IASTED International Conference on Robotics and Manufacturing, 1998, Banff, Canada

A DISTRIBUTED, OBJECT-ORIENTED ARCHITECTURE FOR PLATFORM-INDEPENDENT MACHINE VISION

Nicholas A. Barendt
Dept. of Electrical Engineering and Applied Physics
Case Western Reserve University, Cleveland, Ohio, USA

Frank L. Merat
Dept. of Electrical Engineering and Applied Physics
Case Western Reserve University, Cleveland, Ohio, USA

Andy Podgurski
Dept. of Computer Engineering and Science
Case Western Reserve University, Cleveland, Ohio, USA

Edward Blanchard
Dept. of Electrical Engineering and Applied Physics
Case Western Reserve University, Cleveland, Ohio, USA

ABSTRACT

This paper describes the design of a client/server architecture for machine vision. The server is constructed as a virtual machine, permitting client software to be platform-independent. The client architecture consists of a number of proxy classes that hide the details of server communication, simplifying the construction of client applications. Serializable objects are used for communication between clients and the server. Both TCP/IP sockets and POSIX message queues are currently supported for client/server communication. The server is implemented under VxWorks, a real-time operating system (RTOS), executing on a Motorola processor. It currently supports hardware developed by Imaging Technology, Incorporated. Client software is implemented under VxWorks, LynxOS, another RTOS, and Linux, executing on both Motorola and Intel processors.

Keywords: Machine Vision, Distributed Systems, Open Systems, Client/Server

1 INTRODUCTION

Machine vision systems are becoming more prevalent in manufacturing environments as their associated costs continue to drop and processing speeds continue to increase. The advent of low-cost, high-performance DSPs has initiated a whole new generation of sophisticated machine vision hardware. A large number of vendors now offer turnkey machine vision systems, greatly simplifying the integration of machine vision into manufacturing.

Machine vision is an enabling technology for agile manufacturing in particular, allowing specialized hardware required for a particular component to be replaced by a camera and machine vision software. The CWRU Agile Manufacturing Workcell is such an example [1]. Specifically, agile manufacturing is "the ability to accomplish rapid changeover between the manufacture of different assemblies utilizing essentially the same workcell" [1]. Agile manufacturing often uses robotics to allow a wide variety of tasks to be performed by a single workcell. Flexible part feeding systems, for example, supply a wide variety of parts in random positions and orientations to a robot. Machine vision is used to accurately determine the pose of parts so that they can be assembled by a robot [2].

In recent years, the automation market has been moving towards "open systems" designed to allow the customer to integrate products from different vendors into a control system. Open robot controllers (e.g., Cimetrix), allow the system integrator to control a wide variety of robots with a common software interface. This is in contrast to older systems that were closed and not extensible. In particular, the software and development environments of these older systems are typically proprietary, forcing customers to learn an arcane language or Application Programming Interface (API).

This paper describes the design and development of a platform-independent software architecture for machine vision. A client/server model is used, allowing a large number of concurrent vision tasks as well as remote clients. The architecture uses object-oriented design (OOD) to encapsulate vendor specific hardware and software, allowing applications to be written using a generic machine vision API.

2 PRIOR RESEARCH EFFORTS

While there has been a great deal of research in the field of machine vision, relatively little has been done in the specific areas of platform-independence and client/server architectures. Two exceptions to this are the "DataCube Server" and the DeVious project. Kahn et al. [3] developed the "DataCube Server" to simplify the programming of machine vision hardware produced by DataCube, Inc. Their reasons for creating the server were twofold: (1) only a single process on a single "host" machine could access the expensive DataCube hardware and (2) pipelined programs are difficult to write. Its

single vendor design and narrow application domain limited the DataCube Server.

In the DeVious project [4], Romig et al. developed a distributed environment for computer vision, allowing machine vision operations to be performed in parallel using a loosely-coupled network of workstations. The motivations for this were to take advantage of the computational capacity of a network of computers and to investigate distributed systems for the specific area of machine vision. An important aspect of the DeVious project was that the software was designed to operate in a heterogeneous environment, on both Sun and Digital Unix workstations, taking advantage of accelerated hardware when available.

The current work uses many of the concepts developed for the DataCube Server and the DeVious project. Specifically, the remote method invocation (RMI) system developed for the DataCube Server inspired the one used in our architecture. The use of object-oriented programming to simplify the programming of pipelined machine vision hardware of the DataCube Server has been developed into a platform-independent virtual machine vision processor, further simplifying the programmer's model of the vision system. The DeVious project's ability to take advantage of available hardware was the inspiration for this architecture's use of a virtual vision processor. Finally, the distributed nature of both architectures guided the construction of the client/server communication.

3 DESIGN GOALS

Analysis of the machine vision needs of the CWRU Agile Manufacturing workcell led to the following design goals for this architecture:

1. To preserve the *functionality* of the AdeptVision system currently in use while providing a more flexible, extensible architecture
2. To achieve *platform-independence* permitting applications to be ported between different vendors' systems, without sacrificing performance
3. To support *multiple, local and remote concurrent machine vision tasks*, across heterogeneous platforms
4. To use *Object-Oriented Design* (OOD) techniques to facilitate development and maintenance without incurring unacceptable overhead

The impetus for embarking on the development of a new machine vision system grew out of experience with the AdeptVision [5] system. The AdeptVision System [1, 6] satisfied many of the basic requirements of the CWRU workcell, providing a basic toolset for industrial machine vision.

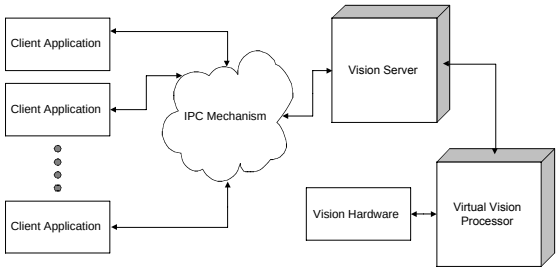

FIGURE 1: ARCHITECTURE OVERVIEW

AdeptVision supports a number of basic machine vision operations, including image acquisition, binary analysis (e.g., connected-component labeling), inspection tools (e.g., "rulers" and edge-finders), and, most-importantly, camera calibration, allowing vision-guided robot motion. Like most machine vision systems, however, AdeptVision is closed and proprietary, limiting extensibility and portability.

Our software architecture needed to support a wide range of machine vision hardware while being open and extensible. There is a great variety of machine vision hardware available from simple frame-grabbers to high-end, multiprocessor systems. The architecture should be capable of taking full advantage of accelerated hardware, without rewriting existing application software.

The CWRU Agile Manufacturing Workcell requires a number of vision tasks for operations like parts feeding. A client/server model is useful because it is a natural way of sharing limited resources among a number of tasks. In addition, the client/server model permits support for remote tasks (clients).

Object-oriented (OO) software has become the standard for many new systems. OO design principles assist the software developer in producing clean, maintainable software. C++ and other OO languages, however, tend to be less efficient than their procedural brethren (e.g., C) in both CPU and memory usage. Although modern computer systems make these efficiency problems less important, dynamic memory allocation can still be a major problem. The creation and destruction of objects in a real-time system must be done carefully, if at all. A well-designed system can eliminate the run-time use of dynamic memory through static objects and pre-allocation.

4 ARCHITECTURE

A client/server model was chosen (see Figure 1) to both promote efficient usage of limited resources as well as assist in the creation of a platform-independent system. Platform-independence was achieved by constructing the server as a virtual machine, allowing client applications to be written to a generic interface. Object-Oriented Design (OOD) was used, programming in C++, to facilitate design, development, and maintenance of the architecture.

One of the important design decisions for this architecture was how to model a generic machine vision system. One way to model a vision system is through its hardware: frame grabbers, arithmetic-logic units (ALUs), frame buffers, cameras, etc. This approach is attractive because it models the system from the bottom up, starting from the hardware. Another view is that the model should simplify the system to its most primitive elements: sensors, images, and operations on images. This abstract view of the system is also attractive because it reduces the perceived functionality of a system to its core. This architecture employs both models since both are useful under certain circumstances.

4.1 SERVER

The vision server's architecture is based upon a hardware-subsystem model of a machine vision system which models the server as a collection of hardware components such as frame grabbers, cameras, and frame buffers. This allows the major subsystems that are found in a typical machine vision system to be encapsulated in object-oriented classes and results in a straightforward OO model of this system. The server also includes a number of classes for managing system resources and communicating with clients.

The server is designed as a virtual machine responsible for efficiently allocating limited machine vision resources between a large number of concurrent tasks (clients). The server's responsibilities include: communicating with clients, allocating frame buffer resources, acquiring images, and performing machine vision operations using the hardware in the system.

To support the increasingly heterogeneous controller environments found in industry, a distributed architecture is important. Unlike other distributed machine vision systems [4] this architecture's distributed nature is designed for flexibility, not for the parallelization of a single task on multiple computers in an effort to decrease execution time. Distributed in this case means that remote clients may use the operations supported by the server.

The server is made up of several component classes

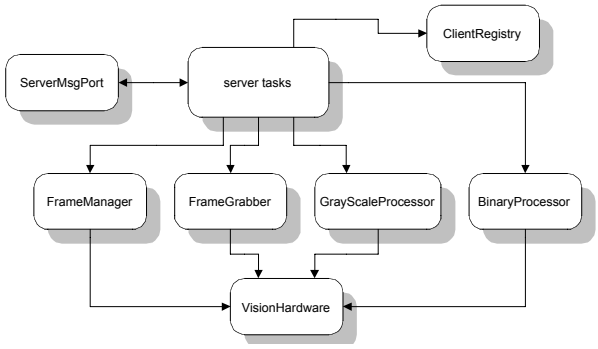

FIGURE 2: SERVER ARCHITECTURE

(see Figure 2). A number of these classes perform machine vision operations, using the services of the virtual vision processor, encapsulated in the VisionHardware class. Clients do not deal directly with the VisionHardware class, but rather with intermediary classes (e.g., FrameManager, FrameGrabber, GrayscaleProcessor, and BinaryProcessor). The client interface and communications mechanisms are encapsulated in the intermediary classes, simplifying the implementation of the VisionHardware class. The VisionHardware class, in turn, encapsulates all machine vision operations implementing them in hardware or in software, depending on the available resources.

VisionServer Class

A composite class, VisionServer, contains all of the required components necessary to provide services to clients. The VisionServer constructor is responsible for creating all server objects (i.e. ClientRegistry, FrameManager, FrameGrabber, BinaryProcessor, GrayscaleProcessor, and VisionHardware) and spawning the server tasks.

Vision Server Tasks

The vision server employs a number of threads to listen for client requests. The vision server tasks are responsible for accepting client requests, binding the request to the correct server object, invoking and executing the requested method, and returning any results to the client. For flexibility the architecture uses a generic datagram communication which may be implemented using many different types of inter-process communication (IPC), including POSIX message queues and TCP/IP sockets. The vision server currently uses an iterative server design [7, 8], although it would not be difficult to adapt it to a concurrent design.

The ClientRegistry

As its name suggests, the ClientRegistry is responsible for maintaining registration information for all active clients of the server. To reduce the amount of communications between clients and the server, the server maintains state information about all active (registered) clients. This state information decreases the amount of data the client needs to send to the server, particularly for often used parameters (e.g., binary threshold values).

The FrameManager

A frame is a structure that contains image data (i.e. pixels). The FrameManager object is responsible for the allocation of frame buffers to clients. This includes physical frame buffers, located on frame grabbers, as well as RAM on the host machine that has been allocated for image storage. In order to provide flexibility for application developers and promote efficient resource usage, the architecture allows frames to be dynamically

attached and detached (i.e. reserved and released). Various types of frames are available: 1-bit, 8-bit, and graphical display in the current implementation.

The FrameGrabber

The FrameGrabber is an abstract representation of a physical frame grabber with few operations. It converts incoming video data from a camera into quantized pixel values in a discrete array. The FrameGrabber class allows clients to acquire images into frames.

The Grayscale and Binary Processors

After an image has been acquired, it may be processed using the methods of the GrayscaleProcessor or the BinaryProcessor. These classes are responsible for performing all image processing and machine vision operations in the system. The GrayscaleProcessor handles all grayscale images, including thresholding operations, which produce binary images. The BinaryProcessor is responsible for all operations on binary images which includes connected-component labeling and "blob" analysis. Only a limited set of these methods has been implemented at to date. In future implementations, these two classes will provide the majority of the extensibility of the system, allowing new machine vision algorithms and operations to be added to the architecture.

VisionHardware Class

One of the tenets of this architecture is that a virtual vision processor can be constructed, along the lines of the Java Virtual Machine (VM). This allows applications to be ported to new platforms easily since only the VM needs to be ported. The VM in this architecture is the VisionHardware class, which represents a virtual vision processor, providing an interface for common machine vision operations. This class is an abstract class, subclasses of which provide implementations suitable for particular hardware configurations.

4.2 CLIENT

The client architecture consists of the software required to build applications for use with the server architecture, including a library containing machine vision and communication classes. The design of the client API is based upon an abstract functionality model of a vision system that has two major classes of objects: cameras and images. A camera is used to acquire data from the real world creating an image stored in a computer. Operations can then be performed on the image to extract useful information.

The client architecture (see Figure 3) contains classes to perform machine vision operations using the services of the vision server. The API is designed to be intuitive, making application building straightforward. The most important tool for providing a simple and intuitive interface in the client architecture is the Proxy design pattern [9]. This design pattern permits the details of the client/server communication to be hidden, simplifying the job of the application developer.

Camera Proxies

Cameras acquire image data into frame buffers. Clients deal with camera proxies that support the same operations as the physical cameras in the system. The CameraProxy class takes care of all client/server communication, including parameter marshalling and remote method invocation. For instance, the *acquire_still()* method performs the operations necessary to acquire a single image from a camera associated with the camera proxy into a specified frame buffer.

Frame Proxies

Frame proxies provide a client representation of a frame that exists within the server. They provide all of the services required in building machine vision applications, from copy operations through connected-component labeling to inspection tools. Frames proxies may dynamically attach/detach to/from server frames. Deadlocks are prevented using a form of Two-Phase Locking [10].

4.3 COMMUNICATION ARCHITECTURE

A communication class, MsgPort, encapsulates the details of particular IPC mechanisms through its *send()* and *receive()* methods. Proxy classes in the client architecture use Command classes to send requests to the server. An enumerated method type in the Command header specifies the method requested, allowing the server to quickly bind the request to the appropriate object. The command classes are serialized, sent through the MessagePort, and received by the server. Using the enumerated method type, the server queries each of the main server objects (i.e. ClientRegistry, FrameManager, FrameGrabber, GrayscaleProcessor, and BinaryProcessor) to see which object the requested method belongs to. It then invokes the requested method, using a standard interface provided by each of the server objects.

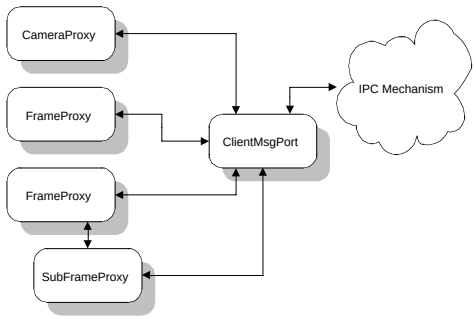

FIGURE 3: CLIENT ARCHITECTURE

Abstract Socket (MessagePort)Class

The Abstract Socket class is a pure abstract base class which provides *send()* and *receive()* methods for the transmission of datagrams. Subclasses of this class provide concrete implementations of these functions for particular communication, such as TCP/IP sockets [11], shared memory, or POSIX message queues [12]. The advantage of using a pure abstract base class is that a number of different communication mechanisms can be supported with no modifications to client software.

Command Class

The Command class is a design pattern [9] that encapsulates a request as an object. A Command class, MsgPacket, is used in this architecture to encapsulate requests and replies between the clients and server. Clients specify the requested operation to a server by an enumerated type. These Command classes have serialization methods (which use XDR [13]), allowing their data state to be captured, transmitted, and reconstituted in another process space or on another machine, independent of the machines architectures (e.g., big-endian vs. little-endian).

Remote Method Invocation

To support distributed processing, tasks on one machine need a mechanism for invoking operations on other machines. A client in a client/server application needs to invoke functions residing on a server. Remote Procedure Calls (RPC) are one way of doing this, making a server request look like a function invocation.

The object-oriented analogy of RPC is Remote Method Invocation (RMI). The RMI system used for this design makes heavy use of the Command class (Section 0) to encapsulate the request and binding operations. A similar remote method invocation system was used with the DataCube Server by Kahn et al. [3].

5 RESULTS

The architecture's infrastructure has been designed, implemented, and tested. The client architecture provides a full interface for the capabilities of server. The server currently contains operations for basic image processing, and includes two implementations of the virtual vision processor class, VisionHardware: Itex15040 and SoftwareVision. Itex15040 provides the hardware specific software required in using the Imaging Technology 150/40 VME-IMA vision board. The SoftwareVision class emulates vision hardware in software, providing the same results as the Imaging Technology hardware, albeit much slower.

The minimum round-trip client/server transaction time for the POSIX Message Queues and TCP/IP sockets was approximately 0.4ms and 2.7ms, respectively, for local clients. A number of experiments involving a large number of clients [14] demonstrated that the overhead of the architecture for local POSIX Message Queue clients was very small and that the maximum latency for remote TCP/IP socket communication varies with network load but the average is fairly constant.

6 CONCLUSIONS AND FUTURE WORK

The architecture has demonstrated its distributed nature by porting client applications to several UNIX platforms (e.g. VxWorks, SunOS, and Linux). Performance has not suffered greatly with the introduction of several layers of abstraction (i.e. client proxies, server, abstract vision hardware classes): typical operations incurred a small amount of overhead, on the order of a few milliseconds. This can be attributed to the work involved in serialization/deserialization of Command objects, IPC, and binding requests to the appropriate object. For many applications, this is not too large a performance hit given the flexibility and portability that the architecture provides.

For this architecture to be useful for more than trivial processing, a number of operations still need to be implemented, including connected-component labeling, geometrical feature extraction (i.e. blob analysis), and basic inspection tools (e.g. line and arc finding). Once these tools are in place, higher-level operations required for a robotic workcell, such as camera calibration, can be constructed.

In the future, we plan to port the system to new vision hardware to test the architecture's flexibility. Without this acid test, it will be difficult to judge the success of the virtual vision processor model. The application of the architecture to other problem domains besides agile manufacturing will be necessary to determine its generality.

The inclusion of a "hardware" thread within subclasses of the VisionHardware class could also improve performance. This thread would exist within the VisionHardware object and would be tailored to particular vision hardware. Clients of the VisionHardware class need to know nothing about this thread; the thread is hidden within the class's implementation. The classes public interface would queue up requests to the vision object instead of directly invoking operations (i.e. a proxy). These requests could then be scheduled by the "hardware" thread to take advantage of the concurrency supported by the vision hardware. In multiprocessor systems, it may be useful to have a number of "hardware" threads, even with a single vision board, so that even the operations that have no hardware support can be executed in parallel.

7 ACKNOWLEDGEMENTS

This work was funded through the generous support of the Center for Automation and Intelligent Systems

Research (CAISR), the Cleveland Advanced Manufacturing Program (CAMP), and industrial sponsors.

REFERENCES

1. Quinn, R.D., *et al. Design of an Agile Manufacturing Workcell for Light Mechanical Applications*. In *IEEE International Conference on Robotics and Automation*. 1996. Minneapolis, MN: IEEE. p. 858-863.
2. Causey, G.C., *et al. Design of a Flexible Parts Feeding System*. In *IEEE International Conference on Robotics and Automation*. 1997. Albuquerque, NM: IEEE. p. 1235-1240.
3. Kahn, R.E., M.J. Swain, and R.J. Firby, *The DataCube Server*, 1993, University of Chicago: Chicago.
4. Romig, P.R. and A. Samal, *DeViouS: A Distributed Environment for Computer Vision*. Software-Practice and Experience, 1995. **25**(1): p. 23-45.
5. Adept Technology Incorporated, *AdeptVision VME Reference Guide*. Version 11.0 ed. 1993.
6. Merat, F.L., *et al. Advances in Agile Manufacturing*. In *IEEE International Conference on Robotics and Automation*. 1997. Albuquerque, NM: IEEE. p. 1216-1222.
7. Nichols, B., D. Buttlar, and J.P. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. 1996, Sebastopol, CA: O'Reilly and Associates.
8. Stevens, W.R., *UNIX Network Programming*. Second ed. Vol. 1 Networking APIs: Sockets and XTI. 1997, Upper Saddle River, NJ: Prentice-Hall.
9. Gamma, E., *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994, Reading, Massachusetts: Addison-Wesley.
10. Tannenbaum, A.S., *Modern Operating Systems*. 1992, Upper Saddle River, NJ: Prentice-Hall.
11. Stevens, W.R., *UNIX Network Programming*. First ed. 1989, Englewood Cliffs, NJ: Prentice-Hall.
12. Gallmeister, B.O., *POSIX.4: Programming for the Real World*. 1995, Sebastopol, CA: O'Reilly & Associates, Inc.
13. Sun Microsystems Incorporated, *XDR: External Data Representation Standard*, 1987.
14. Barendt, N.A., *A Distributed, Object-Oriented Software Architecture for Platform-Independent Machine Vision*, in *Department of Electrical Engineering and Applied Physics*1998, Case Western Reserve University: Cleveland, Ohio.