Of this group of researchers, special thanks go to Ziad Hafed, whose assitance on logistical matters and questions proved very helpful to me while writing this thesis.

I would also like to acknowledge Vinod Nair and Gilbert Soucy, former colleagues and members of the McGill Centre for Intelligent Machines (CIM), for their assitance early on during my work. Though neither worked on the details of the system, both provided assistance in preparing the early versions of the system for crucial demonstrations and in developping processing blocks for the system.

Finally, I would like to acknowledge and thank my parents, sister and wife, Rachel Stephanie, for their support and patience during the course of my studies.

Table of Contents

List of Figures and Tables

Figures

Tables

List of Acronyms

This thesis makes use of several acronyms in its descriptions of the distributed vision system.

AWT	Abstract Windowing Toolkit, a standard Java library.
CR	Carriage-Return. Also known as end-of-line character.
FLOPS	Floating-Point Operations Per Second
FTP	File Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol.
JPEG	Joint Photographic Experts Group
LAN	Local Area Network.
MIPS	Million Instructions Per Second
PTU	Pan-Tilt Unit. Used on some video camera systems.
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
WAN	Wide Area Network.

Chapter 1. Introduction

Development of computer vision and image processing algorithms and testing systems has been far from monolithic. However, in many ways, developments in computer vision systems lag behind the available computer hardware. Certainly, vision and image processing systems have begun to take advantage of the processing power available on modern microprocessor systems, yet there has been relatively little development of other computing technologies in systems used by these fields of research.

Networking technology is one of the technologies that has been very underused in vision systems to date. For instance, few vision systems have been designed to take advantage of distributed computing resources. Though the processing and storage capabilities of modern microcomputers have increased greatly since vision researchers began using them for research, vision systems stand to gain tremendously from the use of multiple microcomputers running in tandem. Such a distributed system would benefit from the increased processing capabilities available, especially when running algorithms that require parallel processing or can be pipelined.

The proliferation of high-speed networking technology and widely available Internet access also allow for remote system access. Usually, single-user vision systems run on single microcomputers can only be remotely accessed if the microcomputer is running an operating system that allows remote access. In those cases, the vision researcher must also be using a computer with a compatible operating system of log in scheme to access the computer that the system can run on. This also assumes that there is no firewall preventing remote access to the system's computer from the researcher's location.

In contrast, a system designed for Internet access can be built with a user interface that is accessible from anywhere. While this does necessitate some security precautions, it will allow researchers to access their work, perform tests and view results from any location with an Internet connection.

Internet technology can also expand an existing distributed system, by allowing the system to incorporate microcomputers that reside on different networks to function within a single system.

The use of streaming video technology is also seldomly seen in vision systems. Partly, this is due to perceived drawbacks to using streaming video, such as low quality video capture devices or lower image resolutions. Most of these contentions are actually unfounded, as moderate quality video capture devices do indeed exist, and even low quality video capture devices can often provide medium image resolutions. The detractors of usage of streaming video often ignore the benefits of streaming video. Streaming multimedia is a natural fit to any vision system designed to run algorithms under real-time constraints. It also does away with the processing requirements for systems that use software to poll video capture devices to pick up still images, which effectively functions as a video stream anyway. Further, video capture sources for streaming video are widely available, relatively inexpensive, and far more useful interesting algorithms on real-world situations rather than on previously stored sets of digitally-enhanced images.

This thesis presents a new approach to the design of vision systems – a new object-oriented framework for running vision algorithms. The system described in this thesis is a distributed vision system. Algorithms and non-algorithmic processing blocks in this system are treated as objects, with input and output access points for information interchange. The objects use streaming multimedia data types for all data exchanges. The system uses Microsoft's DirectShow technology for object design and dataflow control and management. The system's user interface is designed to provide the user with a simple yet powerful medium for controlling the vast computing resources available through the system.

Chapter 2. A Review of Related Work

Though I am unaware of any existing framework for vision algorithms that is quite like the distributed system discussed in this thesis, several systems and software platforms bear similarities to the distributed system. The idea of using distributed computing resources in design is hardly a novelty, even in vision research. Pipelining and parallel processing, both design strategies that lend themselves very well to distributed computing, have been used in computer-based system design for over a decade, and both have affected the design of vision systems.

The distributed system's user interface presents the user an object-oriented framework for testing vision algorithms. Object-Oriented design, like distributed computing, is another area where a fair amount of development has taken place. At least one major vision algorithm development platform employs an object-oriented interface. As well, several competing software packages have been released in the last decade, offering programmers and system developers access to object-oriented design.

Streaming multimedia technology, its protocols and data types, used for data interchange by the processing blocks in the distributed system, is also not a new technology. Though vision research has been slow to pick up on it, several non-vision systems have used it for distribution of video data.

The following review is not meant to be a comprehensive review of all work done in the areas of research that the distributed system touches upon. Rather, it is meant to demonstrate that the distributed system discussed in this thesis is a combination of several older ideas that have been combined to form a unique system.

2.1 Pipelined and Parallel Processing Systems

2.1.1 The PIPE System

The pipelined approach to system design has been in use for decades. One vision and image processing system, designed in the last decade, to take advantage of pipelining was

the PIPE system or Pipelined Image Processing Engine. PIPE was conceived by Dr. Ernest Kent of the National Bureau of Standards [1], now known as the National Institute of Standards and Technology. Designed for real-time or near-real-time performance of functions such as edge extraction, motion detection and disparity map generation, the PIPE was a coarse-grained parallel system designed with three to eight modular processing stages (MPS) for data processing.

Like the distributed system described in this thesis, the PIPE system was designed to perform in a large number of wide ranging applications, all centred on image processing and vision processing. Unlike the distributed system, the PIPE system was designed to function on a dedicated hardware architecture and was not designed with much scalability in mind. The PIPE system could use between three and eight modular processing stages for computations. Each of these stages consisted of input look-up tables, a three input arithmetic logic unit (ALU), two frame buffers, a pre-neighbourhood single value look-up table operator, two 3x3 neighbourhood operators, an output ALU, a two value look-up table operator and dataflow crosspoint switches.

The PIPE system also used video for input, but rather than streaming video, it took input in the form of analog video and then its input stage converted the video into a digital signal. This is where the PIPE system's similarities to the distributed system end. Unlike the distributed system, PIPE was designed to run on application-specific hardware, rather than general-purpose microprocessors. Though it is referred to as a parallel system, it was not designed to allow for all possible parallel processing configurations. The output of each modular processing stage is hardware as an input to the next and the previous modular processing stage. In contrast, the distributed system discussed in this thesis was able to run any number of algorithms, assuming that sufficient processing power in the form of general-purpose computers, was available on the system. Further, the distributed system allowed for any configuration.

Unlike the distributed system, PIPE's user interface was not object based, and required PIPE's users to program their algorithms into the system using PIPE's assembly

language. However, despite the difference, Kent recognized the necessity for a suite of base software algorithms on a vision and image processing system. Though PIPE and the distributed system differ on the algorithms they have available, both systems incorporate base software. In the case of the PIPE system, the base software package included Roberts and Sobel operators, several motion and edge detectors, morphological operators, min/max operators and stereo disparity operators. The PIPE system also provided video capture access to the users, though, unlike the distributed system, it did no allow the user to select between several different types of video capture devices and did not support digital video devices.

PIPE's architecture set an upper bound on its processing abilities. Though the distributed system's architecture also a maximum upper bound on its processing abilities – two hundred microcomputer systems – the distributed system's theoretical maximum processing ability far exceeds PIPE's.

2.1.2 The AIPA System

A more recent system that more closely resembled the distributed system was the Advanced Image Processing Accelerator or AIPA system [2]. Unlike PIPE, AIPA was designed to run using general-purpose processors. Unlike the distributed system, AIPA required that these processors run from four Processing and Storage Modules plugged into a 6U VME64 motherboard instead of microcomputer systems.

The AIPA system itself was an incompletely specified system. It was designed to compose the lowest layer in a three layer image processing system. No user interface was designed for the AIPA system, though its designers recommended the Khoros Visual Programming system, an object-based system that is discussed in the next section, for the user interface. Since no user interface was explicitly designed or selected for AIPA, the middle layer, needed for interfacing AIPA and the user interface was not designed either.

The AIPA system's designers understood that in the field of image processing and computer vision, "the processing needs of end users are not being met. This trend will only increase with the added performance demands of new algorithms."[2]. For any system to successfully run these algorithms under real-time constraints using lower cost general-purpose processors, a distributed solution utilizing multiple processors must be implemented. AIPA, like PIPE and the distributed system, was designed to fit a wide range of industries and applications.

2.2 Object Oriented Systems and Frameworks

2.2.1 General Object-based Systems

Along with the proliferation of object-oriented programming languages, several object-oriented system controlling architectures have been developed over the last decade. In object-based systems, each software-based processing unit is treated as a separate object, complete with it own data input and output access points. Some of the more common object-based systems that are used for object-based are CORBA, DCE and COM/DCOM.

CORBA, or Common Object Request Broker Architecture, was designed by OMG, the Object Management Group, a non-profit consortium created in 1989 with the purpose of promoting theory and practice of object technology in distributed computing systems [3]. CORBA follows OMG's Object Model. It specifies a standard for providing interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. OMG's Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. Under this model, objects are treated as servers in a client-server system, and client programs can request information from these objects through an object interface. Figure 2.2-1 depicts the main components of CORBA and the method used by client software for interfacing with an OMG object under CORBA.

The CORBA standard is quite popular, and is the most often used system for distributed systems on microcomputers. It is a very general standard, and does not constrain the data

types used by objects for information interchange. CORBA itself is a standard, not a system. There are many implementations of CORBA currently available. They vary in their degree of CORBA compliance, quality of support, portability and availability of additional features. Unfortunately, to date there are no fully compliant public domain implementations. ILU, from Xerox Parc, is the most-compliant CORBA implementation available in the public domain. Some commercially available packages are fully compliant with the CORBA standard. Some of these, such as VisiBroker from VISIGENIC offer interoperability with Sun Microsystems Inc.'s Java programming language.

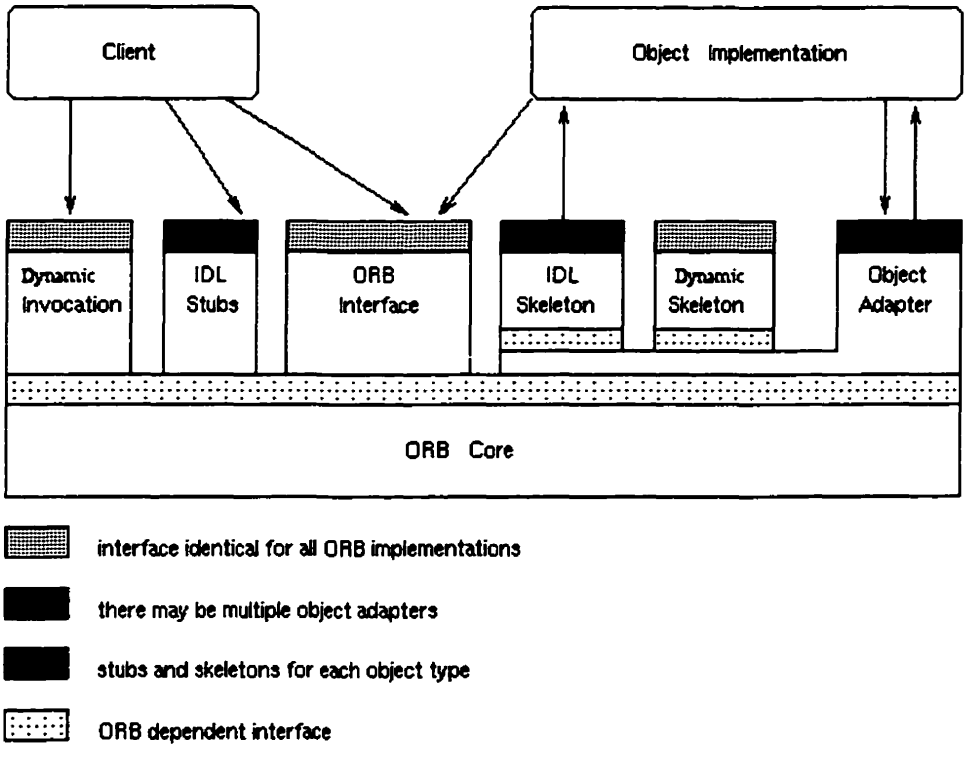

Figure 2.2-1: The main components of the ORB architecture and their interconnections

DCE, short for the Distributed Computing Environment, is another distributed system technology. Both DCE and CORBA support the construction and integration of client-server applications in heterogeneous distributed environments. Though DCE is slightly older that CORBA, it is slightly more stable and scalable than the CORBA standard. It

also was designed with better security features than CORBA. The main difference between the two approaches to distributed computing was that CORBA was a complete redesign of the principles of distributed computing whereas DCE was a tightly integrated package of existing technologies. DCE is still often used in true large-scale, enterprise-class applications [5].

A third standard that has emerged as a competitor for CORBA is COM/DCOM, Microsoft Corp's Component Object Model. Originally designed only for use on Microsoft operating systems, DCOM, or Distributed COM, is in the process of being ported to the Linux operating system[5]. Unlike CORBA and DCE, COM was not simply designed as an open standard in search of implementation. Microsoft developed many implementations of its COM architecture. One of these implementations, was DirectShow, one of Microsoft's DirectX drivers. DirectShow provides a framework for object-oriented design. DirectShow, and its relation to COM, is discussed in greater detail in section 3.4.

2.2.2 Object-based Systems used in Vision Research

Object-based systems are not foreign to vision systems. The most well known object-based vision system developed was the Khoros Integrated Development Environment. Designed by Khoral Research, Inc., Khoros was a software integration and development environment. Included with the system was a suite of software development tools and a vast collection of image processing, data manipulation, matrix processing and scientific visualization applications [6]. The Khoros system also included a user interface called Cantata.

More than a simple user interface, Cantata was a visual programming environment. It allowed users to selectively load applications from among the various applications available as data processing objects on the Khoros system, and then configure interconnections between the loaded objects. These connected objects, or *glyphs*, as they were referred to in the Khoros system, would form complete algorithms.

Though Khoros never was the most popular vision system available, its worth to distributed vision systems, such as AIPA, is quite well documented [2]. Though not restricted to a specific operating system, Khoros' design favoured Unix environments.

2.3 Distributed Systems using Streaming Video

Considering how long technologies such as television and cinema have been standard forms of entertainment in western cultures, it is by no means surprising that streaming video has emerged as a popular network-based technology. Several systems have sought to take advantage of streaming video technology. Some of these include Video-on-Demand, using TCP/IP socket connections, and distributed systems to provide quick access to data stored on video servers [7]. Video-on-Demand systems have been designed by Berkely systems, Stoney Brook Video and the Expertise centre for Digital Media at Limburg University Centre in Belgium. Other streaming video systems, such as the Excalibur [8] Analysis Engine, have been used for video storing, indexing and cataloguing.

These examples are merely the tip of the iceberg. As transmission bandwidth increases, the demand for streaming multimedia will only increase. Already, some Internet websites exist that provide media content using streaming video. This content ranges from alternative news sites to free low-budget television over the Internet. The proliferation of streaming video technology over the World Wide Web has caused Internet browser software to be coupled with multimedia software. Given this, a vision system designed with user access over the Internet using Internet browser software for system access could be designed under the assumption that any user with a browser capable of logging in would have access to multimedia software. Though accessing that software might be difficult, at least the assumption that it is present could aid in system design.

Chapter 3. High-Level Design and Design Theory

This chapter focuses on elements of the design of the system discussed in this thesis. First, a more complete description of the problem than the one presented in the introduction is put forward, to justify the decision of designing this system. Next, a high-level description of the system is presented, including an explanation of how this system addresses the problems described at the beginning of this section.

3.1 Problem Description

Vision systems that use microprocessors to run image processing and computer vision algorithms are not a new phenomenon. In the past, one of the major barriers to development in this area was due to the available tools – either lack of adequate computing power, or lack of ability on the part of the microcomputers to access resources or desired input data. When microcomputers were first introduced, they lacked the hardware to handle full colour images, and so the only workable options required greyscale images. The first general-purpose microprocessor systems also lacked the processing power needed to process images quickly, precluding their use in robotic systems. Finally, the early microprocessors lacked both sufficient storage capacity and network connectivity needed give them access to sources of data. Though the capabilities of microprocessors grew very fast, their use in vision systems was practically non-existent.

About ten to fifteen years ago, the computational power available in general-purpose microprocessor systems reached a point where they could begin to be considered for tasks that were previously the domain of application-specific processors and digital signal processors in several fields, including image processing and computer vision. Since then, general-purpose microprocessor systems have been used increasingly in research and development of new techniques and algorithms related to image processing and computer vision.

These systems have several associated properties that make them useful to researchers in the field of artificial vision. They are easily available, as they are mass-produced for the public, and the cost of these systems has been falling for some time. They are powerful enough to process images under real-time constraints, which is useful in many vision applications. They can access visual data from many different sources, such as peripheral devices or network connections. They also have software tools that aid programming and debugging algorithms. They also have sufficient storage and memory capacity to handle large image files and sufficient graphical processing power to process and correctly render results.

Generally, efforts in the field of computer vision performed on general-purpose microprocessors have tended to follow a similar path. Most work in this field has been concentrated on single algorithm systems, where a single stand-alone application tests one main algorithm by applying it to image data. Image data for these applications tends to consist of still images, either a single image or a series of still images, though streaming video data format have seen increased usage. It is likely that this approach to image processing and vision algorithms remains widespread for several reasons. One reason is the pace of change in microcomputers. In terms of processing power, storage capacity, image rendering and handling, and data acquisition, the abilities of microcomputer systems have grown very quickly – faster than researchers' ability to adapt to those changes and take advantage of the full potential of their computer systems. While it would be false to suggest that researchers in computer vision have not taken advantage of the increased potential of their microprocessor systems, the main potential they seem to be exploiting in most of their work is the increased processor power and storage capacity, which they use to produce algorithms that would have been too computationally expensive for older microprocessor systems. Though this is significant, it still remains that few of the new capabilities of microcomputers are being exploited for vision research. For instance, few vision researchers have taken advantage of the new multimedia capabilities of microcomputers, such as streaming video. While there are drawbacks to using streaming video instead of using still images, those are often much

less important than the advantages that using these techniques afford. This will be discussed in further detail in section 3.2.6.

These factors have combined to bring research in vision and image processing to the point it now stands at. Most software written in vision and image processing research is written to test only a single algorithm, or perhaps two. Generally, the software is written as stand-alone software and must include software code to access input data, usually from a file, and store results in an output file. Any other lesser, "helper" algorithms required by the main algorithm, such as a simple edge detector required for the main algorithm to run, must be either added into the source code before the software is compiled or must be run on the input data separately and report result to a file that is later used as input to the main algorithm. This method of writing vision software produces code according to one of two programming paradigms – statically connected integration and quasi-dynamically-connected integration. Either the code implements a single algorithm, or statically links several algorithms. In the case of the latter, generally only one of the algorithms is the main one under review. If the algorithms are linked, dynamically one might say, at run time, they are linked either by passing data through files, and not run simultaneously, or if they are run simultaneously, such as with the use of Unix pipes, the configuration of the system of algorithms cannot be altered while the algorithms are being run. I refer to the later paradigm as quasi-dynamic since though the integration takes place after compile-time and allows for the system of algorithms to be configured and re-configured at a later time, and so it is not static, the integration must take place before run-time and cannot, in the cases I describe here, be reconfigured while the algorithms are being run. A truly dynamic system would allow the researcher to make modifications to the configuration of the system, including the loading or removal of algorithms from the system, while the system is being run.

These two paradigms that have dominated vision and image processing research thus far, place unnecessary constraints on vision and image processing software. First, they imposes a de facto limitation on the researcher's choices in terms of sources of data, by forcing the researcher to write the software that accesses input data. While it is true that a

researcher could write software to access peripheral data devices, local-area network (LAN) and wide-area network (WAN) based cameras and remote, web-based sources of video, it is also true that the amount of effort expended to write software to access these sources of data is often much greater than the effort required to write software to access data files. As well, software used to access each of these sources of data is not identical – software designed to access network resources shares little in common with software designed to access local resources and peripheral devices. In the end, the additional effort required to access these sources of data under the software design paradigms traditionally used in image processing and computer vision research generally precludes their being used in vision and image processing software.

These paradigms also serve to reinforce the notion of testing no more than a single algorithm per software application. In a statically connected system, it would be difficult to test several algorithms, unless they are being compared against each other. One must be sure that all other algorithms, except the one being tested, are reliable enough to function as required. If several algorithms were being tested in a statically-connected system, then without output at intermediate stages, a researcher would have no way of determining in which algorithm fault lay if a test failed. While intermediate outputs would, perhaps, remedy the situation, assuming rightly that the researcher knows what output is expected at the end of each individual stage, it still follows that, had the researcher known what outputs to expect at the end of any given stage, then why would he or she not test each of those algorithms separately to determine that each one functions as intended and produces the desired effect ?

This is a very strong argument for modularized software code – smaller software applications that perform specialized tasks. Another is the ease of re-use. Integrating several pieces of software code that is written as modularized procedures and subroutines may aid the development of a statically connected system of algorithms. However, "copy-pasting" software modules into a single application's source code is far more cumbersome and thus, a researcher may be forced to choose between using a sub-optimal set of helper algorithms or spending the extra time, compared to the dynamically linked

approach, needed to integrate and then excise each helper algorithm in an effort to find the most optimal set of helper algorithms. Dynamic linking allows a researcher to find the best-fitting "helper" algorithms for an algorithm under design without having to take the effort of adding code to the applications source code and correctly integrated it before compilation to test out the compatibility of the helper algorithms with the main algorithm's requirements, and the effort required to excise that same helper algorithm from the application software if it proves less compatible with the main algorithm's requirements than an alternate helper algorithm.

While these arguments may support the use of the second paradigm detailed above – namely the use of separate programs linked together by pipes – it should be noted that it this paradigm also reinforces the testing of only a single algorithm. Generally, powerful vision algorithms take full advantage of the processing power that has only recently become available on microcomputers. Often, innovative image processing and computer vision algorithms are too computationally expensive to share a single microcomputer's processing resources with one or more other computationally-intensive algorithms and still function under real-time constraints. For algorithms that have no real-time constraints, there is little need and few advantages to testing more than one algorithm at a time. There is a benefit to parallel testing of several real-time constrained algorithms with similar functionality, such as testing several object tracking algorithms that use different approaches to arrive at the same desired result. However, parallel testing of such algorithms without real-time constraints is not preferable to testing those same algorithms separately and then comparing the results. Once real-time constraints are no longer an issue for the researcher, the algorithms do not need to keep up with the incoming data rate. Further, results produced by algorithms that do not run under real-time constraints must be written to output files and be reviewed after the processing is complete for the results to be properly examined.

Not only do these two paradigms promote one source of input data – files – over other forms, they also serve to either to constrain the processing power available to the researcher to a single machine or forcing the vision researcher to set up a complicated

and delicate networked system or leaving the researcher no alternative other than to abandon any real-time constraints on the algorithm(s) under development and/or review.

3.2 The Alternative

In the previous section, the current state of computer vision research software was presented. Researchers are often forced to choose between spending many man-hours on designing software that can take input from multiple sources or to select a single source of input, files usually, and use that source exclusively in developing and testing algorithms. Usually, researchers are limited in the computing resources they have at their disposal to test any given algorithm, especially if they only have personal computers at their disposal. As well, since vision software usually ends up taking the form of stand-alone applications, helper algorithms, such as edge detectors, which must be integrated into a single application with the main algorithm under review. Further, testing multiple algorithms in a single application is difficult, unless each intermediate stage within the application can be monitored by the researcher so that the researcher can fine tune the algorithms individually.

There is an alternative to this. A system can be designed in software, to provide vision researchers with an environment upon which they can test their algorithms. The system would grant access to multiple sources of input, allowing researchers to make use of more practical sources of input data upon which to test algorithms and saving them the time required to write input/output software to interface with their algorithm. To accomplish this, data capture software for each data input source would be part of the system and this software would be loadable on the system, where once loaded it could be connected to the researcher's algorithm processing blocks.

Under such a system, each algorithm would be treated as a processing block, with input and output points. In actuality, the algorithms would either be written as stand-alone programs or as software executable from within another program such as a dynamically

linked library. Some standards would need to be set for these processing blocks such as for data types. The system would also need to be able to handle dataflow issues and be able to configure each processing block so that any buffer allocation within a processing block would be handled by the system.

The system could be further improved by running on multiple computers at once, providing seamless control over multiple microprocessor systems. The vision researcher will be able to view the entire system as one execution environment, upon which computer vision algorithms can be run. The system would need to be intelligent enough to keep track of the data types in use by the various algorithms and would need to handle any handshaking that would take place upon inter-connection between algorithms. Further, for this system to be of any use, it must be able to display the results of data processing at various nodes within the user-configured system of processing blocks.

In the next few subsections, the alternative approach proposed here is explained in further detail. The subsequent sections in this chapter describe the system requirements in terms of high-level design, and then proceed to detail the solution approach chosen and a high-level description of the implementation of the distributed vision system.

3.2.1 Algorithms as Processing Blocks

Representing algorithms as modular processing blocks has several advantages. First, it allows each processing block to be tested and validated independently. It also allows for easy re-use of algorithms. For instance, a researcher is researching several ideas for vision-based algorithms may require simple edge detection to be performed on input before the algorithms are applied to the data. The researcher could first design an edge-detection processing block and test it independently to insure that it was in good working order and functioning according to the needs of the researcher's algorithms. Once the edge detector was shown to be in working order, the researcher could then test out each algorithm he/she designed that required the edge detection function. There would be no need to integrate the edge detection into the software into the software code of every

algorithm that required edge detection; simply loading the edge-detection processing block onto the system, and then loading the algorithm that required it and routing the output of the edge detector to the algorithm under design/study would be adequate.

Designing algorithms into modular processing blocks would also aid researchers in combining their work or in collaborating on research projects. Once a standardized choice for input and output data types would be agreed upon, several researchers' processing blocks could be joined or linked just as the edge detection software was linked in the previous example. As well, a researcher could easily re-use old processing blocks, both those designed by him/her and those designed by others, without the need to spend time to review the specifics on the input/output data handling that takes place within those blocks.

Standardizing algorithms as processing blocks also serves another purpose that benefits the system's design. By writing all algorithms as processing blocks that must adhere to a standardized format, the design of the system's controlling software can be simplified. By using a modular design approach, the system will not need to retain specific information on the inner working of each algorithm. The benefit of this cannot be understated. Under this approach to vision software, the system, which controls the processing blocks, must control each processing block. It must be able to instruct processing blocks on setting up any buffering the processing blocks may need to handle input/output data. As well, it must control dataflow between the processing blocks. Finally, the system must be able to access any significant information on each processing block such as the number and type of data input/output points on each processing block.

Defining a complete programming structure for the processing blocks can be taxing on its own. The format for this programming structure by which processing blocks must be written must be both open enough to accommodate all of the needs – both in the present and in the foreseeable future – of the system described in this chapter and the researchers who will use it and yet closed enough to truly standardize the format for writing processing blocks. Some software packages currently available define such formats for

developing processing blocks. Aside from the Khoros system and the non-Vision-specific packages, such as CORBA, discussed in chapter 2, there is Microsoft's COM and DirectShow. A discussion of the strengths of DirectShow appears in section 3.4.

3.2.2 Central Controller

In order for a system, such as the one described above, to load various processing blocks and then control their operations and their inter-connections, controlling software must exist. In a multi-computer system, such as the one proposed here, the controlling software can either be fully centralized or can be partly decentralized. Both of these choices require a central control hub to relay commands from the system's users to the processing blocks and present information and system updates to the user interface.

It is worthy of note that a true fully decentralized control technically, is difficult to build and manage. As long as there is some central controlling software, a system cannot be referred to as fully decentralized. Any multi-computer system that utilized decentralized control programs, running on each computer within the system must still interact with one or more users to be of use. If a single user is present, then that user's interface to the system would serve as the central controller, and the system could no longer be considered fully decentralized. If multiple users were accessing the system at the same time, then either one would have priority in controlling the system – and that user's interface would be the central controller for the system – or all would have some control of the system. Only the last case can be considered a fully decentralized system. In this case, some agreement between the system's users on sharing the resources would be necessary. Further, the system's user interface would need to be designed so that it would be able to find and connect to all the computers on the system. Finally, for the fully decentralized system, all of the user interface consoles logged into the system would collectively function as the central control mechanism for the system and the system would effectively function as a partly decentralized system.

Note also that in a system where multiple users can access a system with no dedicated central controller and no dedicated user connections to decentralized controlling units, connection-oriented sockets could not be used as a means of transmitting commands from the users to the decentralized controlling units. Unless a means other than sockets is employed, such a system would need to use connectionless socket protocols, and those tend to be very unreliable. For this reason, decentralizing the system to the point that no dedicated central hub exists is no very practical.

The controlling software required for a fully centralized system is far more complex than software designed to function as the central hub in a partly decentralized system. Under both paradigms, the central controller must keep track of processes loaded on the system. It must be able to relay user commands to the appropriate computer in the multi-computer system and return system updates to the user. As well, in a multi-user environment, it must be able to handle multiple user connections. The controller must keep track of all of the processing blocks available on the computers within each system and must also keep track of all of the processing block currently loaded on each computer within the system and on the connections between their input and output points.

In order to make the system as seamless to the user as possible it must make it possible for the user to view the entire system as a single environment for running processing blocks, rather than a set of environments present on several computers. However, in order to balance the processing and memory load on each computer in the multi-computer system, the central controller must keep track of the locations where the processing blocks are loaded and must either use that information to balance loads on the system or present that information to user(s) and allow the user(s) of the system to handle load balancing.

In addition to these functions, a fully centralized controller must handle all dataflow issues and all micromanagement of processing blocks. As well, it should be noted once again that the processing blocks may or may not be written as stand-alone applications – and even were they fully executable applications, there would necessarily need to be a

standard control interface for the controller to access them. A fully centralized controller would require the ability to access these interfaces. Further, in the case of processing blocks designed as dynamic link libraries, the fully centralized controller would need to set up an environment on each computer in the system in which those processing blocks could be loaded and run.

Finally, it should be noted that using a fully centralized approach incurs a delay between the issuing of micromanagement commands and their implementation. In a fully centralized system, every dataflow command must be issued from the central controller. Unless dataflow commands can be designed as unidirectional, without requiring acknowledgement and updating messages from the processing blocks – a situation which is highly improbable – there will be a delay between the time a processing block issues an update to the central controller and the time it receives new instructions. This delay is equal to twice the transmission latency between the central controller and the computer where the processing block is loaded plus the processing time required by the controller to determine the next instruction to send. While the latter delay is generally negligible compared to the transmission delay and likely will show up in a partly or even fully decentralized system, the transmission delay can be considerable, on the order of tens or even hundreds of milliseconds. This is a clear disadvantage of a fully centralized system.

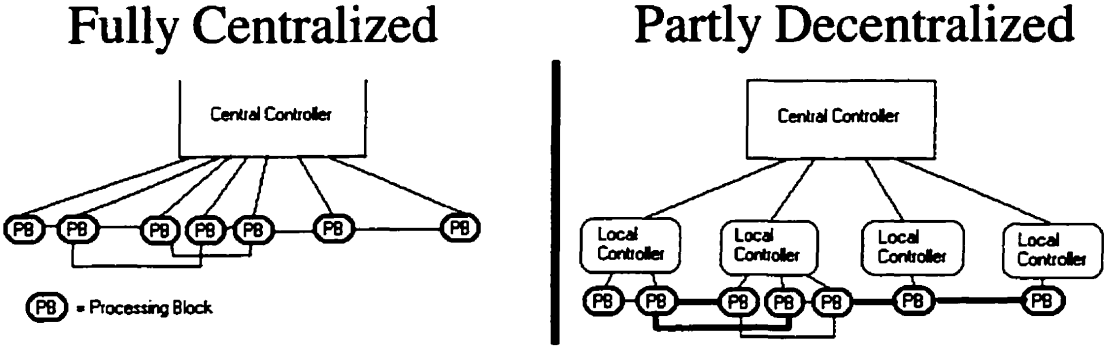

Figure 3.2-1: Fully Centralized and Partly Decentralized systems

The decentralized approach does away with much of the software code needed under the fully centralized approach. In the decentralized approach, only those functions that must be maintained from a centralized location are present within the central controller. Each

computer in the system must have its own local controller to handle local dataflow and handshaking issues. The central controller must be able to make connections between processing blocks on different systems, but all local micromanagement is handed locally on each computer within the multi-computer system.

This approach allows for far greater simplification of the design of the central controller. It means that the central controller can be written as a server with threads or processes spawned as needed to handle each connection. Local controlling software running on each computer in the multi-computer system could gather data on available processing blocks, handle local dataflow and keep track of all processing blocks loaded locally. The controlling software on each machine in the system could then connect to the central server and make this information available to it, and from there available to the system's user(s).

Though the central controller in a partly decentralized system must be designed to take into account the requirements of the controlling software that runs locally on each computer under the system's control, the central controller must also be designed in accordance with the system interface that will be presented to the user(s). A system such as the one described here can be designed to service a single user at a time, or it can be designed with functionality in place for multiple users. The user(s) may require a constant link to the system's components, or may only need to have access to the system periodically to check on testing results. The user access to the system may be set up to function only on a designated dedicated location or locations, or the user(s) may require or desire access to the system from a range of possible locations. While these issues are dealt with in the next section, it is important to mention them here since the decision as to the exact style of control granted to the user interface impacts directly on the design of the central controller.

If the user interface is designed to be constantly connected to the system and if the system is designed to be accessed and controlled by only a single user at any given time, then it is possible to build the controlling software into the user interface. In other situations,

where either the system is designed for multiple users or the user interface is not designed to be constantly in contact with the system, the user interface must be a separate system, though there will be duplication in some of the information stored in the user interface and the central software. This duplication arises from the common needs of both the user and the central controlling hub to monitor the system, including the processing blocks loaded onto the system.

3.2.3 User Interface

The previous section touched on some of the design choices that exist when designing a user interface to control a multi-computer system such as the one proposed here. The choice of design for the user interface is of paramount importance in any multi-computer system. It is also likely the least constrained design choice. The design of the central controller is by far the most constrained choice – as the previous section explained, only two paradigms for central controller design are really valid, and of the two, only one is an optimal solution. As for the software that runs on each computer within the multi-computer system to interface, once the design of the central controller is finalized, there is little left to decide as far as the software requirements. All that remains is to find a suitable method for controlling the processing block. While there is more leeway in selecting the format for writing the processing blocks than in making design choices for the software that interfaces between the processing blocks and the central system, the designer of a multi-computer system for vision algorithms has to choose between designing a customized format or making use of one that already exists. In terms of effort and reliability, it is far more advantageous to make use of pre-existing software packages that have already undergone rigorous testing, than to develop a new software package. This is especially true in the case where the system's designer is not the only person who will be writing software for the system. In that case, the availability of documentation and outside expertise to aid other researchers in designing processing blocks using a pre-existing package far outstrips the benefit of developing a proprietary package. As well, the availability of existing processing blocks on available software packages for such things as data acquisition is an added bonus when deciding to use an existing software

package rather than design a new one. This, of course, assumes that the pre-existing packages are adequate to handle the types of algorithms that the researchers who plan to use the system will require. Thus, if the assumption holds, the only real question in design when it comes to the processing blocks is selecting the most appropriate software package and the most appropriate operating system platform (Linux, Windows) for the system's loaded processing blocks to run on.

The design choices for the user interface are largely unconstrained by the choices made for the other components, or by most other considerations. To an extent, the user interface's appearance can be considered as depending on the information made available to the central controller and on the design of the structure of the processing blocks. On the other hand, the case can be made that the relationship between the these components and the user interface's appearance is actually the reverse – the design choices for the type of information available to the central controller is dependant on the needs of the user interface and the researchers who will use it to access the system. As for other design choices vis-à-vis the user interface, the only real factor in determining the optimal choice is the needs of the system's user(s).

For instance, the user interface can be designed to admit either only a single user or multiple users. Restricting system access to a single user simplifies the system design. In a single user system, the central controller does not need to go to any effort to stop one user from ruining another's tests being run on the system. It merely must prevent any users from logging in after one user has already logged onto the system. This design choice, however, can be dangerous. If anything were to happen to abruptly break the connection between the user interface and the system, the user would be cut off from the system but the system might still consider the user to be logged in. This means that were the user interface not designed to be launched only from a dedicated console, the user would not be able to log on to the system again, and neither would any other user. As well, a single user system would make it difficult for researchers to demonstrate their work using the system. If only a single user could log onto the system at a time, any demonstration of algorithms would have to take place at the location from where the

researcher was accessing the system. Further, if the system were designed for access only from a dedicated console, demonstrations of algorithms on the system at conferences or at any remote locations such as other universities would be impossible without reconfiguring the system to accept a new console as the dedicated log in console.

Multi-user systems have a different set of advantages and disadvantages. First, though, one must define what is meant by a multi-user system. A system with a set number of dedicated user interface consoles or a system that accepts input from a small pre-determined number of users may function very similarly to a single user system. On the other hand, a system that offers far more open access, will be of a different order than a single user system. For the purpose of this chapter and those that follow, a multi-user system will be taken to mean a system that is open to a large number of users and is designed to function while one or more of them are logged in to the system. Under such a definition, the term "multi-user system" could still refer to a system that requires dedicated consoles for user access, as long as there are many such consoles. The reason for defining the term so loosely is simple – there are several possible design choices when it comes to designing the user interface for a multi-computer system such as the one described in this chapter. One of those is the method of accessing the system, which can range from dedicated access points to worldwide access. The decision on the method of access is linked somewhat to the decision on the number of users the system is designed to accommodate. However, these are two distinct criteria for determining the type of user interface to design for a multi-computer system.

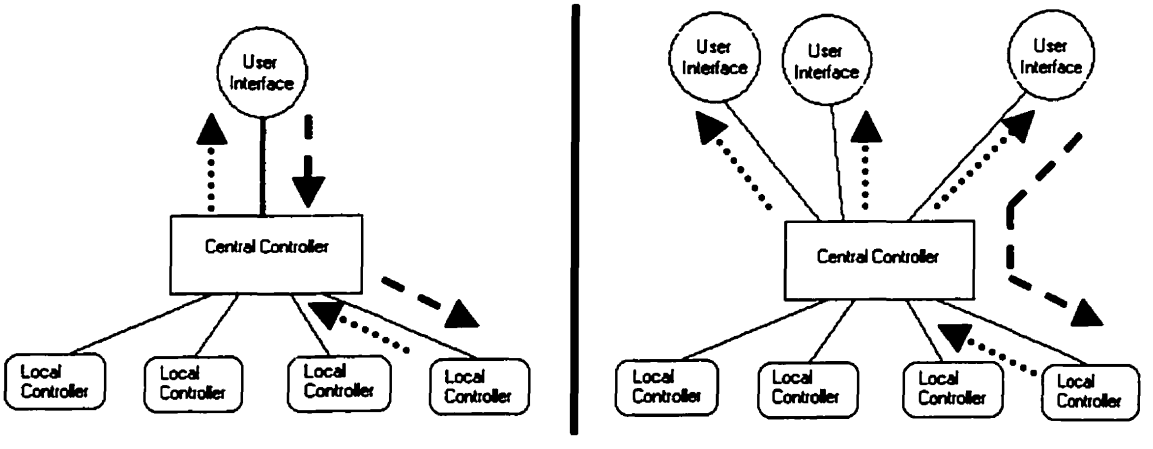

Figure 3.2-2: Single-user and Multi-user systems with datapath indicated for a single command/reply

As mentioned before, a multi-user system designed for a small number of users can be treated like a single user system. Basically, if only a small number of researchers have access to the system, they will likely be able to resolve any differences or any disputes over access to the system themselves and are unlikely to ruin each other's work on the system. Thus, such a system could be treated like a single user system, allowing each user full access to system commands without making restrictions to prevent tampering. Further, on such a system, it may not even be necessary to assign different usernames to each researcher, or to keep a log of the activities of all users of the system.

A system designed for many users cannot be so liberal in granting full access to all users. The assumption here is that there are too many researchers using the system for them to take the time and resolve disputes over such things as allocation of system resources. In these cases, either the user interface or, more likely, the central controller, will need to take steps to insure that the users do not trample on each other's work inadvertently. For instance, the system will need to employ a mechanism to prevent tampering with another researcher's configured network of processing blocks. One method for doing this is for the system to keep a record for each processing block loaded on the system of the user that loaded it onto the system. Once done, the system could restrict access to modify any processing blocks to either the user who initially loaded a given block or the super-user. Other users would be able to see the processing blocks running on the system, mostly for the purpose of load balancing when loading their own processing blocks onto the system. However, they would not be able to effect changes to other users' processing blocks.

Another approach is to restrict access to commands that modify the system's network of processing blocks to one individual at a time. This is not the same as employing a user hierarchy, a concept that will be presented later in this section. Here, assuming no user hierarchy is in place, any user would have the chance to be the system's controlling user or "master user", with unimpeded access to the entire system. Though this system is inferior to the competing approach when it comes to making full use of the system available to as many users as possible, this approach is not without its advantages. Any

multi-computer-based vision system, indeed any system that allows a user to access remote computing resources for the purposes of computations and experiments, must allow its users to access experimental results. Aside from writing results to files and viewing them later, the possibility of having results available for the user to view from his/her console as they are being produced is compelling. In vision systems, results often come in the form of graphs and charts or images. In the case of images, each image can be large, and thus require a large amount of bandwidth to transmit to the user's console. While restricting each user to displaying results at a single node within his/her network of processing blocks may be a fair restriction to place on users to save bandwidth, this restriction may not be sufficient to prevent the overuse of system bandwidth if many users attempt to display results at one node each. In a multi-user, single-master system, the potential loss of bandwidth due to displaying of results, is only a fraction of the potential loss of bandwidth due to displaying of results on a multi-user system with full privileges available to all users.

As indicated before, aside from selecting the type of user access, there are other design considerations to determine. One of these is the range of locations from where the system can be accessed by users. As mentioned earlier, restricting system access to one or more dedicated consoles can simplify the design of the central controller. Doing so also adds a level of security to the system, since only authorized personnel will have access to the dedicated interface console(s). However, placing this constraint on the system will reduce the system's versatility. A system that requires user to access it from one or more dedicated consoles cannot be effectively used as a demonstration platform to showcase vision algorithms. As well, restricting access to dedicated consoles only will severely limit the number of users who can access the system and may cause a problem in terms of logistics. A multi-computer system can generally be scaled up by adding additional computers to it in order to meet increased the demands on system resources that would accompany an increase in the number of users of the system. If such a system were to use socket protocols to make connections across a network between the central controller and the other computers in the system, then adding computers to the system is simply a mater of copying necessary software onto the new system and establishing the socket

connection. However, increasing the number of user interface consoles the central controller will accept may even require a minor rewriting of the central controller software, necessitating a brief shutdown of the system each time such a change must be made. This does not have to be the case. If the system is designed to use configuration files, adding a dedicated console may be as simple as adding an IP address to a file and then copying the user interface software onto the newly added dedicated console. Even so, using dedicated consoles for system access prevents the system's user(s) from running any tests on the system from any location other than the dedicated console(s).

There are, of course, other options for the method of accessing the system. Access can be open to an entire workgroup. This opens the system up to a larger group of researchers, depending on the size of the workgroup, and it impacts directly on the central controller design, since the central controller can no longer be designed with the assumption that certain dedicated connections will always be available. However, as with the accessing through a dedicated console, access is restricted to a certain locality and the user interface software must be made available to each computer in the workgroup that may be used to access the system.

User access can also be expanded to the entire intranet of an organization – university or business. As with full workgroup, LAN or WAN access, opening up system access to more users' consoles entails greater security risks to the system, greater ability to use the system as a demonstration tool among fellow researchers within the same organization, increased demand on system resources and greater need to work out logistical problems related to accessing the system's processing resources. Further, as user access is widened to include a larger number of locations, it becomes easier for researchers to access the system while away from their principal office/research laboratories.

To make the widest possible access, using modern technology, available for a multi-user system, the system must be accessible over the Internet. One of the advantages of making the system accessible over the Internet or over an intranet is the immediate availability of the user interface software on any console with system access, without the need to install

the user interface software before accessing the system. Sites on the Internet and intranets can be viewed using web browser software, and the user interface for multi-user systems with Internet access can be designed to be accessible by standard web browser software. This means that the user interface software does not need to be installed on a machine prior to system access, and it also means that the system can be accessed by any computer, regardless of its type or operating system, as long as that computer can display and access the user interface software.

The most common form of software applications designed to be embedded into an Internet webpage, and accessed and run directly by web browser software is called an Applet. While Applets can be written to function only on certain operating systems, they can also be written to function across all platforms. The standard computer language for writing Applets is Java, which was originally created by Sun Microsystems Inc. Though proprietary software extensions to Java, such as Microsoft's Visual J++, provide additional functional at the cost of only functioning on specific platforms, the standard Java language designed by Sun Microsystems is platform independent. The only drawback to accessing a multi-user system through an Applet, aside from the additional increase in security risks and logistical issues that accompany any increase in the number of potential access points explained above, is the issue of establishing a connection between an Applet an other software. One built-in security feature of Sun Microsystems' Java is a total lockout of all socket communication between the Applet and sockets on any computer besides the one on which the Applet's software resides. Thus, the central controller and the user interface software must be present on the same computer and that computer must have Web Server software installed and running on it for access via Applet to function. It should be noted, however, that Applets are not the only type of software that will function on the Internet, however they are the simplest to manage since the users do not need to have the software on their console in order to access the system – they need only to access the applet using standard Internet browser software. The Java language will be discussed in further detail in other sections that follow.

Aside from the measures that must be taken to meet the constraints for using sockets on Applets, Internet access to the system is simply a scaling up of the number of potential users. Using the Internet as the medium of access, of course, entails added security precautions not necessary for in-house systems that cannot be accessed by anyone outside the circle of researchers interested in the system and their colleagues. However, using the internet also frees the researcher from transporting user interface software or even computer hardware with him/her whenever he/she wants to showcase algorithms developed on the system to people outside the area of system access. This can occur on systems where access is more localized and a researcher travels to a technology showcase or to investors or other researchers intent on showcasing his/her research.

Aside from determining how localized system access should be, another important decision when designing a user interface to a multi-computer, multi-user system is deciding what privileges each user should have. This notion was touched upon above when the possibility of having a master user was discussed in relation to the logistical problem of allocating system resources. However, aside from logistical issues, the designer of a multi-user system must identify the needs of the researchers who will be the users of the system and the privileges one should grant to them. The system can be designed to treat all users equally. Or, the system can be designed with a user hierarchy that will grant certain users more control over the system than others. In certain cases, such as the case of a system with a single controlling user at a time, referred to as the master user, the system may require additional facilities, such as inter-user communication protocols to ensure that the privileges granted to one user by the system will not trample on the research work of other researchers who are lower in the hierarchy. The notion of employing a user hierarchy is discussed in Appendix I.

3.2.4 Displaying Results

As mentioned in the previous section, any computational system must provide some means for users to view results and thereby verify or supervise the operation of the algorithms running in processing blocks on the system. In the case of a computer vision

or image processing system, results tend to come in the form of charts, graphs or images. Given that the results are visual in nature, the natural inclination is to display results as images or video. While the question of whether the system itself should use sequences of still images or streaming video is left to the next subsection, it is clear that whatever choice is made about the type of data that the system will operate on will also affect the decision on the type of data used in the displaying of results.

Aside from the question of what type of data to present to the user, the other major issue surrounding the displaying of results is how should the system construct the display. The display can be transmitted to the user(s) directly from the node that is producing it, or it can be transmitted to the central controller and, from there, displayed to the user(s). The display system can display results at only one node at a time per user or multiple nodes per user, though, as mentioned in preceding sections, displaying more than results at one node per user will severely reduce the transmission bandwidth available to the system's computers. The displayed results can be transmitted directly as they are being produced, or they can be saved into files and accessed after a delay. Finally, the user interface can be designed to display the results directly or a dedicated display program can be designed to display the results at a node selected on the user interface.

For these design choices, there are quite a number of deciding factors. Unlike the other aspects of the system where the design of one piece of the puzzle determines the structure of the other parts, here there are several factors that must be considered in the display system's design. Further, the decisions for the various design choices are interdependent. For instance, the question of where the display transmission should originate depends on several aspects of the user interface design as well as the number of controlling users the system will support, the central controller design and the average expected transmission latencies between the processing blocks and the central controller. If the system is designed to have only a single master user, then the privilege to set displayed nodes could be included among the master user's privileges. In that case, the results at the node or nodes designated as displayed by the master user will need to be transmitted to all of the users on the system. If, however, either the system is not designed with a single master,

or if the system has a single master user but the privilege of setting nodes displayed is not restricted to the master user, then each user should only be shown results at the nodes he/she requested. In the first case, where all users will be shown the same results, it is possible to have the display data transmitted to the users from the central controller. The decision on whether to do this or simply transmit the display from the location where it is generated depends on the average expected transmission latencies between the processing blocks and the central controller and on the design of the central controller, which would have to be modified to accommodate the display data and would need to have enough transmission bandwidth to be able to transmit all display data to all of the users and continue to receive new display data from the processing blocks that are generating the display data as well as receive and transmit instructions.

In the second case, there is very little choice - the display data should be transmitted from computer generating that result directly to the user who requested that result displayed. Otherwise, the central controller will have to keep track of each display and the user who requested it, as well as be able to handle a throughput greater than 2NMD, where N is the maximum number of users allowed on the system, M is the number of nodes each user is allowed to display and D is the expected data rate in bytes per second for the display.

Based on this, it is easy to arrive at the conclusion that to save on transmission bandwidth, the best design choices for a system involve designing a system where only a single master user can set only a single node to be displayed. However, the decision is not so simple, since it impacts other elements of the design. For instance, if the user interface is designed to access the system through the Internet or an intranet, transmitting directly from any computer on the system to the user interface may not be possible. As noted in section 3.2.3, Applets that are run from Internet browser software cannot communicate with computers with the exception of the computer on which they reside. Thus, a system that uses an Applet as its user interface cannot access display data on just any computer on the system. And since, presumably, the central controller will run on a dedicated machine, only the central controller will be able to transmit data directly to the Applet.

There are several ways a system designer can get around this. One is to have the Applet open another *window* in the Internet browser to access the data. This solution requires that each computer on the system have software installed to make it an Internet server. It also requires that the computer generating the display compress the data in a way that it can be accessed directly from Internet browser software. Installing Internet server software on every machine in the system is usually dependant on permission from the system administrator(s) responsible for the network(s) where each machine on the system resides and may not be a possibility. As for the data compression, it is necessary since Internet browsers are not able to download and display raw, headerless, uncompressed video or image data. Once the decision to use data compression is made, a decision on what data compression format to use is required. Some compression formats allow data to be displayed as it is being compressed, while others, such as the AVI format, cannot be viewed until all data is written into a file. The decision will impact on the way the user's console will access the data, the expected delay between the time data is generated and the time it will be displayed to the user and the amount of processing that the display system will draw to compress data.

There is a further complication in the case where the system uses still images rather than streaming video. In this case, the webpage that is used to access the display data must be written to reload/update itself on each new frame of data, typically several times a second. Such a display will also appear jagged compared to one built using streaming video. The issue of whether to the system should use still images or streaming video for data interchange is discussed in section 3.2.6.

By now, I'm sure it is apparent how the various design choices vis-à-vis the display system are inter-dependent and depend on multiple factors. When designing a display system for a multi-user system, the design choices of each part within the multi-user system have a direct impact on all aspects of the display. Further, design decisions on the display system have a tremendous impact on a multi-computer multi-user system. Aside from displaying results, the type of system discussed in this chapter needs very little transmission bandwidth to transmit data between the central controller and the processing

blocks if a partly-decentralized architecture is used. As well, data throughput requirements between the system and the user interface, aside from those of the display system, are very low. Therefore, the design and constraints placed upon the display system are can be said to determine the entire nature of the system's components.

3.2.5 Distributed Computing Resources

So far I have been discussing the design of a multi-computer system without fully explaining the advantages and disadvantages of building a system that uses the combined processing power and memory capacity of multiple computer systems. Simply put, using a distributed architecture, an architecture that makes use of computing power that is distributed among several computers, allows a single user to harness more computing power. For a vision system built run on a single microcomputer, if the system could not supply enough processing power for a given researcher's needs, the only solution is to purchase a more powerful microcomputer or to run the system on a large and expensive mainframe. Were the same researcher to experience a lack of processing power on a distributed system, the solution would be far simpler and less costly – simply add one or more computers to the multi-computer system.

Designing such a system is a non-trivial task, as my work in this field and this document will attest to. However, once done, this system would be a boon to researchers in need of more processing power than a single microcomputer can provide. Presumably, a distributed system is most useful for research that requires algorithms to run under real-time constraints. Where these constraints are not present, researchers would, presumably, be able to manage with a single microcomputer, and though a distributed system would compute results more quickly, such a system, though desirable, would hardly be necessary.

It should, however, be noted that the processing power realistically available to a researcher on a distributed system is <u>not</u> NP, where N is the number of computers on the system and P is the average processor power in either MIPS or FLOPS. The distributed

systems that I have described here work best if each machine on the system has locally running software to control dataflow on the processing blocks loaded on that computer. This software will draw some of the processing power away.

Further, there is no guarantee that another user will be using a given computer in the system locally and thus reduce the processing power available to the distributed system. As well, even if measures are taken to ensure that the machines that are part of the distributed system are not used by any other, there is still a processing and bandwidth cost to be paid whenever processing block on two different computers in the system are connected. There is no way to get around this and still take advantage of the full processing power of the system. Allow me to illustrate. Let us assume the overhead from the local processor block managing software is minimal. Further, let us assume that each computer in system is dedicated to the system – i.e. not for use by other users besides those using the system. If the system's users decide not to connect any two processing blocks that are on different computers in the system, then the processing power available on the system is approximately NP. However, given that the system's users have decided not to take full advantage of the fact that the processes running on the computers on the system can be linked, then no test can that requires more processing power than P can ever be run on the system. Once a user decides to link processing blocks on two different computers in the system, the processor power available to him/her is approximately 2P-T, where T is the processing cost of transmitting a single data stream from one computer to another in the system.

The processing cost for each connection between computers in the system depends on the method in which the link is achieved. If raw data is transmitted between computers, then the processing cost is low. At worst, if a user attempts to transmit compressed data on a system that transmits only uncompressed data, the processing cost in transmitting the data is the processing requirements of a single decompression processing block. It bears reminding, though, that processing costs are not the only measure of the cost to a distributed system of a transmission between computers. There are also costs in bandwidth and memory, though the memory cost is negligible compared to the

bandwidth costs. Transmitting raw data requires more bandwidth than transmitting compressed data of the same resolution. At ten frames per second, a system with computers connected to a 1 Megabit per second channel can be very overtaxed by a single 160x120x24bits stream.

Eg. 160x120x24bits = 460,800 bits/frame, or 460,800 x 2 = 921,600 bits/frame if a computer takes this image for input and output. At ten frames per second, the system would need a 10 Megabit per second channel for each computer.

Compression would reduce bandwidth requirements. However, video compressors can be quite computationally expensive. A distributed system that uses video compression for transmissions between computers on the system might end up using half or more of its processing power on video compression and decompression. There is no optimal solution for this, short of equipping each computer on the system with video compression hardware. Using compression is computationally expensive but relatively cheaper for bandwidth. Using raw data is computationally cheap but highly expensive in terms of bandwidth. There is one solution that I have to this – a compression technique I designed, which I refer to as Foveal compression. Though it is a lossy compression, it is relatively computationally inexpensive compared to standard compression techniques, provides up to 20:1 compression (15:1 for standard 4:3 aspect ratio video), and is designed so that some algorithms can even be run on the compressed data. This compression is discussed in section 3.4.2.

Aside from the loss of processing power in the system due to overhead on data transmissions between computers in the system, there is also the loss of processing power due to transmission and generation of display data for results. Each node where data is gathered for display to users must have its data converted to the format of the display system and then transmitted to the user. If the display system transmits all data to users from the central controller, then each node whose results are being displayed must transmit data to the central controller. In this case, the central controller's computer must have sufficient bandwidth for the maximum possible number of displayed nodes. Even if the display data is not routed to the user through the central controller, it must still be

prepared for display and the computer generating the display will require processing power both for converting the data to the display system's format and for the processing block that transmits the data to the user(s).

From all of this, it is clear that the total processing power available on a distributed system is not simply NP. Users of a distributed system must be aware of this when balancing the processor block load on each computer in the system. Designers of distributed systems must take care in design to ensure that the system can run reasonably, with a reasonable resolution for the video streams.

3.2.6 Streaming Video vs. Still Images

In a single computer system, as opposed to a multi-computer system, the choice between streaming video and still images is heavily weighted in personal preference – that of the designer. Only two other factors play a part in the decision. One is the desired range of variability in available image resolutions. Typically, devices that can capture video can only produce video streams with certain specified image resolutions. If more variability is desired, use of still images is required.

The other factor that plays a part in the decision is the availability of software that can operate on one of the two types of data. For instance, in section 3.2.3, the notion of using an existing software package for defining the processing block format and controlling the processing blocks is discussed. If an existing software package is used, it may not accept both still images and streaming video data, and so the choice of which to data type to use will be decided by the software package that the system's designer chose as a framework upon which to build the system. Further, if the designer chose a package that works with both types of data, then the system can be designed in such a way to allow researcher using it to choose the type they prefer to use.

Multi-computer systems are somewhat different. Unlike single computer systems, which present very few constraints, multi-computer systems must be able to contend with

dataflow issues while passing data between processing blocks on different systems. A system that uses still images to pass data between computers will have poorer results than one that uses streaming video for all information interchanges over a network. This is due to the need, in systems that use still images, for the processing block receiving data to request new input data. While there exists no transmission latency between processing blocks running on the same computer, the transmission latency between processing locks on different computers is non-trivial. To work around this, the processing blocks designed for a multi-computer system using still images would have to be designed to constantly request new image data. Once this is done, the system will effectively function like a streaming video system, though the video capture devices may become overloaded with requests for new images.

The issue of video quality has been raised with me by other researchers. The basic complaint put forward is that the quality of video streams produced by standard video capture devices used on personal computers, such as the standard quality Logitech QuickCam, is very poor. This is often true. However, the suggestion overlooks the fact that these same devices produce still images with the same low quality. There are video input devices that can produce higher quality images than standard computer video capture devices, and those that can produce video streams, also produce better quality streaming video than that of standard computer video capture devices. Further, these devices do not experience a major decrease in image quality, between the still images they produce and the streaming video they produce. Some produce steaming video with the same or nearly the same quality as the still images they can produce. Thus, while the issue of image quality of the data used on the system is an issue of some importance in determining what type of data the system should use, it is important to note that the image quality is dependant in large measure on the quality of the video capture devices used on the system. If the quality of standard devices is insufficient, the problem will likely manifest itself regardless of which mode – still images or streaming video – the device is set to. Essentially the only difference, from the perspective of a video capture device, between producing a video stream or producing a sequence of still images is the header information and format of the data presented and, for the majority of video

capture devices, each still image must be requested while the streaming data is produced at whatever frame rate the device and the computer it is connected to can handle. Rather than use video quality as a criteria for selecting one data type over the other, in cases where improved image quality is required, researchers should invest in better quality video capture devices.

Another issue that has been raised by fellow researchers has been the low resolutions available on streaming video. Lower resolutions are typically used on streaming video due to bandwidth considerations. Raising the resolution on a stream increases the bandwidth requirements, which in turn may force the system to reduce the frame rate if the required bandwidth is greater than the maximum throughput on the transmission channel. Perhaps it is unknown to these researchers that the same issue exists when using still images. The frame rate in a streaming video system is subject to the same constraints that the frame rate in a still image system would be subject to if run on the same machines. Thus, this is not a reason for preferring one data type over another.

3.2.7 Definitions

Before proceeding further, I feel there is a need, for the sake of clarity, to define certain terms that will aid in understanding the sections and chapters that follow. The first of these are *upward* and *downward* communications. Upward communications refers to messages sent from the computers running processing blocks to the central controller, or from the central controller to the user interfaces. Downward communications refers to messages sent from the user interface to the central controller or from the central controller to the locally running processing block controllers on each computer in the system. Collectively upwards and downwards communications are also at times referred to as vertical communications in this document.

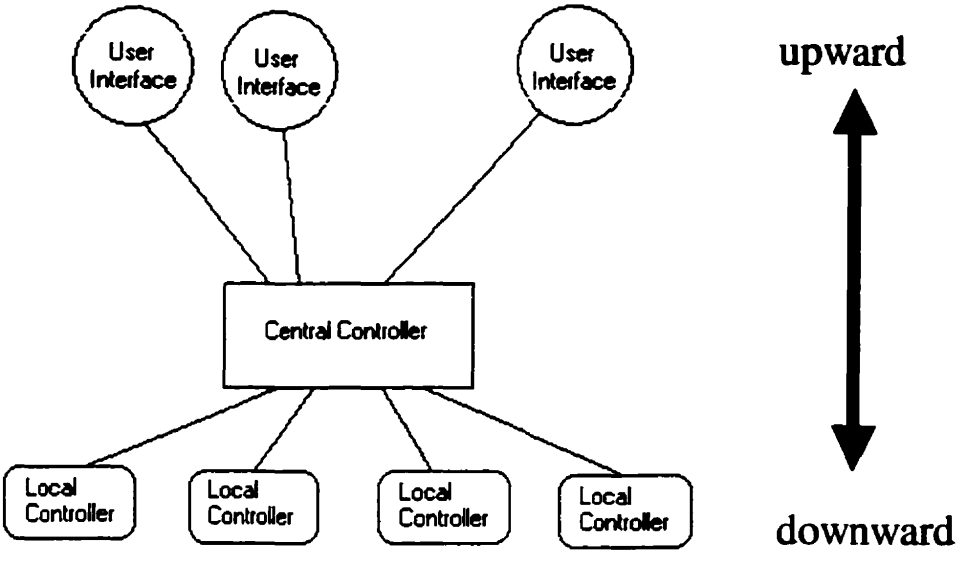

Figure 3.2-3: Upward and Downward directions in System description

The term *system* itself in this document refers to the central controller and the computers that run the processing blocks. The user interface is not considered part of the system for the purpose of this document, nor is the user end of the display system in instances where the two are separate pieces of software. Rather, the user interface is considered to be logging into the system, whereas the various computers in the system are *connected* to the central controller. In general, there is some room for leeway on whether or not to include the user interface in the definition of the system, however, I have chosen not to include it.

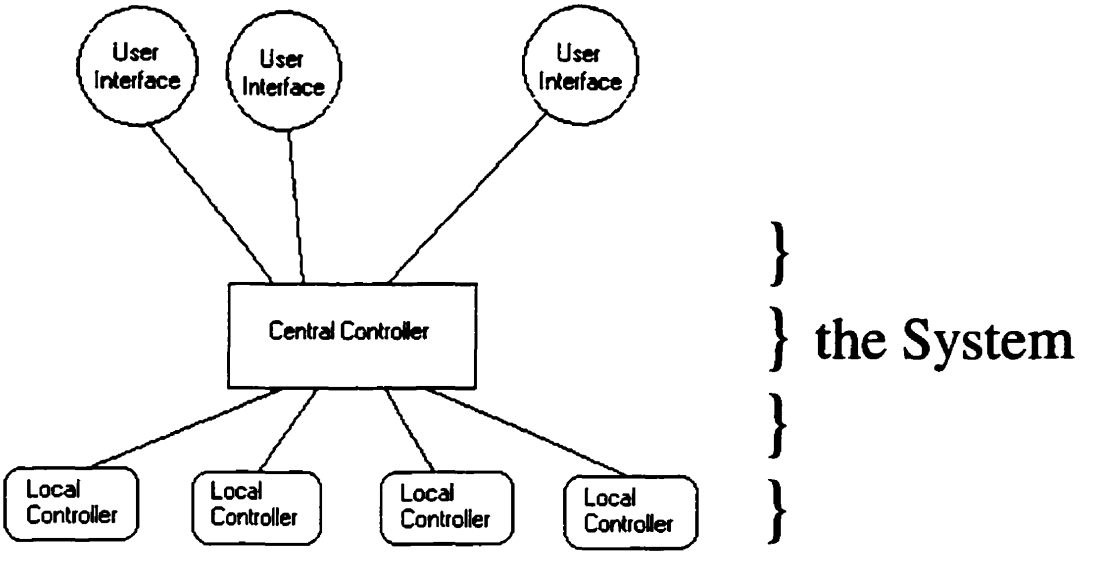

Figure 3.2-4: Diagram indicating that the system comprises the local processing block controllers (and the processing blocks running on them) and the central controler

There are two exceptions to the use of the term *system*. One comes in the term *display system*. This refers to the software that is part of the system and software that is run on the user's console, either as part of the user interface or as separate software, which is used to produce a display of results at one or more nodes in the system. In effect, the term can be considered a shortened form of the term *the system for producing and displaying results*.

The other exception to the usage of the term *system* occurs in the terms *inter-system connection* and *inter-system communications*. Connections between processing blocks on different computers are handled by processing blocks that act as intermediaries and establish a connection over computer network between themselves. Such connections are referred to in this document as *inter-system connections*. Communications over inter-system connections are referred to as inter-system communications. Technically speaking the term intra-system connections would be more accurate, however, using the standard definition of the term *system* defined in this section, any connection between any two processing blocks could be called an intra-system connection.

The other possibility would be to use the term inter-computer connection. There is a problem with this approach as well. Vertical (upwards and downwards) communications are also sent over inter-computer connections. For this reason, connections between processing blocks on different computers are referred to as inter-system connections, and the word "system" is taken as shorthand for the phrase "computer system" in this term. I admit that this nomenclature is somewhat inaccurate. Still, since inter-system communications are quite important within a distributed system, a term needed to be reserved to refer specifically to this type of communications.

At times, the term intra-system communications is used. Wherever it appears, it refers to connections between two processing blocks loaded on the same computer.

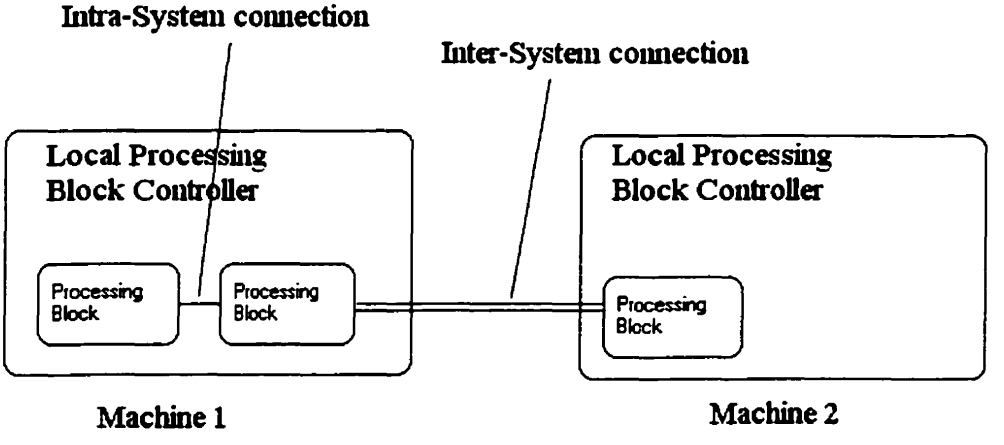

Figure 3.2-5: Illustration of the difference between inter-system and intra-system connections

Intra-system connections refer to connections between two processing block loaded on the same computer. These computers are running software locally that provides an environment upon which the processing blocks can be loaded. Since it runs locally, this software is, at times, referred to as *local* controller software, to distinguish it from the central controller software. Local controller software is responsible to handle dataflow issues between all processing blocks loaded on the local computer.

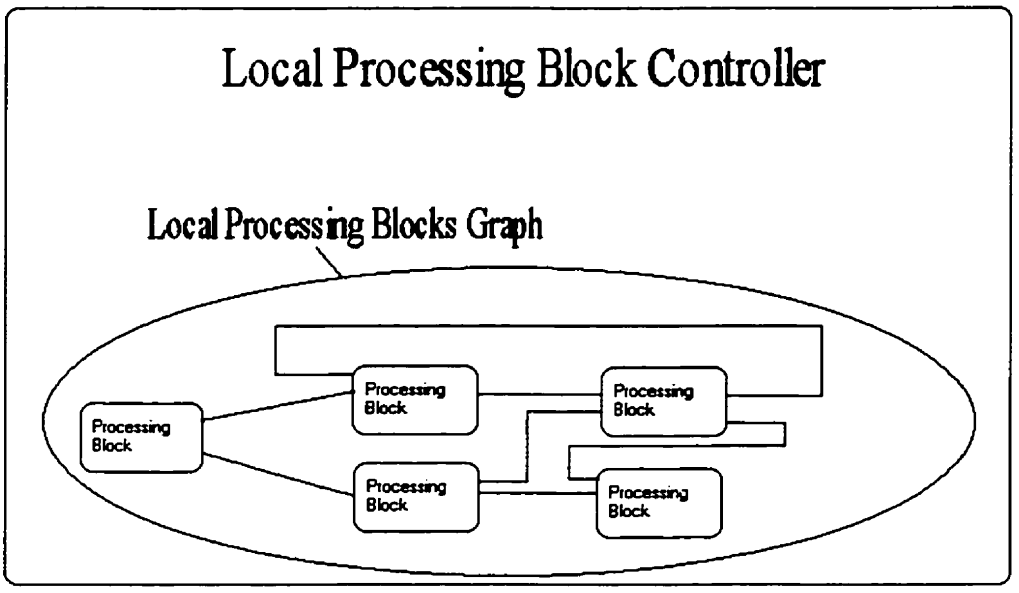

Figure 3.2-6: Illustration of Local Processing Blocks Controller, depictng local Processing Blocks Graph

The collection of processing blocks loaded on a computer's local controlling software are referred to by several names. They are referred to as the *network of processing blocks*, *processing blocks graph* and *system graph*. Here, the word system in system graph refers to the entire system as defined in this section.

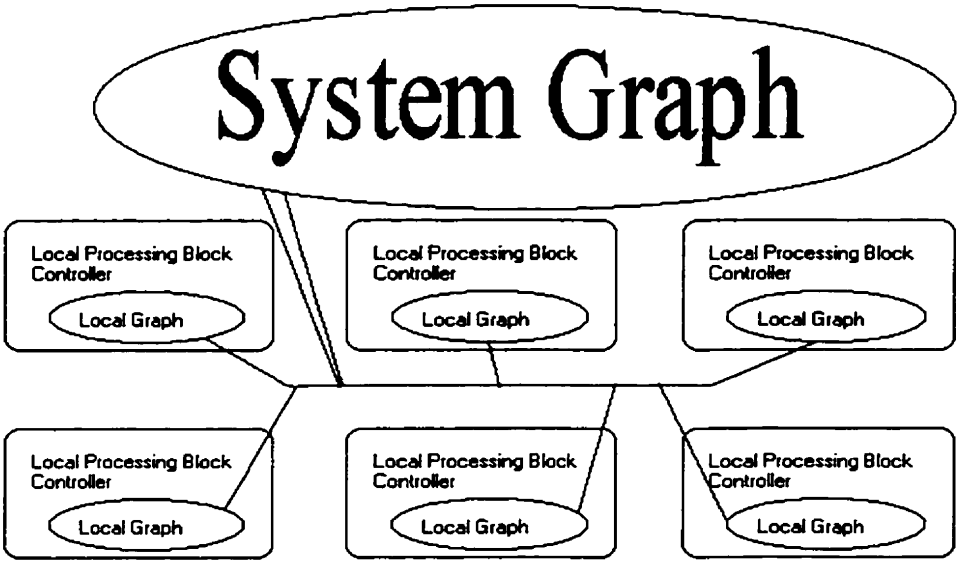

Figure 3.2-7: Depiction of System Graph. The System Graph is the union of all the local processing block graphs

In the discussion concerning the user interface, the term *master* user was presented. The definition of this term is the user who is the only one who has control of the system's resources. In a system where there is a master user, only the master user can load/unload processing blocks from/to the system graph and only the master user can make or break connections between processing blocks or issue commands that would change the operating mode of the system graph.

The terms *platform independent* and *platform specific* should also be defined. For the purpose of this document, the "platform" referred to by these terms is the combination of the type of microcomputer and the operating system running on it. For instance, a Windows NT system is one platform – a microcomputer designed with a general-purpose microprocessor built by either Intel Corp. or American Micro Devices (AMD) and running the Windows NT operating system designed by Microsoft Corp. Software that is *platform independent* can be run from any microcomputer, while *platform specific* software can only be run on one or more platforms that meet a certain criteria.

Two more terms that require definitions are *real time* and *complex*. Both of these terms are used extensively in technical publications, and several definitions of each term exist. In this publication, I often use the adjective complex in a differential mode, to indicate that one item is more grand, complicated or more difficult to build, manage and maintain than another. I also use the term complex, as in *complex commands*, to differentiate these from simple – single line commands.

As for the term *real time*, I use this term applied to a system. A system operates under real time constraints if that system can complete its computations on input data before the next input sample is ready to be processed without forcing data to be dropped. This measure is irrelevant in systems where computational blocks poll input devices for data when they are ready to process new data, unless the input devices grab data at a constant frame rate and drop data if no request for data is received.

Finally, I feel the need to define the term *area of system access*. This term refers to the collections of locations from which a user can access the system using the user interface software. A system that has only a single, dedicated console for user access has a very narrow area of system access, which encompasses only that one console. A system that provides user access over the Internet has a very wide area of system access, encompassing all computers with suitable Internet connections.

In the sections and chapters that follow, more new terms will need to be defined, and some terms that will be used interchangeably with those defined here will need to be presented. When the need to define new terms in this document arises, those new terms will be presented and defined.

3.3 Design Goals and System Requirements

In the previous section, I explained the components required by a distributed system such as the one I am proposing. In so doing, I touched upon the various design considerations that go into the design of each component. It should be clear by now that, while some design choices are easier to implement or provide more functionality to the user than others, ultimately, the design goals set for a distributed system carry the most weight in its design. The advantages and disadvantages associated with each design choice may serve to guide the design towards attainable goals and may serve as a deciding factor between two or more choices that fit the design goals. Once the design goals are set, however, they determine the basic structure of the system, its components and its user interface.

My initial design goal was to build a distributed system that would use streaming video for dataflow, and allow user access over the Internet. As explained above, choosing streaming video over still images is very much based on user preference. I had experience with using streaming video in the past. Further, the streaming video software package

that I had the most experience with, DirectShow, fit quite well with the system's needs for definition of the processing blocks and local processing block control. I will detail the relevant aspects of DirectShow in section 3.4.

As to the goal of building an Internet-accessible system, I saw several advantages. First, Internet accessibility would mean that I would not need to copy my entire user interface onto each computer that I wanted to use as a console. Second and far more important, I would be able to access the system from anywhere. The system would, effectively, be its own demonstration tool.

From my initial goals, and my study of distributed systems, I was able to arrive at certain basic requirements for the interface and the components in the system.

3.3.1 User Interface Requirements

To provide Internet access, the user interface would be a Java Applet. To allow the Applet to communicate with the system, the Applet would use sockets. Due to the restrictions placed on socket communication for applets, the Applet would reside on the same computer as the central controller.

The user interface would provide the user with a way to connect to the system and would provide a graphical view of the processing blocks graph. Since Internet access opens the system to certain security risks, the system would be password protected, and so the Applet would need to provide a user login interface with a password to prevent unlicensed users from accessing the system. Other security measures for the user interface were considered, however, since the focus of this system was not on security, those issues were dropped and left for later implementation. A discussion of those issues appears in Appendix I, section I.5.

Since the Applet would be the user's only entry point allowing control of the system, certain basic pieces of information would be required for it. For instance, aside from

providing a graphical representation of the processing blocks graph, the Applet would have to display an itemized list of the processing blocks available on a given system. It would need to provide the user with a means of loading and unloading processing blocks and well as connecting and disconnecting them. The Applet would need to warn the user in the event that a major change occurred to the system, such as the addition or removal of a computer from the distributed system. As well, the user would need to have control of load balancing and so the Applet would need to be designed to allow the user to access the individual computers on the system independently. This would allow the user to determine where within the system each processing block should be loaded.

3.3.2 System Control Requirements

Given my experience with DirectShow and my decision to rely on it to provide local control of processing blocks, the distributed system that I sought to design would have a partly decentralized control structure. The central controller would be responsible to act as the central hub of the system, relaying upwards and downwards communications while keeping track of the system's processing blocks graph. Given the multi-user nature of the system, the controller would need the ability to route messages from one connection to another or from one connection to many others, so that all of user interfaces connected to the system would be updated when one user made a change to the system. The central controller would also provide certain essential services to the users, such as issuing sets of commands needed to make inter-system connections.

3.3.3 Processing Block Requirements

The processing blocks are the lowest level within the distributed system. In a sense, they are the foundation of the system. The properties and abilities of the processing block format and the local managing software delineate the boundaries of the user's control of the system.

For the system to function and perform in the manner I required, it would need to be able to load and unload the processing blocks from the processing blocks graph on each local

controller. It would also need the ability to initiate connections between locally loaded processing blocks, handling all handshaking during connections, and break established connections. In order to aid the user in selecting processing blocks to load, the local controlling software would need the ability to locate the processing blocks stored on the local computer and report back to the central controller on the name and type of each processing block available to it. Further, the processing blocks themselves would need the ability to report the number and type of connections that were possible to the local controlling software. As well, the local control software would need to keep track of the processing blocks locally loaded. Since I selected streaming video over still images, the processing blocks format would need to be designed so that the processing blocks would operate on streaming video, either frame by frame or several frames at a time. Finally, since inter-system connections are a requirement for the system, the format for writing processing blocks must be open enough to allow the processing blocks to make socket connections and to allow the local controlling software to set variable in the processing blocks or, at the very least, to pass messages to the blocks.

As the next section will show, Microsoft corp.'s DirectShow met the requirements I had set for the processing blocks.

3.4 DirectShow

3.4.1 A Quick Introduction to DirectShow and COM

DirectShow is one of a group of software packages designed by Microsoft corp. called DirectX drivers. It functions as the streaming multimedia driver for DirectX. Under DirectShow, processing blocks are called *Filters*. Each Filter object is compiled as a dynamically linkable piece of compiled code that may be linked to a process running on a computer and configured. DirectShow Filters comply with the COM standard.

COM is the component object model, a binary standard for information interchange. It is a binary standard that defines how objects are created and destroyed and, most importantly, how they interact with each other. As long as applications follow the COM

standard, different applications from different sources can communicate with each other across process boundaries. Since COM is a binary standard, it is language independent. Under DirectShow, Filters and *Filter Graph Managers* – applications designed to manage Filters – can be designed using Visual C++ and Visual Basic [9].

All processing blocks written under DirectShow, regardless of function, are called Filters, yet they are not all true filters in the engineering sense. Filters can belong to one of many categories and come in three types – Source Filters, Transform Filters and Renderer Filters. Source Filters are a local source of streaming data. These include Filters that access multimedia files stored on the local computers data storage drives or on remote computers, Audio capture Filters, designed to capture audio input from a computer's microphone and Video capture Filters, designed to capture video from local or remove video capture devices. These Filters are the starting points in a COM applications' *Filter Graph*, the DirectShow term for processing blocks graph.

Renderer Filters include all Filters that render streaming data to output devices such as the local computer screen, speakers or a socket. These Filters are end points in a COM applications' Filter Graph. Transform Filters include all Filters that run any form of operation on a multimedia stream. While Source and Renderer Filters are not true filters in the engineering sense, most Transform Filters are. Researcher's algorithms on the distributed system would be designed as Transform Filters. Figure 3.4-1 shows a sample Filter Graph.

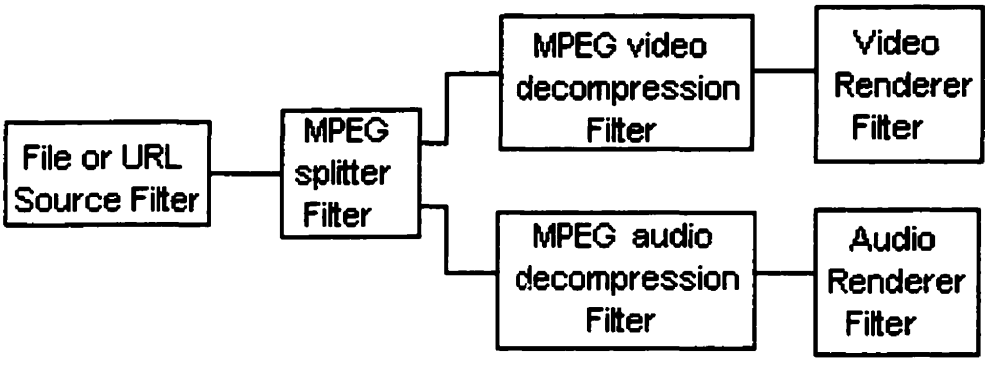

Figure 3.4-1: Block diagram of a Filter Graph, depicting decompression and rendering of an MPEG-compressed stream of data.

Filters have data entry and exit points called *pins*. Under DirectShow's specifications, Filter pins are unidirectional. They are used to pass streaming multimedia data, audio or video, compressed or uncompressed, from one Filter to another. Source Filters, typically, have one or more output pins and no input pins. Renderer Filters typically have a single input pin and no output pins. Transform Filters can have any number of input and output pins.

Figure 3.4-2: Block diagram of a simple Filter Graph complete with Filter pins

When a connection between one Filter's input pin and another Filter's output pin is initiated in a Filter Graph manager, DirectShow handles the handshaking and attempts to configure the *downstream* Filter – the Filter further from the Source Filter(s) - and *upstream* Filter so that the two are using the same data type to transfer streaming data.

In addition to pins, Filters have another means of access, which is necessary to allow a Filter Graph manager program to control them. Filter Graph Manager software accesses Filters using COM interfaces. These interfaces, provided by DirectShow, allow the Filter Graph Manager to retrieve information from the Filters and their pins, handle dataflow between Filters and configure the Filter Graph. One group of these interfaces consists of enumerators, which allow a program to scan the local computer's registry for register Filters.

Under Microsoft's suite of Windows operating systems, each computer contains a software registry that stores information such as the location of certain program on that computer's storage units. DirectShow Filters and drivers for video and audio input and output devices must be *registered* in the registry in order to be accessible. This is very significant in that it fulfills one of the requirements of the distributed system – that a list of the processing blocks available on a given computer in the system be accessible by the system.

Unknown to most personal computer users, DirectShow Filters are found on all Windows computers. These include decompression software for standard multimedia compression types, and Filters designed to access the compute screen. As well, DirectShow is able to access standard drivers for video and audio input and output devices connected to the local computer and make them appear to be standard Filters to Filter Graph Manager software, complete with input or output pins. Thus, when using DirectShow to build an application, a developer has access to an existing bank of Filters, including full access to peripheral devices.

For a more detailed look at DirectShow, consult Microsoft's website.

3.4.2 Previous DirectShow Experience: Foveal Compression

DirectShow is not new to me. My previous work in streaming video involved designing a type of video compression for streaming video. The compression scheme, Foveal compression, was designed somewhat along the functioning of the human eye. In short, it took an image and ran a pixel-averaging scheme on it. The pixels close to the centre of the image were left intact, while the red, green and blue values of those further out were average in square regions consisting of 2x2, 4x4 or 8x8 pixels depending on how far from the centre they were.

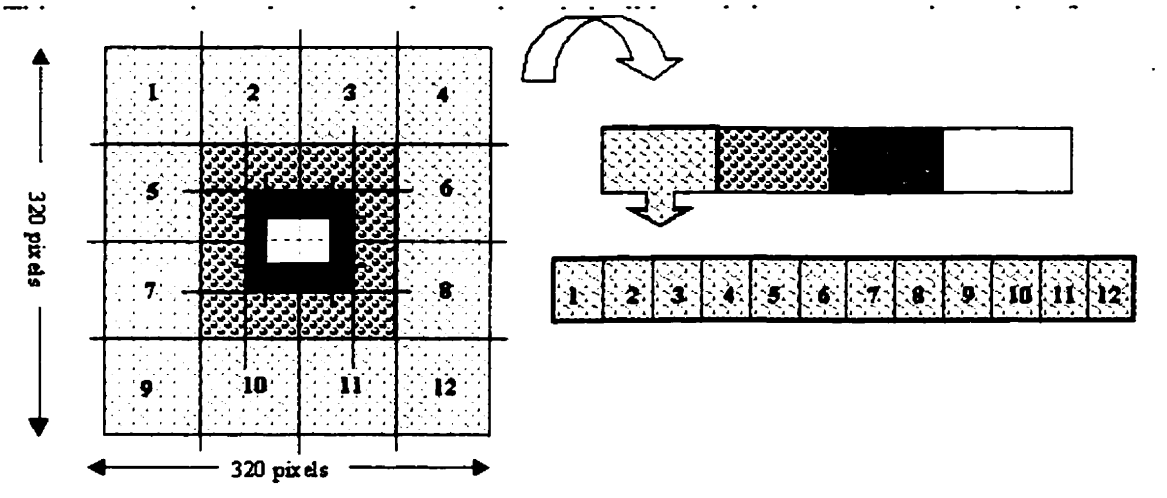

Figure 3.4-3: Foveal compression. On the left, the subdivision of a 320x320 pixels image into 3 annular regions with 64:1, 16:1 and 4:1 compression, with 1:1 compression in the centre. On the right a depiction of the compressed data buffer. The resulting compressed output is 15,606 bytes, compared to the 307,200 bytes in the uncompressed image.

My purpose in designing the distributed system was not tied to the work I had done on the Foveal compression and the Filters I had written to operate on Foveal-compressed streams. Rather, I present this work for two reasons. First, my work with the Foveal compression gave me a good understanding of DirectShow and all its capabilities. This experience led me to conclude that DirectShow could provide all of the functionality needed for the processing blocks in the distributed system.

Also, the Foveal compression could be useful in the distributed system for inter-system connections. I would not impose a lossy compression, such as the Foveal compression, as the standard compression to be used for inter-system connections in the distributed system. No doubt, many applications would suffer tremendously from the decreased image quality. However, the Foveal compression is a very computationally inexpensive compression, and the fact that operations such as edge detection can be run on the compressed stream is an added bonus. In certain circumstances, this compression scheme could be useful, since it considerably lowers transmission bandwidth requirements.

3.5 High-level Design

3.5.1 Overall System Design Objectives

Now that the purpose of this system has been defined, its high-level design choices discussed and explained and its basic requirements delineated, it is possible to begin the task of explaining system's design. The overall design of the system, consisting of two components, as well as the user interface and the display system are explained in the subsections that follow. Each component described follows the requirements set for it. Aside from requirements, there were certain objectives I set and certain ideas I explored in an attempt to optimize the system's operation as much as possible. Some of these were implemented during the course of my work, while others were left to future work. Chapter 6 in this document is a discussion of ideas for improvements to the system I designed that I have left to future implementation.

One of my objectives in design was to reduce the transmission bandwidth requirements as much as possible, especially for upwards/downwards communication. Though the bulk of the communications in this system was inter-system communications, it is the speed of vertical communications that defines the time lag between the moment a user issues a command and the moment all the connected user interfaces have been informed of its execution. Since all users must be updated on system changes regardless of which user issued the commands to make those, each user command sent downwards must generate N replies, where N is the number of connected users. If many users are connected, this can tax the system's central controller greatly, especially if each upward message generates a downward message requesting additional information. By designing the system's components to transmit only information that is necessary, the central controller and the local processing block controllers would be able to update all users as quickly as possible. Also, by designing the user interface to retain certain information that is unlikely to change, the amount of data transmitted vertically through the system is reduced.

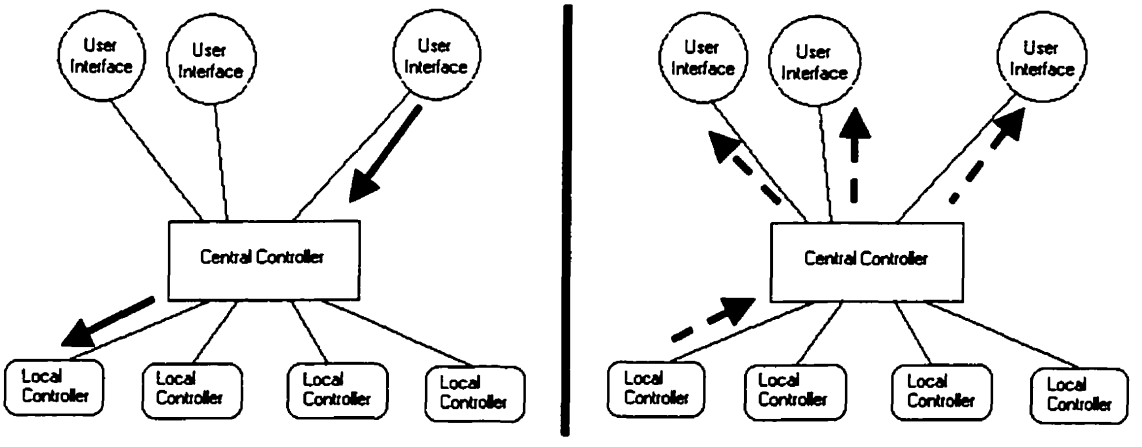

Figure 3.5-1: Datapath for a downward command and the upward system updates it generates

Another objective in design was to maximize the re-use of software code as much as possible. The central controller software, in a sense, acts like a telephone operator, routing messages from the user interfaces down to the appropriate local processing block controllers and broadcasting updates from the local processing block controllers to the connected user interfaces. For this reason, the form of upward and downward messages between the central controller and the local processing block controllers could easily be re-used for upward and downward messages between the central controller and the user

interfaces. Since the central controller must maintain a record of all system configuration data in order to perform such services as establishing inter-system connections, it must contain software code to store any system update information it receives from the local processing block controllers. This information is also required by the user interfaces in order to present a graphical representation of the system to the users. Thus, re-use of data types used to store the information and software code used to retrieve it from the upward messages is possible and quite time saving. Care must be taken in re-using the code, since only the design of the central controller can assume that it will be operating from the moment the system is turned on, whereas the user interface is able to log in after the system has already been started. Of course, unhindered re-use of code, including data types necessitates the use of the same programming language. The user interface's programming language was selected by the necessity of having an Internet-based interface. The central controller's language was selected for the ease of code re-use.

Continuing along the lines of my second objective, I found it was necessary to design the format of upwards and downwards messages to allow easy re-use of code. As well, I decided to make the vertical communications format as simple as possible to reduce the amount of processing required to extract relevant data. Therefore, I designed the format for vertical communications to look like STMP or Telnet protocols. All messages are sent as an uncompressed stream of ASCII characters ending with a *CR* character and each message that can be sent upwards or downwards in the system has an associated command number. The details of the system's vertical communications are discussed in chapter 4 and proposed future enhancements to these communications can be found in Appendix I.

My third design objective was rooted in the design of the user interface. The interface would serve as the entry point into the distributed system for the researchers that would use the system. I wanted a design that was a simple as possible, while still offering researcher all the information and control they needed for the system to be useful to them.

In my previous experience with DirectShow, I made good use of GraphEdit, a Filter Graph Manager program that is included as a utility with Microsoft's DirectShow. GraphEdit is a development program for testing DirectShow Filters and Filter Graphs. It presents a graphical representation of a Filter Graph to the user, allowing the user to load and unload registered Filters, connect and disconnect Filter pins and handle dataflow issues. When I first began using GraphEdit, I was struck by how simple the program was to learn and use. In designing the system, I decided to take some ideas from GraphEdit's graphical representation and control features when trying to make the user interface as easy to master as possible. Since the distributed system I was designing would use DirectShow Filters as processing blocks, I felt that it was likely that the users of the system would first test their Filters on a local computer using GraphEdit to insure that they functioned correctly as DirectShow Filters. In this case, the more like GraphEdit my system's user interface would be, the easier it would be for researchers to move between the two. Though some GraphEdit features, such as providing access to Filter property pages and allowing users to save Filter Graphs, were not included in my system, most of the other functionality of GraphEdit was incorporated into the final product. To a great extent, the system's user interface resembles GraphEdit, though unlike GraphEdit, it is able to manipulate Filter Graphs on many computers at once remotely. Some of these features are discussed in Appendix I. Figure 3.5-2 shows a Filter Graph on GraphEdit.

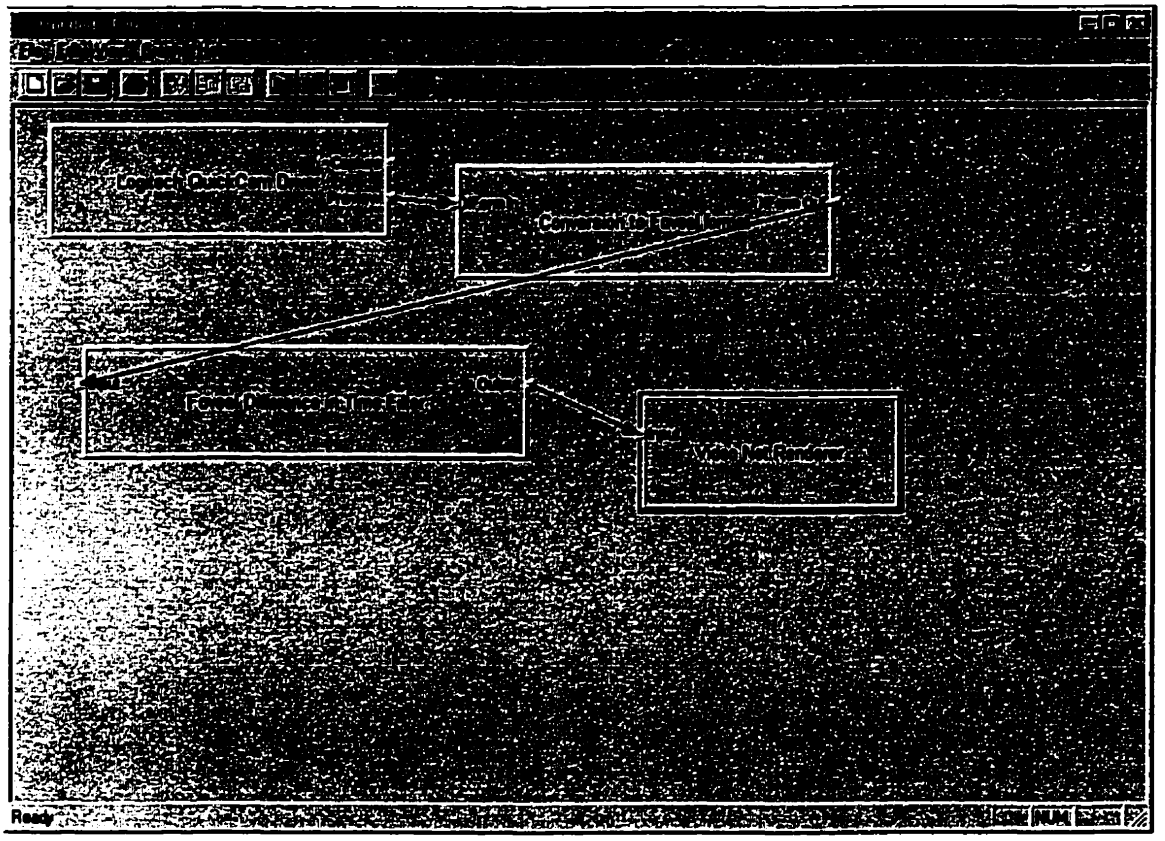

Figure 3.5-2: Sample GraphEdit application

The subsections that follow do not discuss fully the implementation of the components. That discussion is left to chapter 4. Rather, the following subsection discusses the design of each component from the point of view of high-level design.

3.5.2 Central Controller Design

In my design, the Central Controller is the central hub of the system. It was written to function as a server, and all other software, either inside the system or accessing it, were designed to connect to it via TCP/IP socket connections. I designed the Central Controller to fulfill five purposes:

1. Route upwards and downwards messages to their intended destinations.
2. Provide a central location so that each connected user interface could access the entire system through a single socket connection.

3. Retain complete information on the entire system.
4. Provide essential services, such as cross-computer services, that could not be provided by a local controller, due to the local controller's inability to access other computers on the system.
5. Allow only a single user to log on to the system as master and allow only a single local processing block controller per computer in the system.

Several models exist for servers. I chose to design the central controller as a single thread per client server. One thread listens for connections, and when a connection is attempted, it spawns a new thread to handle that connection. Since the listening socket has no way of discriminating between attempts to connect made by local processing block controllers and those made by a user interface or user's display, the code designed to handle connections is able to handle all expected types of connections. Once the connection is made, the central controller uses the login message sent to it by the software attempting the connection to distinguish between the various types of connections.

Since the central controller was designed using a multi-threaded approach, and since it was designed to be able to relay messages upwards or downwards, the central controller was designed to use Inter-Client Message Handler, or ICMH, built with internal pipes to allow its threads to communicate with each other. The controller was designed to store a list of all connections, including the name or IP address of the connecting computer and the connection type. This list could be accessed whenever a thread needed to broadcast a message to all connections of a given type or when a message was intended to a specific client accessed by type and index number. In the event of a disconnection, the central controller was designed to send message to the threads to inform them of changes in their index, as the connection at the end of the list took over the index of the disconnecting client. In the event a processing block controlled disconnected, all user interfaces would be informed to allow them to adjust their information about the system.

The Central Controller was designed to retain all data about the system. In addition to the connection list, I designed it to maintain an indexed list of the registered Filters on each

computer in the system, itemized by category, and a list containing all of Filters loaded on the Filter Graphs of each computer on the system, including their local connections. The Central Controller also retains a central listing of Filters.

Since the Central Controller was designed to route messages between the connected user interfaces and the local processing block controllers, it required a means for converting messages sent by one of these to the other. Though, as section 4.5 will explain, a very similar convention was used for all upwards and downwards communications regardless of source and destination, some message format conversion was still required.

Not all messages received by the central controller were meant to be merely re-routed after relevant data was stored. Some system services, such as setting up the display at a node in the system's Filter Graph or setting up an inter-system connection, would be difficult to accomplish from a decentralized point in the system. I designed the Central Controller to provide these system services.

The Central Controller also acts as an arbiter, allowing only a single user to be the system's master. Since I did not design a hierarchy of users into the system, the Central Controller was designed to grant master status to the first user to log on and to maintain that status until that user logged off. When the user logged off, another connected user, assuming another was available, would be selected to be the new master. Users connected in *View only* mode, as opposed to *Master* mode, are restricted in their access to certain commands on the system. I designed the Central Controller to prevent *View only* mode users from issuing any commands that could modify the system's Filter Graph, such as loading/unloading Filters, connecting/disconnecting Filter pins or starting/stopping the dataflow of the local Filter Graphs.

The Central controller was also designed to arbitrate within the system. Running multiple local processing block controllers on the same computer in the system is redundant, and so the Central Controller was designed to prevent a local processing block controller

from connecting to and joining the system from a computer that was already part of the system.

Since I wanted to maximize code re-use, and since more code was needed to process upward signals, which contain Filter Graph and registry information, than downward signals, which consist of single line commands, I chose to design the Central Controller using the same software language as the user interface. Since one of my goals for the system was a user interface accessible by Internet web browser, the user interface was designed as a Java Applet. For this reason, I chose to design the Central Controller in Java as well. The *JavaServer*, as it has come to be known, and the user interface Applet, have very similar code for handling updates. They share the same data structure that keeps track of every Filter in the system, indexed by its local computer and its index within the local processing block controller's Filter Graph. They also share data structures for listing every registered Filter on each computer in the distributed system, indexed by its type and name. Unlike the *Applet*, the JavaServer was designed under the assumption that any changes to the system's Filter Graph occurred after it went online, since, without a central controller, there is no system. Thus, only the JavaServer was in a position to know such things as which Filters in the system were merely added to produce the display at a given node or to establish an inter-system connection. The Applet was designed so that these Filters would not be displayed to the user when viewing the entire system and though they would appear on the Applet's screen when the user was viewing only a single computer's Filter Graph, the Applet was designed to prevent the user from modifying these Filters directly. These so-called *undisplayed* Filters will be discussed further in section 3.5.4 and in chapter 4.

3.5.3 Local Processing Block Controllers

As previously stated, I chose to use Microsoft's DirectShow drivers for the local processing blocks and so I designed the local processing block controllers as Filter Graph Manager software. The software was designed to establish a connection to the Central Controller using a TCP/IP socket. After connecting up to the system, the local processing

block controller software would wait for instructions from the Central Controller. The Central Controller would inform the connected user interfaces of the presence of the new computer on the system and then request data from the local processing block controller on the local Filter Graph and registry.

I designed the local processing block controller software to accept simple instructions from the Central Controller and reply with an update in the case of success or an error message in the case of failure. The local controllers were designed to handle commands requesting information on the local registry, the configuration of the local Filter Graph and its current dataflow mode – running, waiting or stopped. They were also designed to receive commands either to load or unload a Filter on the local Filter Graph, connect or disconnect two Filter pins or to modify the local Filter Graph's current dataflow mode. I refer to these instructions as *Filter Graph modifying instructions*, since their purpose is to request modifications be made to the Filter Graph's configuration or its dataflow. The Central Controller allows only one user, the master user, to issue Filter Graph modifying instructions. I also designed the local controllers to accept request for configuration information needed to set up the Filters that transmit to and receive data from processing blocks outside the local Filter Graph.

The local controllers store all relevant data in data structures designed for that purpose. A list of registered Filters is built once and unless there is a request to rebuild the list, the local controller merely retains the list and transmits it, after some formatting, to the Central Controller upon request. The local controllers were also designed to retain a full list of the locally loaded Filters, including the local connections between Filters.

In keeping with my goal of reducing the data size of vertical transmissions, I initially decided that the local controllers would reply with a short message indicating success upon successful execution of a Filter Graph modifying instruction. As I began to design and implement the user interface, however, I found that this would not be feasible. I opted instead to have the local controllers issue updates on the Filters and their connections. To reduce upwards communications, I decided that the updates would

contain information only on the Filters that changed from the most recent instruction, unless the Central Controller explicitly requested full information on the configuration of the local Filter Graph.

3.5.4 User Interface Applet

As mentioned above, one of my objectives in designing the Java Applet that would serve as the user interface to the distributed system was to design its front end to be simple yet powerful. I wanted the user to control the interface using simple point-and-click and point-and-drag mouse commands, and reduce the user's reliance on the keyboard to a bare minimum.

My expectation was that researchers using the distributed system would debug their first Filter(s) using Microsoft's GraphEdit to ensure that they had properly assimilated the knowledge on how to build proper Filters. For this reason, I used tried to give the Applet the look and feel of GraphEdit.

Under my design, the Applet code was to reside on the same computer as the JavaServer, which served as the Central Controller for the system. This way, the Applet would be able to connect to the JavaServer through a TCP/IP socket, since one of the security features of Java prevents Java Applets from making socket connections to any computer aside from the one on which they reside.

I decided to give the user two viewing options when viewing the system. The first view, known as the System Filter Graph view, was designed to present the user with a view of the entire system as a single Filter Graph. In this view, Filters used to make inter-system connections and those used to produce the display would not be displayed. I refer to those Filters as *undisplayed* Filters.

The second view would allow the user to see the entire Filter Graph on a single computer in the system. This view would include undisplayed Filters, but they would be drawn

with greyed-out text and the user would not be allowed to issue Filter Graph modifying instructions directly affecting these Filters.

I decided that, regardless of the selected view, the user would need to select one computer on the system at any given time to be the system *focus*, the computer that all Filter loading instructions would be directed at. The user would be able to switch the system focus at any time. Both views would present the user with a list of Filters registered on the system focus.

Further, to prevent delays while waiting for updates after issuing commands, the Applet would be designed to operate two threads. The main thread was responsible for drawing the graphics on the screen and sending the user's instructions to the JavaServer. The secondary thread was responsible for receiving updates from the JavaServer and issuing some information requests if necessary.

3.5.5 User Display System

The user display, and indeed, the entire display system, presented a difficult challenge to design. I decided to allow only one node to be displayed to all users at any given time to save on precious transmission bandwidth while still allowing all users to see the display and thereby get some use out of the system even when not connected as the system's master. However, my other decisions were not so easy to make. My original preference was to use an Internet browser to view the displayed stream. I designed the user interface to be a web-accessible Applet, so it was safe to assume that the users on the system would have access to Internet browser software. Typically, web browser software, regardless of platform, can either display streaming multimedia data using browser plug-in software or can launch proprietary software to display streams. Thus, it seemed logical to access the display using a webpage launched by the Applet. Unfortunately, this was not to be.

Using a web browser for the display presented several question to be resolved. First, there was the question of which computer would provide access to the display stream. Would the display be transmitted to the users' web browser windows directly from the computer that was generating it or would that computer transmit it to the JavaServer for rebroadcast to the users ? There was also the question of the multimedia format that would be used for compression of the display, since browsers cannot display raw video signals.

Giving each computer in the system the ability to set up and transmit the display required each computer on the system to run web server software. Even if I were inclined to do so, the administrators in charge of the network where the system computer reside strongly requested that this not be done. This meant that the computer running the JavaServer would also use its precious transmission bandwidth to receive the display stream, compressed preferably, and then broadcast it to all the users' browser, or rather, make the stream available on the web to any browser that connected to it. This could cripple the system, by reducing the bandwidth available for upwards/downwards communications. This could also increase the transmission latency in the display system. The increase could be significant if the user and the computer generating the display were close-by while the JavaServer was geographically far away. In this case, rather than making the low latency transmission from the local Filter Graph to the user's console, using the JavaServer would force the display to be routed through two high latency channels.

The question of video compression format was also a major issue. DirectShow Filters cannot be used to publish a stream to the Internet. Further, DirectShow comes with a File Writer Filter that can only write AVI compressed files. The trouble with the AVI format is that its header contains information on the full file's size and thus an AVI file cannot be accessed until it is completely written, unlike certain other formats that can be accessed frame by frame as they are being compressed. Thus, using the AVI format would require the display system to save small portions of the stream into small files that could be accessed. The larger the stored AVI files, the longer the delay between the onset of the results being display and the moment the user would see them. Shorter files would cut down on this time lag, but the local display system would need to store many of them

to insure that even users accessing the system from a very remote site, with a high latency transmission channel would view the files in the sequence in which they were recorded. The system would need to use the files like a circular buffer, overwriting older files, to make sure that the generating computers did not run out of storage space. Increasing the number of files would increase the frequency of discontinuities in the display transmission, since I expected that there would be some discontinuity or pause between the time the display hit the end of one file and the time it began to show the start of the next one. Using multiple files would also cause a micromanagement issue in the web browser – the web browser would have to request the next file in the display before the currently displayed one was completed.

Using other compression formats presented their own challenges, primarily because DirectShow did not include software to write files in newer formats and because, unlike decompressor Filters, compressor Filters are not commonly available on computers. I could select one of the video compressor Filters I had access to and use it for the display, but these compressors used either non-standard MPEG or proprietary Microsoft formats, and so the goal of universal system access could not be accomplished by them and the system would still be using up too much of the central Controller's transmission bandwidth on the display.

Instead, I decided to build a separate program to view the displayed stream. The program would be proprietary, using DirectShow Filters to receive the display stream and render it to the screen. The User's Display program would connect to the JavaServer briefly to query the server on the location – IP address and port number – from where the display stream was being transmitted. It would then connect to the stream and render it. This was far from an optimal solution, in that it would only work on Windows computers. However, it would alleviate the drain on the Central Controller's transmission bandwidth and reduce the latency in the display stream and would require installation and possible registering of Filters. At a later time, a Linux-based equivalent could be designed to display the video stream on computers running the Linux operating system. Alternatively, the solution proposed in Appendix I, section I.3 could be implemented.

Chapter 4. Implementation of the Distributed System

This chapter describes the implementation of the distributed vision system described in the previous section, including relevant design choices made during the implementation stage of the system. Wherever possible, I have tried not to repeat explanations from the previous chapter. Since the previous chapter touched on the high-level design of the system in great detail, this chapter attempts to only address issues that arose during implementation of the system' components.

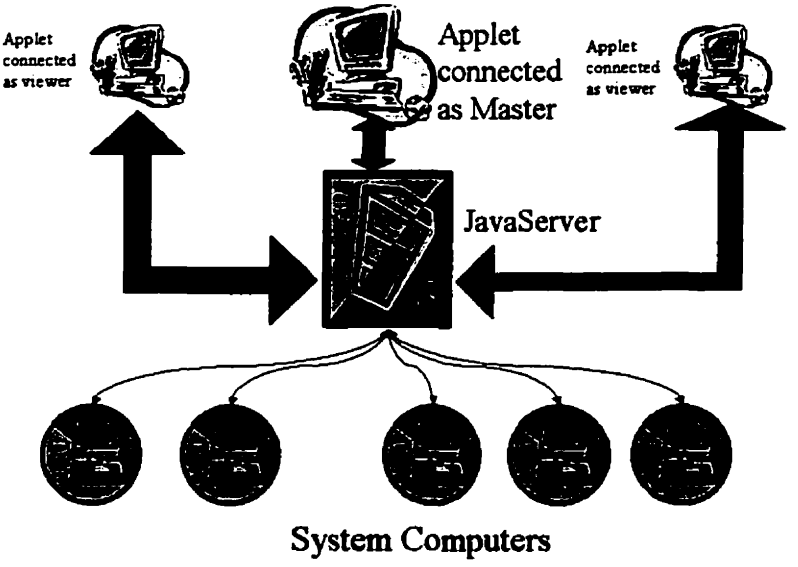

Figure 4.1-1: View of the System

4.1 C++ Clients Implementation

4.1.1 An Application to Manage Local DirectShow Filters

Given that I had chosen to use DirectShow for the processing block format design on the system, local processing block controller software would need to be written in a computer language that could manipulate DirectShow Filters and their COM interfaces. Though Visual J++, Microsoft corp.'s extension to Sun Microsystems' Java, has the ability to access some COM objects, only Microsoft's Visual Basic and Visual C++ can be used to create Filter Graph Manager software. I therefore decided to use Visual C++ to write the local processing block controller software. Since this software was designed to connect to

the central controller, as the client in client-server communication, I began referring to the local controlling software as *C++ clients*.

4.1.2 List of Registered Filters

One of the major system requirements for the C++ clients is that they be able to provide a list of all the Filters registered locally to the system's Central Controller. The C++ clients use a COM enumerator interface to enumerate all of the Filters available on the local computer by type.

There are eight Filter types recognized by the C++ clients. They are: Audio Capture Sources, Audio Compressors, Audio Renderers, Device Control Filters, (Generic) DirectShow Filters, Midi Renderers, Video Capture Sources and Video Compressors. Incidentally, these are the same Filter categories that GraphEdit uses for locating Filters.

When a C++ client is brought on-line, it builds a full list of all of the Filters on the system. Whenever it receives a request from the JavaServer to transmit the local registry, it transmits this list. Currently, the system does not allow a user to download Filters to a computer and have them registered there, and so there is no need to have the C++ clients rebuild their registry lists. The possibility of adding this feature to the system is discussed in Appendix I, section I.2.5.

4.1.3 Filter Graph Management and Issuing Updates

Aside from issuing an unprompted message when connecting into or disconnecting from the system, the C++ client software is designed to send messages only when prompted to by messages from the JavaServer. The C++ client waits for requests for information or instructions on Filter Graph configuration or dataflow. Once an instruction is received, the C++ client acts on it and then replies with either an update on its status in the case of success or an error message explaining what went wrong in case of failure. A more complete discussion on upward and downward communication, including the C++ client's responses to requests and instructions appears in section 4.5.

The C++ client is able to load Filters to its Filter Graph, unload them or make or break connections between Filters. I designed the C++ clients to be able to handle up to 200 Filters, under the assumption that no researcher would ever try to run more than 200 Filters on a single machine due to processing power limitations on microprocessors. It keeps track of the entire local Filter Graph, including the names of the Filters, their pins and the pin connections between Filters. The C++ client also keeps track of the Filters whose last-updated status has been transmitted to the JavaServer. This way, when sending an update on the Filter Graph status in response to a successful execution of a Filter Graph configuration instruction (Load / Unload / Connect / Disconnect), the C++ client is able to send information on only the Filters whose configuration information changed in response to the last issued instruction. This measure reduces the amount of upwards communication traffic, since the typical configuration operation affects only one to three Filters, and so information on only 1 to 3 Filters must be sent rather than information on the entire local Filter Graph. The C++ client can handle over 100 Filters.

Aside for Filter Graph configuration, the C++ clients must also handle dataflow. DirectShow allows Filters to operate in one of three dataflow modes – Play, Pause and Stop. Other dataflow issues, including buffering, transfer of data through shared memory buffers, etc. are handled directly by DirectShow dataflow classes and do not need to be controlled by the Filter Graph manager. In response to any dataflow-related instruction, whether it is a request for the current dataflow mode or an instruction to alter the dataflow mode, the C++ client always replies with the dataflow mode. In the case where an instruction to alter the dataflow mode is received, the C++ client first attempts to carry out the instruction and then responds with the dataflow mode, which usually is the same as the requested new mode.

4.1.4 Inter-System Connection and Display Filters

In order for the system to function over multiple computers, several Filters were needed. First, I designed a pair of Filters to transmit and receive data over a network connection.

These were necessary to allow inter-system connections. Since my aim in building this system was to design a system for computer vision research, these Filters were designed only to transmit video data, not audio data. In theory, I could build transmission and reception Filters for audio data, and then the system could be used for video conferencing. The Filter pair is able to transmit either uncompressed or compressed using the Foveal compression. I chose not to include other types of compression due to the general lack of compression Filters on computers – only decompression Filters are widely available. The possibility of adding compressed video is discussed in section I.1.7 of Appendix I.

Aside from the transmission pair, there were two other Filters that the system required. One of these was a displaying Filter. This Filter was built out of the standard video transmitter Filter described above. The difference between the two Filters is that this one is able to broadcast to up to ten different locations. The other Filter is the *Null Renderer.* The Null Renderer is a Renderer Filter that accepts any type of input data and does not render the data. Its main purpose is to serve as an end point for Filter Graphs. In order to function properly, a Filter Graph needs to have starting point and end point for data. In the absence end points, DirectShow will simply not allow dataflow to begin. While DirectShow does come equipped with renderers, since the computers that are running the C++ clients are not assumed to be within reach of the researchers using the system, it would be inappropriate to render data to these remote computers. This is especially true in view of the possibility that another individual might be using a computer that is running the C++ client software to run other software. It would be most annoying for that individual to have a video stream pop up on his/her computer screen. Further, were this person to close the video window, the C++ client's Filter Graph would be stopped, an effect that would be detrimental to the researcher using the system. To avoid this tug-of-war scenario between the researcher logged in remotely and a local user, it would be far simpler if the researcher had a renderer Filter that would serve as an end point but would not disrupt any local users.

4.2 JavaServer Implementation

The JavaServer is the Central Controller in the system. It is the central hub of the system and the only component that must be running for the system to exist. This last detail is important. Since the JavaServer is assumed to have been running since the system went online, only the JavaServer is assumed to know which Filters are *undisplayed*, and which node is the display system set to. One of the responsibilities of the JavaServer is to transmit three lists to any Applet receiving a Filter Graph update from any C++ client. Since the JavaServer is the central hub and the point of system access for the user interface, it is the only piece of software that knows when an Applet is being sent a message, since all upward messages must be sent through the JavaServer. The three lists that the JavaServer sends to the Applets are (1) a list of the *undisplayed* Filters, which the Applet will treat differently, (2) a list of Filter connection corrections for the Central Filter Graph list and (3) the location of the displayed node.

Both the Applet and the JavaServer keep track of all of the Filters in the system. Each Filter is cross-referenced on two lists – a central listing and a listing indexed by C++ client. Whenever an Inter-System connection between two Filters is made, the transmitter and receiver Filters as well as any necessary intermediate Filters are set to *undisplayed* in the central listing by the JavaServer. The JavaServer then adjusts the central list so that the two Filters that are connected by the inter-system connection are listed as connected to each other rather than the transmission pair. In order for the Applet to display this connection, the JavaServer informs the Applet that the transmitter and receiver Filters are to be left undisplayed for the Applet's full system view, and a link should be drawn connecting the two Filters that were connected over the network by the transmission pair.

Since the user can log off and log in at any time, the JavaServer always transmits the full lists of corrections to the Applets. The displayed node's location is transmitted so that the Applet can draw a red circle over that node to indicate to the user that it is the designated node for the display.

The JavaServer has access to all of this information because it is the point where these special commands are issued. One of the services it provides is to perform these complex commands. When the user requests an inter-system connection through the Applet, the JavaServer's thread which is handling the user's Applet receives the request and then begins transmitting instructions to the threads handling the C++ clients to load the necessary Filters and make the necessary connections. The JavaServer takes care of setting up the receiver Filter so that it can access the transmitter Filter's listening port.

When the user requests that the displayed node be set or moved, the JavaServer also takes over, issuing commands to load and connect the display Filter. It also keeps track of the displayed node, for the benefit of the Applets, and the machine and port number where the displayed stream can be accessed. If a user Display program logs in, the JavaServer can then give it the IP address and port number of the display stream so that it can access the display.

Since the JavaServer is of multi-threaded design, and since its main function is to route messages from Applets to C++ clients and back, the JavaServer required a method for inter-thread communication. To that end, I implemented the ICMH or inter-client message handler. Essentially, this is a Java class that all of the threads have access to. It contains all of the stored system data, as well as a list of all connected clients including their name and type (C++ client, Applet connected as master, Applet connected as View only, Display program) and a set of communication pipes. Each client-handling thread in the JavaServer must listen to its input pipe as well as its socket. The socket carries messages from its client, whereas the pipe carries messages generated by another thread.

Among the capabilities of the ICMH is the ability to deal with the possibility of a disconnecting client. Though the ICMH does not check for downed connections, a future enhancement to the system discussed in Appendix I, section I.1.5, the ICMH contains code to handle the situation where when a client logs off from the JavaServer. For Applets, cleaning up after a logging off means checking if the Applet was the system's master user, and if so, selecting another connected user to be promoted to master status.

In the case of a C++ client logging off, the JavaServer has several things to do. It re-indexes the list of connected C++ clients, wiping out the Filter Graph and Filter registry listings for that C++ client and removes all Filters loaded on that client from the Central listing. It also informs the Applets that a C++ client has logged off so that they can take similar actions. I have thought of another approach to responding to C++ clients logging off. It can be found in section I.1.5 of Appendix I.

The ICMH is also used to inform connected Applets that a new C++ client has connected to the system. As well, since the ICMH keeps track of the system's master user, the JavaServer's Applet-handling threads consult the ICMH whenever they receive an instruction over their socket that would change the configuration or dataflow of one or more Filter Graphs on the system.

4.3 Applet Implementation

Since one of my goals was full Internet access to the system, I decided that to implement the Applet using standard Java, without any language extensions such as Microsoft's Visual J++. This meant that all of the Applet's graphics had to be designed using Java's Abstract Windowing Toolkit, or AWT. The AWT is a platform independent toolkit that includes many Java classes that are useful for graphics generation, but lacks a fair amount of the functionality available when using platform-specific Java library extensions.

The Applet is of dual-threaded design. The main thread is responsible for handling logging into the system and the Applet's graphical display, including transmitting user commands to the JavaServer. The secondary thread is responsible for receiving and handling all upwards messages. Just as the JavaServer's threads share access to the ICMH, the Applet's two threads share access to the Applet's parent class – distribApplet – which stores a full record of the system's status. This dual-threaded design allows the Applet to continue to function after issuing an instruction without waiting for a response from the JavaServer. When the response arrives, the Applet's information is updated and its display changed to reflect the changes in the system.

The Applet shares a significant portion of its code with the JavaServer. Most of the code accessed by the secondary Applet's thread, which handles all post-login upward communication, is almost identical to code in the JavaServer. As well, the Applet uses the same data structures as the JavaServer to store information on the registries and Filter Graphs of all computers on the system.

4.3.1 The Applet's Graphical Display

The Applet's graphical display was designed for ease-of-use. Most instructions require simple mouse point-and-click or point-and-drag operations. The only exceptions are the login window, which requires the user to type his/her username and password, and the remove/unload Filter and remove/disconnect Filter connection instructions, which require the user to press the 'Delete' key when the object desired for removal has been removed. The Applet presents users with a variety of information. The upper tab of the Applet informs the user whether he/she is connected as the master user or not. It also informs the user on the system's dataflow mode and presents the user with a list of all C++ clients connected to the system, from which the user can select one C++ client at a time as the *focus*, the C++ client where all Filter loading instructions will be directed.

The Applet's right tab has two purposes. In *Filter Insert* mode, it allows the user to view the registry of the current *focus* C++ client and select a Filter to be loaded. The list is indexed by category and alphabetically sorted. The lower part of the right tab instructs the user on how to use the Applet to insert a Filter onto the Filter Graph.

The other function of the right tab is *Display Node* mode. In this mode, the user can select a node to be the displayed node or stop the display. Once again, the bottom of the right tab instructs the user on setting up or turning off the display.

The bottom of the Applet is a message log window. It presents certain messages to the user, and will change the text colour for certain messages. For instance, messages

indicating a change in the system such as a new C++ client connecting are indicated in blue. Error messages are shown in red. These colours are used to draw the user's attention to important information. Unfortunately, Java's AWT is very restricted in its graphical abilities to maintain compatibility with all operating systems. Thus, when the message log's text colour is changed, all text in the message log is changed to that colour.

Though the bottom tab contains a "Save Log" button, this button as of yet, has no functionality tied to it. The possibility of saving the log and later retrieving it is discussed in Appendix I, section I.1.6.

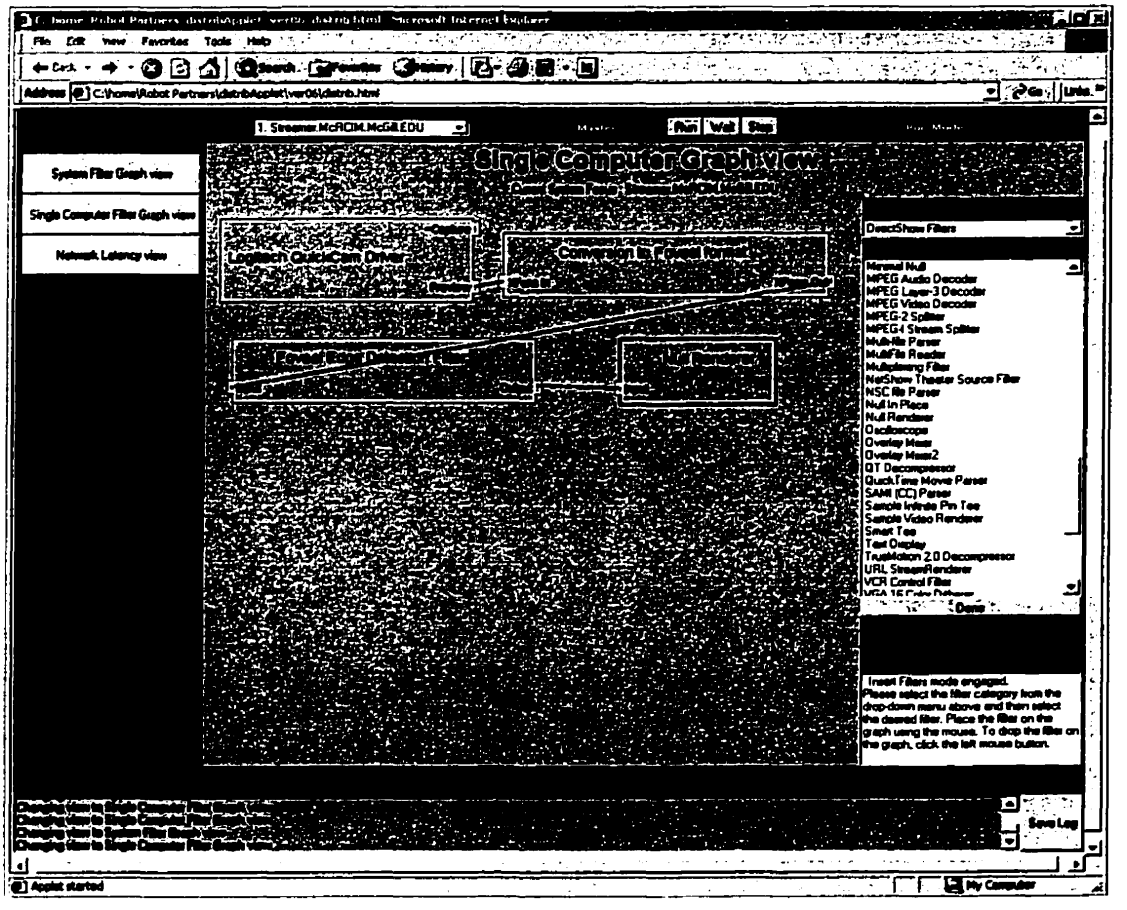

Figure 4.3-1: Applet in Single Computer Graph view with the right tab in Filter Insert mode. Compare with Figure 3.5-1.

The Applet's left tab is used for switching between the different views available for the centre of the Applet. The Network Latency view is part of a planned future enhancement

discussed in section I.1.1 of Appendix I. The other two views present Filter Graphs. The System Filter Graph view presents all of the Filters loaded on the entire system, except the *undisplayed* ones, as a single Filter Graph. The Single Computer Filter Graph view, shown in figure 4.3-1, presents the Filter Graph on the current *focus*. This view includes *undisplayed* Filters loaded on the current *focus*.

4.3.2 Information Request System

One of my design objectives was to reduce the amount of vertical communication traffic through the system. In a multi-user system, all of the users must be updated on system changes. Since I designed the JavaServer as a multi-threaded server, where no thread ever waits for replies to its messages, the JavaServer always routes any information update to all connected Applets, not just the Applet that issued the request. Every time an Applet requests information from one of the C++ clients, all of the Applets receive the reply. If many unnecessary messages were to be issued, the operation of the JavaServer, which generates all of these messages, would be unnecessarily hampered.

I decided to design the Applets so that they would never request any information they already had. Each Applet would keep tabs on whether its information on the C++ clients' Filter registry lists, Filter Graph configurations and Filter Graph modes was up-to-date. Before allowing a user to access the system after a successful login, the Applet would pick up all of the data it was lacking.

I also did not want an Applet issuing more than one instruction at a time. In this way, I would keep the system from being overloaded by multiple identical requests from all connected Applets in response to some occurrences such as a new C++ client connecting to the system.

The Applet's information request system is invoked after any system update, Filter registry update or Filter Graph update message is received by the Applet. If the Applet is lacking information, it determines its single most pressing lack of information and issues

a request for that information. After an Applet has logged in and finished updating itself on the system, it only requests data whenever its focus is changed and the only other upward communication traffic in the system is issued by Filter Graph modifying instructions issued to the system by the master user's Applet.

4.4 Display System Implementation

There were two parts to the implementation of the display system. First, there was the work of setting up the display at a given node. The theory behind this was discussed in chapter 3 and the implementation of this has been discussed along with the that of the JavaServer and the Applet since this part of the display system was largely implemented within the Applet and the JavaServer. Briefly, the master user is able to select a single node to be displayed. If one is already selected, he/she can select a new node to be displayed, and the JavaServer will take care of removing the display from the previously displayed node. The JavaServer keeps track of the node that is displayed and transmits this information to all connected Applets. The Applets indicate the displayed node by drawing a red circular dot on the Applet's graphical representation of that node.

4.4.1 User's Display Program

The second part of the Display system's implementation is the transmission and delivery of the display stream. Unlike the display system set up, this part does not involve the Applet at all, and its involvement with the JavaServer is minimal. The lion share of the implementation of this part of the display system is the implementation of the User's Display Program.

The user's display program is designed to run on computers running one of Microsoft's Windows operating systems. Since I designed it as a simple Filter Graph manager it also requires DirectShow be installed to function. It is designed to operate two socket connections. The Display program briefly opens a socket connection to the JavaServer. Upon identifying itself as a Display program (as opposed to a C++ client or Applet

connection), the JavaServer transmits a message through the socket identifying the location of the display stream by IP address and port number and the display stream's data type information. The JavaServer then terminates the connection.

Once the Display program has this information, it constructs a Filter Graph with a receiver Filter, to pick up the stream, and a Video Renderer Filter to display the stream on the local computer screen. The display system transmissions, like the inter-system connections, allow streaming data to be either uncompressed or compressed with the Foveal compression format. In the case of a Foveal compressed stream, the Display Program places a Foveal decompressor between the receiver Filter and the renderer.

As mentioned in section 4.1.4, the Display Filter allows a maximum of ten connections. Thus, it is possible that a Display program's attempt to connect to the Display Filter will be unsuccessful. However, the Display program will continue attempting to connect indefinitely unless it is made aware of a new Display node. If the master user moves the display to another node, all of the connected Applets will reflect this information on their graphical display. The user can then redirect the Display program to the new node by clicking on the 'Reload' button on the Display program. This will cause the Display program to remove the Filters it has loaded in its Filter Graph and reconnect to the JavaServer to restart the process of retrieving the display stream information and building a new Filter Graph to connect to it and display it.

4.5 Inter-Program Interaction Protocols and Functionality

Chapter 3 explained the necessity of upwards and downwards communication in the distributed system. Essentially, upwards and downwards communications transmit user commands to the C++ clients residing on the computers in the system and update system status information retained in the JavaServer and the Applets connected to it. In order to handle information interchange, I devised a communication protocol for use in upwards and downwards communications. The protocol itself consists of text messages, each with an associated number and name, and has a resemblance to Telnet or SMTP protocols.

A secure system, complete with secure transmission protocols, was not one of the objectives of this work on the distributed system. The work was more a proof of concept than anything else. For this reason, I made no effort to add data encryption to the upwards/downwards communication protocol. I did, however, include an identifier in every message called the "authentication tag". In the current implementation of the distributed system, this tag was constant, always indicating the point of origin of a message – "JS" denoting the JavaServer, "CPPC" denoting a C++ client and "APPL" denoting an Applet connection. However, this could be modified, at some future time, such that the tag would change after each message was sent or received. Ideas for adding security features to the system's communications are discussed in Appendix I.

The basic form of messages under the *vertical* communication protocol is the same for both upward and downward directions. The only commands that do not obey the standard form for messages are commands associated with connecting to and disconnecting from the JavaServer. Other than this, messages sent between the JavaServer and Applets differ from those sent between the JavaServer and C++ clients by a single tag indicating the C++ client the message is intended for(downward) or is referring to(upwards). Here are the message formats:

Format #1 <cmd#><auth><cmd name><parameters.....> (JavaServer – C++ client messages)
Format #2 <cmd#><auth><cmd name><cppc#><parameters....> (JavaServer–Applet messages)
where:
<cmd#> denotes the command number, a 3 digit number indicating the command
<auth> denotes the authentication tag. "APPL", "JS" or "CPPC" are acceptable tags.
<cmd name> denotes the command name.
<cppc#> indicates the index o the C++ client on the system that the message is intended for or referring to (JavaServer – Applet messages: Format 2 only). A negative value for this identifier indicates that the message is intended for all C++ clients on the system.
<parameters ...> are additional message-specific parameters
And each message must end in a CR character.

Under the system, the JavaServer acts in part as a relay station. A messages sent from a connected Applet and intended for the C++ clients is received by the JavaServer, then converted from format #2 to format #1, with its authentication tag changed from "APPL" to "JS". Then the message is retransmitted to the C++ client(s) indicated in the original message's <cppc#> field. A message sent from a C++ client connected to the system is received by the JavaServer. The JavaServer records any relevant data and then converts the message from format #1 to format #2 by adding a tag identifying the C++ client where the message originated, and then changes the authentication tag from "CPPC" to "JS". Afterwards, the message is retransmitted to all of connected Applets.

Some messages sent between the Applet and JavaServer are not intended to be delivered to the C++ clients. These messages, generally requests for special services offered by the JavaServer such as initiation of inter-system communications between two filters running on different C++ clients, are stopped by the JavaServer, which then performs some processing on them rather than modifying and retransmitting them.

4.5.1 Downward Communications

Downward communications consists of commands and information queries sent down the chain from an Applet to the JavaServer or from the JavaServer to a C++ client. Whenever a message is sent downward from the Applet to the JavaServer, the JavaServer first checks the message's authenticity, by verifying the authentication tag and confirming that the message number and message name correspond. The basic commands required for the distributed system to function are:

1. Add/Load a Filter onto the Filter Graph
2. Connect a Filter to another Filter
3. Break a connection between two Filters
4. Remove/Unload a Filter from the system's Filter graph
5. Change system's Filter Graph mode (run/wait/stop mode)

In addition to the downward commands, there are several information queries messages required for the downward communications. These are:

1. Requesting Filter Registry information
2. Requesting Filter Graph information
3. Requesting Filter Graph mode information

Finally, the list of messages in the downward communications protocols include several error messages that can be sent downwards in response to misunderstood commands in upwards communications.

Command number	Command name	Description
001	ERR	Error: Unknown login (in response to an incorrect C++ client login sequence)
004	ERR	Error: No space left (the JavaServer can handle a maximum number of connections)
100	CPPCconnected	JavaServer confirming C++ client login attempt (*JavaServer to C++ client only*)
100	none	Applet login attempt beginning (*Applet to JavaServer only*)
101	none	Applet login: Username field
102	none	Applet login: Password field
104	SYSUPDATE	Applet requesting system update including names of all connected C++ clients and the size of their respective registries. (*Applet to JavaServer only*)
110	REGDATA	Requesting information on full registry
111	REGDATA	Requesting information on a single registry entry
120	GRAPHUPDATE	Requesting information on Filter Graph changes since last update
121	GRAPHUPDATE	Requesting information on entire Filter Graph
122	GRAPHUPDATE	Requesting information on a single Filter loaded on the Filter Graph
20X (200, 201)	ADDFILTER	Add/Load Filter to Filter Graph 200 – by name, 201 by index number in registry
21X (210, 211)	ADDCON	Connect two Filters on the same Filter Graph 210 – Filters and pins identified by name, 211 – Filters and pins/pin type identified by index numbers on Filter Graph and Filter respectively
220	ADDNETCON	Connect two Filters on different C++ clients through network transmission filters. (*Applet to JavaServer only*)
251	FILEIO	Requesting port number used by a C++ client for most recently loaded network transmission Filter (*JavaServer to C++ client only*)
252	FILEIO	Send information to C++ client to set up network receiving filter to connect to specified computer/port number with specified data type. (*JavaServer to C++ client only*)

30X (300,301)	REMFILTER	Remove/Unload Filter from Filter Graph 300 – by name, 301 – by index on Filter Graph
310 (310-313)	REMCON	Disconnect two Filters on same C++ client 310 - refer to a single Filter and pin pair by name 311 - refer to both Filter and pin pairs by name 312 - refer to a single Filter and pin pair by index number on Filter Graph and Filter respectively 313 - refer to both filter and pin pairs by index number on Filter Graph and Filter respectively
320	REMNETCON	Disconnect two Filters/Break Filter connection on different C++ client (*Applet to JavaServer only*)
400	GRAPHMODE	Place Filter Graphs in Stop mode
401	GRAPHMODE	Place Filter Graphs in Wait mode
402	GRAPHMODE	Place Filter Graphs in Run mode
403	GRAPHMODE	Requesting current Filter Graph mode information
450	PING	Network latency – *function not fully implemented*
475	DISPLAY	Requesting system display be set at a node specified in the message parameters (*Applet to JavaServer only*)
500	QUIT	Breaking connection

Table 4.5-1: Full list of downward messages

Table 4.5-1 contains a full list of all downward commands, listed by command number. Note that since communication between the user's Display program and the JavaServer is not considered part of upwards/downwards communications, messages sent between the user's Display program and the JavaServer are not included in table 4.5-1 or in this entire section of chapter 4.

4.5.2 Upward Communications

Upwards communications consists mainly of responses to downward requests and commands. Unlike downward communications, which consists of single-line messages, upwards replies are often compound messages. For instance, each full registry update consists of one line indicating the total number of registered Filters, eight lines indicating the number of Filters registered in each of the eight Filter categories and one line for each registered Filter indicating its index number, category and name. Thus, a C++ client running on a computer that has 130 registered Filters (typically, computers have between 120 and 140), transmits a 139-line message every time it receives a request for its list of registered Filters.

As well, each Filter Graph update includes one line indicating the number of Filters loaded on the Filter Graph and the number of Filters whose information is included within the update. There is also one line for each Filter in the update, indicating its name, index and the number of input and output pins it has. As well, there is one line for each pin indicating its name, type (input/output), index, indices of Filter and pin it is connected to (-1 if not connected) and the media type used between this pin and the one connected to it for information interchange. Thus, a Filter Graph update consisting of two transform Filters, each with a single input pin and single output pin, is seven lines long. A full Filter Graph update for a Filter Graph consisting of ten Filters can easily exceed thirty lines. Typically, when responding to successful execution of a Filter Graph configuration commands, the C++ clients only issue a short update, transmitting information on the Filters that changed due to that command.

Most upward messages are received by the JavaServer from the C++ clients and then slightly modified (see message formats above) and rebroadcast to all connected Applets. Since both programs are written in Java, they share a fair amount of code used for authenticating and deciphering upward messages. It should be noted that, though both the JavaServer and the Applet contain a central listing for all Filters, the assumption that the central listing on each connected Applet is identical to the central listing on the JavaServer is false. Though the central listings on all connected Applets contain the same information as the one on the JavaServer, the indexing of the Filters on the central listing may differ since the Applets do not need to be logged into the system at all times for the system to function. Whereas the indexing for the individual C++ client Filter Graphs is handle by the C++ clients themselves, the central listing's indexing is not fixed by any one program. While the JavaServer's central list reflects the order in which Filters were loaded and unloaded on the system's C++ clients, the indexing on the Applets is partly a function the point during the system's operation when that user logged in. This means that communication between the JavaServer and the Applets can never use a Filter's index on a central listing.

Chapter 5. Experimental Results

5.1 Frame Rates and Throughput Results

Frame rates achieved on the system are really more a matter of DirectShow's performance than the system's. The system provides a framework for controlling numerous DirectShow-based Filter Graph Manager programs – the C++ clients – on multiple computers. Still, it is useful to showcase the capabilities of the system. The tests on throughput were run on GraphEdit using either an Axis 200+ NetCam connected directly to the CIM network at McGill University in Montreal, Quebec, Canada or a Logitech QuickCam VC computer camera (with a parallel port connection).

5.1.1 Single Computer Frame Rates

The first tests shown here were run on a single computer that is part of the system, with no displayed node set. These results provide a baseline for the system. If multiple computers are used or if a display is being transmitted to one or more users, the results shown in this section would indicate the best frame rates one could expect to achieve.

Table 5.1-1 shows results achieved using the QuickCam under various resolutions, while table 5.1-2 shows results achieved using the Axis 200+ NetCam. Several points should be noted. First, in 8 bits per pixel mode, the Logitech QuickCam provides a Greyscale image. In 16 bits per pixel mode, the QuickCam must take a 24 bits per pixel colour image and convert it to 16 bits per pixel, using RGB 555 format. Second, the Foveal compression always turns image into a square image SxSx24 bits per pixel, where S is a multiple of 32. Note also that the QuickCam's capture mode has a higher frame rate but has a longer delay between capture and display. This difference is not apparent when the image is merely being rendered directly, however it becomes acute when the signal is operated on. In 640x480, with Foveal compression the delay in capture mode is ~10-12 seconds long, whereas the delay in preview mode is ~1-2 seconds long. The QuickCam's maximum frame rate is 30 frames per second (fps).

Resolution	Frame Rate (preview mode)	Frame Rate (capture mode)	Frame Rate Foveal (preview mode)	Frame Rate Foveal (capture mode)
160x120, 8bits	19.68 fps	29.97 fps	19.56 fps	29.97 fps
160x120, 16bits	18.62 fps	29.97 fps	18.52 fps	29.96 fps
160x120, 24bits	19.08 fps	29.97 fps	18.87 fps	29.97 fps
320x240, 8bits	9.50 fps	13.66 fps	8.78 fps	12.79 fps
320x240, 16bits	8.77 fps	12.29 fps	8.06 fps	11.53 fps
320x240, 24bits	9.09 fps	12.95 fps	8.48 fps	12.23 fps
640x480, 8bits	9.03 fps	12.92 fps	4.00 fps	5.69 fps
640x480, 16bits	8.27 fps	11.77 fps*	3.79 fps	5.51 fps
640x480, 24bits	7.92 fps	11.60 fps*	3.84 fps	5.73 fps

Table 5.1-1: Frame Rates achieved using a Logitech QuickCamVC connected to a single computer in the distributed system with no displayed node set.

Notes:

1. FPS = frames per second.
2. The Logitech QuickCam VC that I used was not stable in tests in capture mode at resolutions of 640x480x16 bits per pixel and 640x480x24 bits per pixel without Foveal compression.

The Axis 200+ NetCam is presented for comparison. It is a camera designed for connection directly to a network port. The NetCam captures single still images in JPEG format. The Source Filter designed by one of my colleagues accesses the NetCam and creates a stream out of the still images. The resulting frame rates are rather low compared to the QuickCam.

Resolution	Internal Compression	Frame Rate (uncompressed)	Frame Rate (Foveal Compression)
176x144, 24bits	Low	0.50 fps	0.48 fps
176x144, 24bits	Medium	0.52 fps	0.50 fps
176x144, 24bits	High	0.52 fps	0.51 fps
352x288, 24bits	Low	0.43 fps	0.39 fps
352x288, 24bits	Medium	0.48 fps	0.45 fps
352x288, 24bits	High	0.49 fps	0.50 fps

Table 5.1-2: Frame Rates achieved using an Axis 200+ NetCam accessed by a single computer on the distributed system with no displayed node set.

5.1.2 Multi-Computer Frame Rates and Inter-System Connection Throughput

Single computer frame rates are of limited use for the system. If a researcher wanted to use only a single computer to avoid the detrimental effects of limited transmission bandwidth, it would be far simpler to use Microsoft's GraphEdit. Table 5.1-3 indicates the frame rates achieved on the system using a QuickCam. The data has been transmitted from the computer with the QuickCam running GraphEdit to another computer also running GraphEdit and then back to the original computer. For the Foveal-compressed frame rates, the computer with the QuickCam was running both the Foveal compressor and decompressor Filters.

Resolution	Frame Rate (preview mode)	Frame Rate (capture mode)	Frame Rate Foveal (preview mode)	Frame Rate Foveal (capture mode)
160x120, 24bits	12.78 fps	17.02 fps	11.71 fps	15.03 fps
320x240, 24bits	7.13 fps	12.00 fps	5.86 fps	10.55 fps
640x480, 24bits	6.59 fps	11.51 fps	3.40 fps	4.92 fps

Table 5.1-3: Frame Rates achieved using Logitech QuickCamVC data transmitted between two computers in the distributed system.

Theoretically, a C++ client running on a computer with a 100 Megabit per second channel can transmit up to 56.9 frames per second at 320x240x24bit, 14.2 frames per second at 640x480x24bit and 227.55 frames per second at 160x120x24bit. However, as these results show, each inter-system connection (and each display connection) reduces the system's Filter Graph frame rate.

5.1.3 Upward and Downward Transmission Bandwidth Requirements

The all of messages transmitted down through the system and many of the possible upward messages consist of single line messages. These messages are typically 50 bytes in length or less. The majority of messages sent upwards in the system, however, involve either the list of Filters registered on a C++ client's computer or updates on a C++ client's Filter Graph configuration or dataflow mode. Filter Graph dataflow mode updates are single line messages, but Filter Graph configuration updates and Filter registry

information are always multi-line messages. In tests on the system, I measured the typical length of Filter registry list information messages at 5 to 6 kilobytes. As well, though most Filter Graph configuration updates involve less than four Filters, with typical update lengths of 50 to 400 bytes, transmission of a C++ client's full Filter Graph configuration information can possibly exceed 100 kilobytes. It should be noted that this is not typical. Based on limitations on computer processing power on microprocessors, the typical C++ client would have less than 20 Filters loaded at any given time. Under those circumstances, a C++ client could transmit its entire Filter Graph's configuration in less than 2 kilobytes.

So far, I have referred only to the size of individual transmissions. While downward messages are sent from one source and received at a single destination, upward transmissions between the JavaServer and the Applets are broadcast to all connected Applets, with few exceptions, such as error messages. Thus, the transmission bandwidth required by the JavaServer to transmit a message upward is the message length multiplied by the number of connected Applets. The more Applets are connected to the JavaServer, the longer it will take a large upward transmission to reach all of them.

5.2 Full System Tests

The results discussed thus far have been dealing with components of the system rather than its overall performance. Aside from measure MIPS or FLOPS, there is no true quantifiable measure of a system's performance. In this system, given that the computing power available is not fixed, these measures are meaningless and pointless to compute. The processing power available to the system's master, in terms of MIPS and/or FLOPS, depends on too many varying factors. The first factors to consider are the number of computers that are on the system and processing power available on each of them. These values can yield a theoretical upper bound on the system's performance. However, they do not tell the whole picture. The system Filter Graph's configuration also plays a role in determining the available processing power. This is especially true in the case where the number of inter-system connections is high and the display system has several connected

User Display programs. The system Filter Graph configuration also influences the effective MIPS/FLOPS available to the user. In cases where there are many computers on the system, it is entirely possible that some C++ clients may have no Filters loaded in their local Filter Graphs, and so, though they are present on the system, they will not be contributing to the system's performance. This brings up another point. Though the system can be viewed as a single entity by the user, it is actually a collection of separate computers. Though the total processing power of the system may be in the range of billions of operations per second, the actual processing power available on any given computer in the system is far lower. Further, transmission costs, in terms of both processing power drain by transmitter and receiver Filters and transmission effects such as latency and slow connection, can further slow down all Filter Graphs on the system, and reduce the effective number of operations per second of the system.

Thus, the only type of system measure that is valid and meaningful is the successful or failed results of tests on the entire system during typical operations. During the course of the implementation of the distributed system, the system was put through preliminary testing of its functionality. The system was tested with up to three C++ clients connected at a time, with multiple inter-system connections running between them. These tests were all successful once all of the implementation issues were worked out. Due to a general lack of vision algorithms among DirectShow's library of Filters, tests were usually performed using the Filters I designed to operate on Foveal-compressed video streams.

The display system also functioned well. This result was expected since essentially, the display Filter was almost identical to the inter-system connection Filters. Unfortunately, though I would have liked to be able to test the effects of running the system with a user viewing the display while logged in on a slow Internet connection or a remote high-speed connection, the logistics involved proved too onerous. For the slow connection, I could have used my home computer network to log in, however, this would have necessitated several trips per testing session between McGill and my home at a round trip time of just under two hours. The remote high-speed option was also interesting, though it would have required the outside assistance that could not be guaranteed.

Chapter 6. Conclusion and Comments

6.1 Fulfilment of System Requirements

This thesis presents a new approach to the design of vision systems. Overall, the system has met the requirements for a useful distributed vision system. The JavaServer fills the role of system central controller well, providing all of the functionality required of it. It bears noting, though, that the JavaServer requires a fair amount of processing power to function. Often, the JavaServer uses up much of the processing power and communication bandwidth on the computer that is running it.

The C++ clients are also quite successful in their role as local processing block controllers. They provide the system with all of the functionality required by the system's specifications.

The Applet and Display system are less than optimal, though. This is especially true for the display system, which presently can only transmit video to users on Windows-based machines. Though it is true that work is proceeding to port DCOM over to Linux [5], it is still unclear as to whether DirectShow software will be able to run on the Linux version of DCOM. Without a useable Linux version of DCOM, a Linux version of the User's Display program will need to be designed. Further, even with a Linux version of the Display program, the user will still need to download the Display software in order to use it. This solution is far from optimal, yet given the available choices, and the objective of minimizing communication traffic through the JavaServer, this was the best solution possible.

The Applet also is lacking in important functionality. Much of this is discussed in Appendix I. The most pressing deficiency in the Applet is its lack of property page support. Without property page support, researchers lose flexibility in fine tuning algorithms on the system. This feature is discussed in section I.2.2 of Appendix I.

6.2 Non-Standard Applications of the Distributed System

So far, I have discussed the possibility of using the system as a research tool to aid researchers in testing algorithms under real-time constraints. Aside from reasearch and development applications, the distributed vision system can serve in a wide variety of commercial and industrial applications. As well, with some additional work, this system's research capabilities could be expanded greatly. For the sake of brevity, I will only cite one example for each.

With some additional work invested to design an equivalent to the C++ client for use on other types of computers, the system could be expanded beyond Windows-based personal computers. Section I.1.8 of Appendix I discusses the possibility of designing a Linux version of the C++ client software. This is not the only possible extension. A suitable C++ client equivalent could be developed for mobile robotic systems connected to a network using wireless network connections. This would allow the robots to gather data and move about while offloading any computationally intensive decision making algorithms to the computers on the system. In fact, designing a C++ client for the robotic system is not even a necessity. A DirectShow Filter could be designed to interface with the robotic system in the same way the current Axis 200+ NetCam Source Filter access the NetCam's IP address and pulls data from the NetCam. Perhaps an even better model is the Pan-Tilt Camera Source Filter and PTU Control Filter. Together, these Filters access a video source and then transmit messages to its PTU, giving it instructions on what direction to pan or tilt. A mobile robot connected to the distributed system in this manner would require very little onboard processing capability and yet have access to tremendous processing power. Further, the system would be very easy to debug.

The distributed system could also serve commercial purposes. One of the best examples of this would be a security system. In an office setting, the distributed system could be set up on all of the computers. Each computer could have a video capture device, such as a Logitech QuickCam or Creative Webcam attached to it. The C++ client on each computer could then either run a face recognition algorithm on the video input, or

transmit the video data to another computer, perhaps compressing it first, which would run the face recognition on the video input. In the case of an unrecognized face, the system could either lock the computer or alert security by posting the video stream with the unrecognized face on the computer screen of the building's security guards. If this solution is too consuming of processing power, the same solution could be implemented using cameras mounted throughout the building.

In both cases, the distributed system could also serve a dual-purpose, and be used to supplement the building's night watchmen. During nighttime, the system could run the input gathered from video sources through motion detector algorithms based on detecting temporal differences between video frames.

6.3 Concluding Comments

Overall, the distributed vision system performs well. Aside from the property pages, it essentially allows a user to control multiple computers as if a single GraphEdit session is running on all of them. The system allows a user to configure it to run any vision algorithm, including serial/pipelined and parallel processing configurations.

One area of concern is network bandwidth used by the system. Currently, the system offers very little in the way of video compression for transmissions, aside from the lossy Foveal compression format. Though there are cases where transmission bandwidth is relatively more expendable than processor power, the current system offers no choice to the user on the method of data transmission. This is one situation that should be remedied. The Appendix that follows contains a large number of ideas for improvements to the system. It is my hope that many of these features be added to the system to improve it.

References

[1] Battaglia, M. P., "Parallelism for Imaging Applications." In *Wescon '93 Conference Record*, 1993, pp. 125-129.

[2] Messner, R. A. & Bloomfield J., "A Modular Advanced Pipelined Image Processing Accelerator." In *IEEE Aerospace Applications Conference*, Vol. 4, 1996, pp.407-422.

[3] Keahey, K. "A Brief Tutorial on CORBA", Advanced Computing Laboratory, Los Alamos, NM, USA, http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html

[4] Brando, T.J., "Comparing DCE and CORBA", *MITRE Document MP 95B-93* (March 1995). http://www.mitre.org/support/papers/tech_papers99_00/brando_domis/

[5] Vepstas L., Linux DCE, CORBA and DCOM Guide, Copyright (c) 1996-2000. http://www.linas.org/linux/corba.html

[6] Young, M., Argiro, D. & Kubica S., "Cantata: The Visual Programming Environment for the Khoros System." *Khoros Whitepaper, Khoral Research, Inc.*, January 18, 2000, pp. 1 - 9

[7] Van Reeth, F., Raymaekers, C., Trekels, P., Verkoyen, S. & Flerackers, E., "A Distributed Video Retrieval System Utilising Broadband Networked PC's for Educational Applications." In *IEEE Proceedings on Multimedia Modeling – MMM '98*, 1998, pp. 47-48

[8] Excalibur Video Market Backgrounder, online white paper : Excalibur Technologies Corp., 1999, http://www.excalibur.com/

[9] "DirectShow and COM", Microsoft DirectShow help file. 1998.

Appendix I. Towards the Future: A Discussion of Future Enhancements to the System

While defining the parameters and objectives of this thesis, I considered many ideas for features of this system. Those that were deemed essential to the proper functioning of the system were incorporated into the work done of the thesis, while others were left for later implementation. As well, during the implementation stage, ideas for additional useful features arose. Presented here is a discussion of the ideas for future improvements to the system.

I.1 Potential Future System Features

I.1.1 Network Latency Information

The concept for this feature is to provide the user(s) of the system information on the transmission latencies between computers tied in to the system. This would allow the user(s) to make more informed choices when deciding which computers in the system would host the processing blocks. Figure AI.1-1 depicts my concept of how this information would be presented to the user.

To get information on latency, two possible methods were arrived at, both of which rely on ICMP protocol. Either the system focus C++ client would send out a ping to each other C++ client in the system or each C++ client in the system would send out a ping to the system focus. The round-trip time of each ping would be divided in two to yield the approximate unidirectional transmission latency that would then be reported to the user-interface.

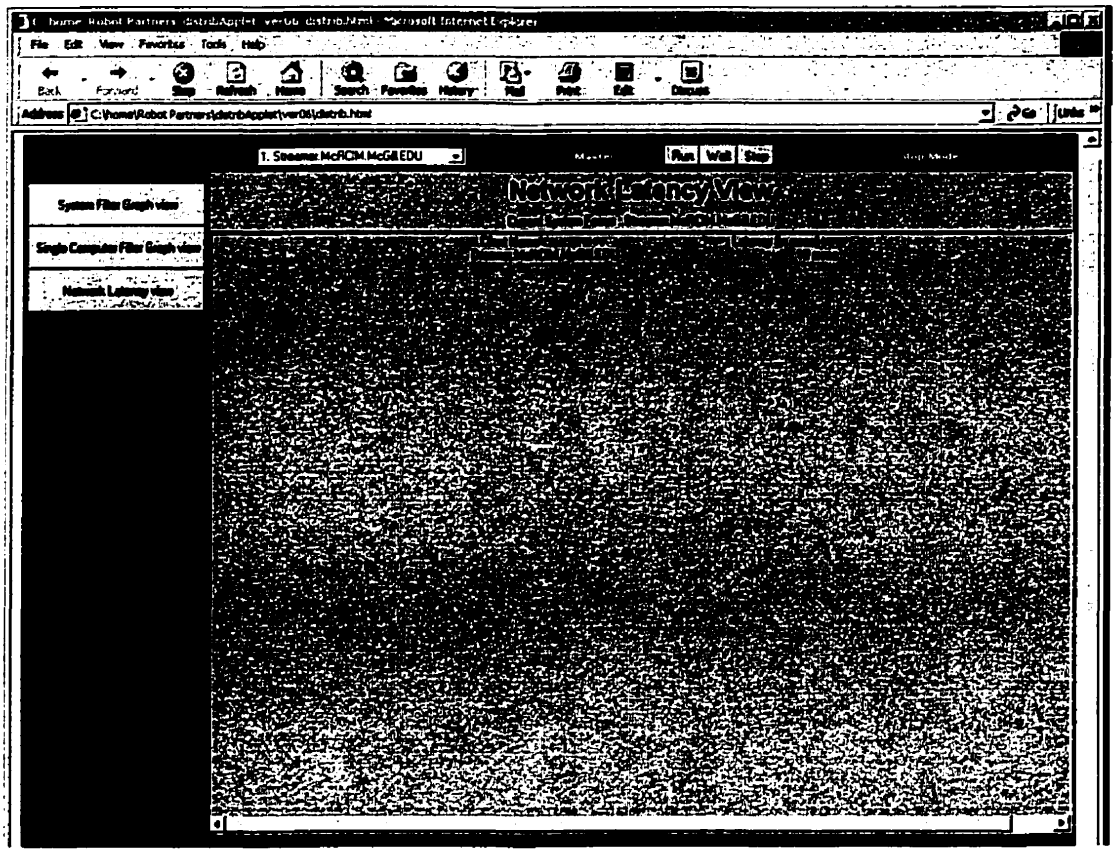

Figure AI.1-1: Network Latency view on User Interface

I.1.2 Computer Resource Utilization Information

Hand-in-hand with network latency information, is information on the computer systems where the C++ clients reside. Though the presumption exists that the system's master user has knowledge of the computers that the system is running on – their processor speeds, and memory capacity - this will not always be the case. Further, the user has no way of seeing how much of each computer's resources are available to the C++ clients. In some cases, the computers that run the C++ clients may actually be in use locally by another user, and may be running other applications besides the C++ client software. Further, while the user may have some qualitative knowledge about the relative amount of computer processing power consumed by each Filter, that knowledge is unlikely to translate into an accurate estimate of the available processing and memory resources on the computers in the system. Since the user may be located far away from the system's

computers that are performing the actual processing, the system must assume that the user cannot simply walk over to the computers and use local software to determine the computers' utilization of processor power and memory capacity.

A display in the user interface which gives the user an approximate estimate of a single system's resource utilization would aid the master user's task of load balancing – balancing the processing load shouldered by each computer within the system. The most difficult part of adding this feature to the system would be gaining access to system information on resource utilization at the C++ client level. The task would be further complicated by the workings of the various Microsoft operating systems. While the C++ client software runs equally well on Windows '98, Windows NT and Windows 2000 platforms, access to system information is not handled identically on those three operating systems. Still, it should be relatively straightforward to give the user information on each system computer's processor speed, memory capacity and the relative utilization of both.

I.1.3 Automated Load Balancing on Computers in the System

In its present form, the system relies on a human user to balance the use of computer processing resources on the system. Even after features such as network latency information and computer resources information are added to the system, the system's master user will still have the task of determining which Filters to load on each computer within the system. One possible future enhancement to the system would be to design a software add-on that would determine the most optimal way to balance the load on the system's computers and would then shift Filters from one C++ client to another to re-balance the loads on the system's computers.

This type of feature is not essential to the proper running of a system, but it would serve to improve the performance of the system. Its main purpose would be to simplify the work of the system's master user, allowing that user to concentrate on designing a system that performs the actions he/she requires of it, and leaving the task of balancing the load

to computer software, probably located in the Applet's code. This software would also allow the user to quickly take advantage of any new C++ client that joined the system after he/she designed the system's Filter Graph.

I.1.4 Detection of Unresponsive C++ Clients

While the JavaServer is currently able to contend with dataflow issues and system records in the event that a C++ client disconnects from the system, there is no provision in the system currently to detect abrupt C++ client disconnections, where the C++ client does not inform the JavaServer that it is disconnecting. This type of disconnection can occur during to system shutdowns on the local computer, or a cut to the C++ client computer's power due to a power outage. It can also occur if the C++ client is terminated from the Windows Task Manager. Further, even in cases where the C++ client does not actually shutdown, it is theoretically possible that the C++ client to become unresponsive due to an unforeseen transmission error.

In these cases, it would be useful for the JavaServer to keep track of all client connections – both C++ clients and Applet connections – and periodically verify that they are functioning fine. One way of doing this would be to spawn another thread in the JavaServer, whose sole responsibility would be to periodically check on each connected system, by sending a request to those systems to respond that they are still alive. By using a separate thread rather than building each client handler thread within the JavaServer to perform a periodic check, the JavaServer is also to determine if one of its client handler threads is misbehaving due to an improper message from the client it is handling.

Upon detection of an unresponsive C++ client or an unresponsive client handler thread within the JavaServer, the JavaServer could simply shutdown the connection to that client, using its currently existing ability to remove a client connection from its listing. Alternatively, it could attempt to re-establish the connection, though that might not be possible, especially in the case of a power failure affecting one or more computers running C++ clients.

I.1.5 Quick Recovery in Response to Downed Computer Connections (Automated or User-Controlled)

At present, the system is designed with C++ clients disconnecting by simply excising all records of that client's Filter Graph and registry, and overwriting the data (see section 4.2 for further details). Unfortunately, this means that in the case where a C++ client disconnects from the system, all information, including the Filters loaded on that system and the interconnections between those Filters and other Filter Graphs would be totally lost. This leaves the system's master user with the responsibility of reconfiguring the Filter Graphs on the remaining C++ clients, loading Filters onto them to replace the Filters that were loaded on the disconnected system. Without constant, or at least regular, supervision, an accidental disconnection of a C++ client may alter the system Filter Graph to the point that the system will no longer be processing data in a manner that is useful to the master user who originally set up the Filter Graph. This means that letting the system process information overnight without periodic supervision will be risky. While the intention of this system is to test algorithms with human supervision, it must be noted that the ability of the system to correct for foreseeable problems and thus allow it to function alone without human supervision would be a welcome improvement.

In order to accomplish this, the JavaServer would need to be slightly reprogrammed to select one C++ client that would host all the Filters from the disconnecting C++ client. Once that was done, the JavaServer would simply need to issue commands to that C++ client to load Filters and connect them up to each other, before engaging the current client-shutdown recovery code.

Another feature that could be built upon this would be the ability to allow the user to select the C++ client(s) where the Filters would be reloaded. Rather than simply allowing the JavaServer to select one C++ client on which to load all of the Filters that were on the disconnecting C++ client, this would give the user an extra measure of control.

Both of these features would be easy to implement once the system's user interface has in place the ability to move a Filter from one C++ client's Filter Graph to another. This feature is discussed in section I.2.4 in this Appendix.

I.1.6 Saving and Reloading System Logs and the System Filter Graph for a Quick Restart of the System

One feature that is bound to be important to researchers hoping to use the system to conduct vision research is the ability to save a Filter Graph configuration on the system. Rather than rebuild the desired Filter Graph every time a user wants to test out algorithms in design stages, a user would be able to save the set up to a file and then call up the file at a later time to restore the system to its desired configuration. The most logical place for this information to be stored would be on the JavaServer's computer. Once the JavaServer has the ability, described in section I.1.5 above, to restore/reload Filters from a disconnected C++ client to the rest of the system, restoring the system from saved files should be quite simple.

I.1.7 Video Compression for Inter-System Transmissions and Display

As discussed in section 4.1.4, currently, the system's transmission Filters, used for inter-system transmissions, can transmit either raw video frames or video that has been compressed using the Foveal compression defined in section 3.4.2. The same is true of the display broadcast. Inter-system transmissions are transmissions of data between two C++ clients on the system. Unlike upwards and downwards data transmissions, which are relatively infrequent and rarely amount to more than a few kilobytes during user login, and perhaps as high as a single kilobyte per logged in user during standard system operation, these transmissions are ongoing and very large. Transmission of a single stream of raw video data, at 320x240 resolution, with a 24-bit colour depth, amounts to 225 kilobytes per frame. For a video stream produced by a Logitech QuickCam operating at a modest 4 to 6 frames per second, this can amount to 1 Megabytes per second.

Transmissions of Foveal-compressed streams are not quite so costly. The same stream compressed using the Foveal compression uses up 15.2 kilobytes per frame or 61-91 kilobytes per second at 4-6 frames per second. Of course, as discussed in section 3.4.2, the Foveal compression is a lossy compression, and it is assumed that the users of this system would not want the system to use the Foveal compression for inter-system transmissions unless the user had already compressed the data using the Foveal compression.

The same issues are true for the display. When displaying a the results at a node within the system's Filter Graph, the display Filter would be transmitting 225 kilobytes per frame per display viewer for raw video or 15.2 kilobytes per frame per user for display resolution in the example above. Despite the limitation on the number of Display clients the system will allow the display Filter to connect to (see sections 4.1.4 and 4.4.1), this could cause severe reductions in the frame rate for the Filter Graph on a given C++ client in the system, since the Filter Graph will have to wait for these data transmission Filters to complete the task of transmitting before picking up a new frame from the source.

One possible solution to this that has been examined is the use of video compression Filters to compress the video data and reduce the need for transmission bandwidth. The basic requirements for the compression Filters are:

(a) widespread availability of the chosen compression/decompression Filter software and

(b) header information either within each frame or a fixed size packet (such as the Foveal compression)

Widespread availability of the compression and decompression software is important. If the compression algorithm selected to replace transmissions of standard raw video is not available on most Windows computer systems, then each computer system that runs a C++ client will need to have the software installed and registered. Also, some compression formats that are very well designed for information interchange, such as Microsoft's ASF format, are not open formats, and so while the decompression software

is available, the compression software is not, and neither is information on the file format needed to replicate the compression software. Further, if the decompression Filter software of the selection compression format is not widely available, then any user who wants to use the display Filter will need to install the decompression Filter in the system registry of his/her computer system in order to view the data.

Header information is also important. Unless the Filter at the receiving end knows how large the next incoming frame will be, it will not be able to determine where one received frame ends and the next one begins. Some compression schemes, such as AVI, include header information at the start of a file. Unfortunately, these compression schemes require the header to include the entire file size, and thus, are not useful when attempting to transmit a stream of indeterminate total length.

Compressing video streams prior to transmission is also not without its drawbacks. Compression algorithms are often very computationally expensive. Running one or more instances of a video compression Filter can tax a computer's resources. Using compression algorithms will significantly improve the performance of the system if all of the C++ clients are being run on powerful computers with low transmission rates (such as computers that use standard modems to connect to the network). However, for less powerful computers with higher available transmission bandwidth, the use of compression Filters for inter-system transmissions becomes less important, which is why this feature was not considered a necessity in building the system.

I.1.8 Linux/Unix Extension

There has been some interest expressed by fellow researchers in designing software to allow computers running the Linux operating system to be used in the system. Currently, the C++ clients can only be run on microcomputers that run a Microsoft operating system, and have Microsoft's DirectShow drivers installed.

A Linux version of the C++ client would be on a totally different order than the current Windows implementation. Without access to DirectShow or even COM/DCOM, the software designed for the Linux client would have to be written from scratch in most cases, and handling of handshaking and information interchange between processing block would have to be completely the responsibility of the Linux client. Also, without DirectShow drivers, each processing block would have to be designed such that it would appear to the system as the Filters appear under DirectShow – i.e. a processing block, with input and output entry points and a point to query for information functioning in a manner similar to the DirectShow interfaces.

Under a Linux client, each processing block would likely be a stand-alone program, unlike the DirectShow Filters, which cannot be run without a COM environment. The Linux client would have access to a listing, by directory, of available processing blocks, and be able to run and control those programs as processes. It would also need to be able to make interconnections between processes using Unix pipes, instruct the processing blocks on such things as buffer allocation, gather information from the processes and handle the dataflow Still, the idea of a Linux client is intriguing, though it would take thousands of man-hours to complete and be proven reliable in testing.

I.2 Future Improvements to the Applet/User Interface

Among the features that could be foreseeably added to the system are several enhancements to the systems user interface.

I.2.1 Data Type Information

When the system was originally designed, it was designed with this feature in mind. As mentioned in section 4.2, at present, the information about the input and output data types for every Filter are stored in both the JavaServer and the Applets. While this information is not essential to a user, it would be convenient to have, and since the information is

already available, all that remains is to construct a viewing panel in the Applet to display this information.

I.2.2 Filter Property Pages

The system that has been described here is patterned after GraphEdit. One of the features available in GraphEdit is the ability to set Filter properties at runtime, often while a Filter is running. There are several ways to access Filters, but property pages are usually the best choice. A system user does not need to know much about a Filter to set is property page values, which is not the case for passing messages to Filter through files. Further, any attempt to pass messages to a Filter from the system's user interface would require information about that Filter to be stored within either the JavaServer or the Applet's code. None of this is necessary if read/write access to Filter property pages is incorporated into the system. Though DirectShow is not designed to transmit property page layout information to remote computers, DirectShow does contain COM interfaces that will allow the local software, in this case the C++ clients, to access information about the Filter properties that can be modified in a given Filter's property page(s) and then transmit that information upwards to user interface. The user interface can also be modified to provide a suitable view of the Filter properties that can be modified by a given Filter's property page(s).

I.2.3 Chat Feature

The chat feature seems, at first glance, to be the least needed feature for a system designed to test computer vision systems. However, this feature might actually prove to be more important to the smooth operation of the system than most of the other features described in this chapter. The system is designed to allow multiple users to access it, view the Filter Graphs on each individual computer within the system, view the displayed node set by the master user and receive information from the system. The users access the system through a web-based interface that grants them access from any location around the world. As well, the user interface does not provide users with any information about the other users logged in to the system. The only information available to users that can

help ascertain whether or not others are logged on to the system is the part of the user interface that keeps track of whether the applet is in master or view only mode. Thus, there is absolutely no reason to expect that the users will be able to contact each other in the event that one user logged in without master privileges, requires master privileges on the system to run an experiment.

At present, the system has no hierarchy (see section I.4.2) to allow a more senior/more important user to take precedence over a less important user and replace a user of lesser ranking as the system's master user. As well, at present there is only a single username accepted on the system (see section I.4.1). Even if the system were modified to accept several different usernames and a user hierarchy, there is still the possibility that a lesser-ranked user might need access to the system. The simplest way to this would be to build a simple chat function into the system. The chat feature would allow such a user to contact other users on the system and request access. The chat feature would be simple and would not need to have the ability to direct messages at only a single user. When a user would send a chat message to the other users, they would get a message in their log box informing them that a new chat message had arrived. They could then switch their applets to Chat view – selectable from the left tab like all the other views – and read the message.

The chat feature could also be helpful in another way. It would allow multiple users to have a chat-session conference discussing the results of an algorithm being tested by the master user.

I.2.4 Ability to Move Filters from Machine to Machine

Sometimes, to balance the load on the computers in the system, the master user may decide that one or more Filters should be moved from one C++ client to another one. Generally, the Filters available on any one C++ client are available on all C++ clients within the system, so the issue of the availability of the Filter on its new C++ client host is not as relevant (and it is addressed in section I.2.5). Rather than forcing the master user to go to the trouble of first removing a Filter from one C++ client, then loading the same

Filter on another C++ client and then re-establishing the connections between that Filter and the other Filters it used to be connected to, it might be simpler to just give the system the ability to shift a Filter from one system to another. Once that's done, the user interface could be modified to grant the user the ability to select a Filter and move it to another C++ client within the system. As noted in section I.1.5, the ability to shift a Filter from one C++ client to another will have other positive implications as well.

I.2.5 Downloading and Registering Filters to C++ Clients

As noted in section I.2.4, generally, the Filters available on any one C++ client are available on all C++ clients within the system. This is not always the case. One feature that was considered for this system early on was the ability to download a Filter to a C++ client and have the client register the Filter and then update it registry listing. This would ensure that any processing block could be placed and run on any computer within the system and would allow the master user to concentrate on other matters than optimizing the system given then location where the desired Filters are registered. This would also allow a researcher to develop parts of an algorithm locally using local DirectShow software, such as GraphEdit, and then test the full algorithm by downloading the Filters he/she had designed to C++ clients on the system, make connections and then test the Filters together. This would save the researcher the time and trouble of manually copying/uploading and registering Filters onto the computers running the C++ clients on the system.

In order to accomplish this, the user interface would have to be redesigned to display a full list of all the Filters available on the C++ clients on the system, rather than the current system of displaying only the Filters registered on the C++ client that is the current system focus. If a Filter selected to be added to a C++ client's Filter Graph by the user did not exist in the registry of that C++ client's computer, then the Filter would be downloaded from another C++ client either through the JavaServer or through an FTP connection.

I.2.6 Smart Accessing of Filters

One alternative to allowing Filters to be uploaded from one C++ client and downloaded to another would be a method of loading a Filter on the C++ client where it resides, and setting up connections to it all with a single command issued from the applet. In general, this alternative is less preferable than allowing the system to download Filters. However, in the case where a researcher wants to grant others access to his/her software without allowing his/her software to be downloaded by others, this method might be preferable. As with the previous section, here the user interface would need to be redesigned to inform the user of all the Filters available on all the C++ clients on the system.

It might actually be possible to combine this idea with that of downloading Filters, if a flag could be set within the JavaServer identifying some Filters are "do not download" Filters that must be run from their native C++ client.

I.3 Future Improvements to the Display System

I.3.1 Publishing Streams to the Internet

In the current system, the master user can select a single node to be rendered and displayed. Once the display is set up, the user must access it using a separate displaying program. The only version available for the displaying program is a program written for Windows-based operating systems using DirectShow Filters. So, though the Applet can be viewed and the system configured from any computer with a Java-capable browser, only a user with a Windows-based personal computer and a copy of the displaying software including the necessary DirectShow Filters can make full use of the system.

For the system to achieve its goal of being fully accessible across all platforms, the display function must also be viewable from any platform. Since most browsers-equipped machines contain software to display certain multimedia formats taken from on-line sources, replacing the current system requiring a dedicated program to render the display

with a browser-based solution whereby the user would simply access a webpage for the display would seem to be in order.

Several possible solutions along these lines have been examined. One of these, likely the most ambitious, calls for the publishing of a live stream (as opposed to a pre-recorded one) to the internet, where a standard web browser will be able to access it. Another, far less ambitious solution would be to save the display information into short media files, which could then be accessed by a web browser. Both of these solutions require access to video compression software, file writing ability and the ability to transmit the data from the C++ client where it was being produced to the JavaServer's computer for publishing and redisplay. However, there are some significant differences between the two approaches.

Publishing live streams would produce a better display. The only delay in transmission would be the delay in transmitting the video data between the C++ client and the JavaServer's computer and then in retransmitting the published stream to the user's computer. This option requires the ability to compress video using compression formats that retain information on the size and resolution of a given frame or group of frames. Not all video compression formats retain this information, and the compression software for those that do are not part of the standard bank of Filters that are available on a Windows-based system, though the decompression software often is available for popular multimedia formats. This requires either writing a Filter to perform the compression of a raw video stream into a standard compressed video format or obtain such a Filter elsewhere. Of course, such software could also be used to compress data in inter-system transmissions (see section I.1.7). However, at present, DirectShow does not come equipped with a Filter designed to publish a stream of video data to the Internet or to write compressed video data, other than video compressed in the AVI format, to a file. And it should be noted, that the AVI format would not be suitable for this solution since it has a header that includes information on the entire file size and thus, an AVI-compressed file must be completely written before it can be viewed.

As for the other solution, the software is available – a standard AVI compressor is available with DirectShow – however, this solution has less desirable effects. In order to function, the system must use several files as a circular buffer, overwriting one at a time in a fixed order with new display data. The length in milliseconds of each file would be a key determinant in the delay between the capture of video data and the displaying of that data to the user. However, the total amount of video time represented by all of the files must be long enough to handle the possibility of long transmission latency between the node producing the video data an the user's browser. For example, if each AVI compressed file contains 50 ms of data, and the transmission latencies between the JavaServer's computer and the user's computer is 500 ms respectively, then "circular buffer" of compressed data files must contain at least 11 files long so that the user sees the entire AVI file before it is overwritten. Since transmission latencies may, in some cases, stretch into several seconds, the number of files necessary to contend with this possibility is very large. As well, there must be a system in place to inform the browser to begin displaying the next AVI file. Since JavaServer-to-user latency may be long this will mean that the user will only get a display half of the time, unless the timing can be set up elsewhere. The browser can be set up to automatically switch from one file to another and request the next file in time to receive it. However, this is unlikely to improve the display and there will still be delays as the user's computer sets up for each new display. Lengthening the AVI files will reduce the number of delays, but it will also increase the delay between video capture and video display.

Both of these solutions would simplify the work of the user to view video data at the displayed node. However, since the emphasis of this thesis' work is on harnessing the processing power of the distributed computing resources available, and since the current display fits the thesis parameters of allowing the user to investigate the processing at any node in the system (and with lower transmission delay than either of the two solutions presented here, I might add) the work of publishing the displayed stream to the Internet will have to remain for future implementation.

I.4 Future Improvements to Applet – JavaServer interaction

I.4.1 List of Valid Usernames/Passwords for Logging into the System

At present, the system is hard-coded with accepted usernames/passwords. This works well when only a few researchers are accessing the system. However, as the number of users of the system increases, a proper list of users would serve to keep track of who was using the system and how often. As well, it would ensure that only authorized users would log onto the system, and that users no longer authorized to use the system would be prevented from doing so.

I.4.2 User Hierarchy

Continuing along these lines, a hierarchy of users would also be beneficial. Such a system would grant only viewing privileges to some users, and then rank the other users' priority on the system. If a more important user than the current master logs onto the system, the system would suspend the current master's master privileges and transfer them to the higher-ranking user. This way a super-user identity could be set up to deal with the system in the event of error in the JavaServer thread that was handling the previous master.

I.5 Future Improvements to upward/downward data transmission security

I.5.1 Varying Authentication Tags

The current transmission protocols include an authentication tag that is meant to authenticate the message's sender. Since addressing all of the security concerns was not part of the thesis' objectives, the authentication tags were left as simple, constant tags. To improve security, the system's components could vary these tags according to an encoding scheme. This would help prevent any tampering with upward/downward communication.

I.5.2 Message Encryption

Another measure that would aid security in upward/downward communication would be encrypting entire messages. Instead of merely using a system of varying authentication tags as described in section I.5.1, the messages themselves could be encrypted. In order for the message encryption to be effective, the form of encryption key itself should be selected such that no message gets encrypted the same way twice.