HIVE: A DISTRIBUTED SYSTEM FOR VISION PROCESSING

Amir Afrah, Gregor Miller, Donovan Parks, Matthias Finke and Sidney Fels

Human Communication Technology Laboratory
University of British Columbia, Vancouver, BC, Canada

ABSTRACT

We have built a novel vision processing system architecture called Hive. Hive fills a gap in the vision middleware by providing mechanisms for simple setup and configuration of distributed vision computation. Hive facilitates communication between independent cross-platform modules via an extensible protocol, allowing these distributed modules to form a vision processing pipeline. A plug-in interface allows general software to be represented as Hive modules: e.g. drivers for hardware devices such as cameras or implementations of particular vision algorithms. The modules are set up as a peer-to-peer network which allows for automated data transfer, callbacks and synchronization. We describe the architecture, communication protocol, plug-in interface and control system for the modules. A distributed face tracking system demonstrates the simplicity and flexibility for creating complex distributed vision applications using Hive.

Index Terms— Computer Vision, Distributed Systems, Middleware, Architecture

1. INTRODUCTION

In recent years, there has been much activity from the vision processing community on creating advanced algorithms for video analysis in applications such as surveillance, industrial vision and entertainment. These algorithms aim to find precise and efficient methods for extracting meta-data from streams of images, based on a set of criteria. The complexity of dealing with the variability and volume of vision data often leads to algorithms that are quite computationally expensive. The increasing progress in this field has encouraged a continued need for vision system development in order to prototype and evaluate algorithms. Since system development is usually not the foremost concern when developing vision applications, proof of concept systems are often developed in an ad-hoc fashion resulting in systems that are application specific and generally not robust. The lack of a standard application programming interface (API) for vision system development is forcing vision researchers and developers to tackle many system development issues prior to focusing on their primary goal.

The three major issues facing vision system developers can be categorized as follows. First, the lack of abstractions for dealing with physical devices such as cameras, files and processing platforms leads to significant low-level development efforts. For example, even simple tasks such as obtaining images from a camera are not trivial primarily due to the large variability of video data sources. This issue is further complicated when dealing with multiple heterogeneous sources and systems that require source-device communication.

Second, the continued need for increasing performance is not met by general purpose processors. Many vision algorithms can be executed on massively parallel processing systems which current CPUs do not support. Although very efficient at executing sequential code, in image processing applications where the volume of data is very high and parallelism can be effectively exploited, the performance of sequential CPUs quickly degrade and other parallel, distributed platforms such as FPGAs and GPUs have significant advantages [1, 2]. Unfortunately, taking advantage of these more effective platforms requires significant development effort.

Third, the lack of standardization and modularity of code (platform dependency and connectivity issues) results in significant re-development effort. Due to the lack of a uniform framework much of vision algorithm development is project specific, providing no simple method for re-use. These limitations have had a great negative impact on the vision processing community by increasing development time and limiting unified development.

Taking these problems into account, we have designed Hive based on a layered architecture that addresses the necessary abstraction, performance and standardization requirements of vision system development. Hive's architecture allows for creating a solution that is scalable and portable. Figure 1 shows a simple application using Hive. The camera is connected to the network using a Hive interface on the source drone (or module), which is subsequently tied to the processor drone to compute results from the image data and send them back to the controlling application.

2. RELATED WORK

Vision system development issues have been identified and addressed for a number of years. There has been efforts to

978-1-4244-2665-2/08/$25.00

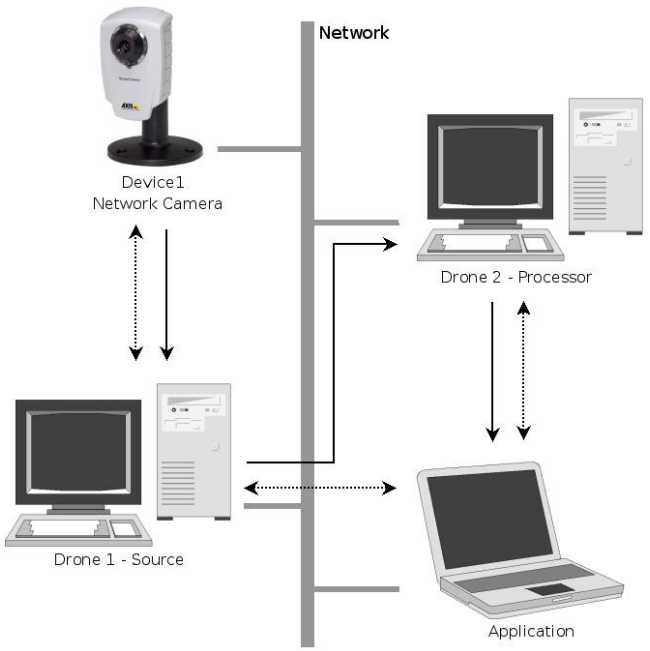

Fig. 1. This is an example swarm set up using Hive to accomplish a vision processing task. The network camera is interfaced to Hive using Drone 1 (the source), which is connected to Drone 2 (the processor) by the application. The application then collects the results from Drone 2.

standardize image acquisition such as Video4Linux and IIDC for IEEE 1394 cameras (FireWire)[3, 4]. Video4Linux provides standardized access to video devices by including an API in the kernel, but this solution is inherently platform specific. Solutions such as IIDC have been successful in standardizing a specific subset of camera devices that use a particular bus (i.e. firewire) however it has yet to spread to the vast majority of other video sources such as network cameras, capture cards, USB cameras, and pre-recorded video.

Open Source Computer Vision (OpenCV) is a set of libraries that provide a function based approach for tasks and algorithms in computer vision[5]. OpenCV provides easy methods for accessing image files and higher level algorithms however it does not support issues involved in building vision systems i.e. dealing with source details or means for distribution etc. Implementation of Hive modules could rely on OpenCV for functionality.

Generally speaking, improvement in processing efficiency has been addressed by increasing parallelism. Various hardware acceleration methods have been proposed to exploit parallelism in vision algorithms. Solutions such as programmable hardware (FPGAs), GPUs and multi-core systems can lead to great performance boosts at the cost of significant development effort to use them effectively. This inhibits adoption by general vision researchers and developers. There exists a significant trade-off between development time, resources and expertise that determines performance gained.

Distributed systems have also been used for increasing computational performance. Existing libraries and standards for distributing general processing tasks include Message Passing Interface (MPI) and Common Object Request Broker Architecture (CORBA)[6, 7]. MPI provides abstractions for distributing processing over a cluster of computers by explicitly controlling task division and timing. CORBA provides a method for remote function calls via a client-server model. Although these standards provide suitable mechanisms for distribution of vision systems, they are overly complicated and do not directly support vision systems.

CoreImage and CoreVideo by Apple Inc. provide a plugin-based architecture for image processing that utilizes graphics cards for hardware acceleration. This architecture provides a set of low-level filters and operations that can be easily accelerated using the GPU by aggregating their operations on video streams. However, the CoreImage/CoreVideo do not support flexibility for different implementation networks and different hardware acceleration methods.

Jovanovic et al. present a middleware for distributed smart cameras focusing on a policy based approach for reconfiguring a network of smart cameras for a variety of different applications[8]. Detmold et al. present a middleware for distributed video survallance that manages much issues particular to survalance and large arrays of smart cameras[9]. These middleware focus on particular applications and devices (homogeneous smart cameras) without addressing more diverse applications requiring variety of different devices and sensors which is necessary for researchers and developers of smaller scale projects.

Aritas et al. present RPV, a programming environment for real-time parallel vision processing. RPV creates a programming model targeting the issues of managing data flow and its corresponding processing[10]. RPV provides a method for data gathering, pipelining, data and function parallel processing with each processing module receiving, processing and sending data. Although RPV does address the issue of basic control flow of data, it assumes a homogeneous cluster of processors and it fails to address data source and processor management as well as more complicated data flow and non-uniform synchronization.

There has been efforts toward creating systems targeting similar issues in robotics and haptics research. RTPM YARP and Player are two frameworks that support distributed processing for controling robots [11, 12]. YARP is an open source set of platform independent libraries and protocols that facilitate communication between different modules. YARP focuses on a communication methodology that follows P2P fashion, however it does not support many of the specific vision system requirements such as uniform access and configuration of service modules. Player defines standard protocols and provides a multi-threaded framework that facilitates

communication between multiple devices and clients. This architecture fits well with sensors and control routines for robotics however a more flexible communication paradigm is required for vision processing that allows for different distribution and synchronization methods. RTPM is an architecture for distributed realtime collaborative haptic applications [13]. RTPM provides a middleware for development of realtime haptics networks using remote procedure call based communication protocl. This system is similar to Hive however based on a much more focused client based network model.

Layered architectures have successfully provided abstraction and modularity of services applied to a variety of problems. Perhaps the most successful and well known layered architecture is the Open Systems Interconnection Basic Reference Model (OSI model) that has standardized the TCP/IP network protocol[14]. The OSI model abstracts various levels of service required in networking into seven layers with standard interfaces. This architecture allows programmers to develope portable modules that use networking services at various layers. We take inspiration from this architecture to abstract various layers of functionality for distributed, parallel video processing.

We recently became aware of another distributed system using the name Hive[15]. We believe this does not pose a problem since our version is specifically designed for vision processing systems.

3. HIVE ARCHITECTURE

Hive has been developed to address a gap in current computer vision middleware, as there exists no simple system for distributed processing and development of reusable modules. Specifically, the architecture of Hive has been designed to accommodate a number of requirements:

- *Abstraction and Encapsulation*: the low-level functionality of the system should be hidden from the application developer. This feature also allows for different implementation approaches to accelerate performance.
- *Plug-in interface*: the interface to Hive from low-level device implementation to application development is kept simple without sacrificing flexibility and power.
- *Flexible communication*: the communication protocol is extensible, flexible, and contains mechanisms for synchronization between modules and automatic data transfer.
- *Cross-platform*: Hive is intended to work on multiple platforms, so basing the system on a layered architecture allows for easier transfer to different systems.

A Hive system consists of a number of *drones* which are connected together into one or more *swarms* by an *application*. The term drone is used to describe a device or service which uses Hive for communication and is remotely configurable. Drones can also be connected together to form a processing pipeline, or swarm. Configuration and connection commands are issued by applications to set up a swarm to accomplish a specific task. Applications can construct multiple swarms in order to perform various complicated tasks simultaneously then collate the results. Applications and drones are both Hive modules. The term *user* applies to the developer using Hive to construct a distributed system.

Although Hive has been designed to allow general purpose network pipeline systems, the main goal has been to apply it to computer vision systems. Swarms tend to begin with a source drone such as a camera module which supplies image data to the video processing pipeline (such as background subtraction and object tracking drones). Multiple swarms can be set up to process the same incoming data to retrieve different information, or to perform the same task in a distributed manner to speed up the rate of processing. Error handling for distributed processing is built into drones as an exception model, which can pause processing or be handled by the application which controls the swarm.

Hive is based on the concept of encapsulated and distributed processing. By creating a Hive module which performs a specific task (e.g. background subtraction), this module can run in its own process independently of other modules. It can then be connected by an application to a data source and have its output connected to another drone. Drones can be reused by different applications and can run on any computer on a network. They can be upgraded or changed and as long as the interface remains the same, they can perform their task transparently. There could also be multiple drones on a network each doing the same task with different input sources. Likewise, drones can easily live on a single multi-core computer sharing memory and various internal buses. To provide these features, Hive provides a layered set of functions.

3.1. Hive's Layered Model

Applications and drones both sit on top of the layered architecture of Hive as shown in Figure 2. The architecture is comprised of four layers, two of which are shared between applications and drones. The layers are, from the bottom up:

- *Transport*: deals only with inter-communication of Hive modules, using a peer-to-peer system. Each drone has a server listening for new data, as well as a client for sending data to other drones. The transport layer is also used by applications for communication with drones. Systems may be implemented over a network using protocols such as TCP/IP, or using shared memory and internal buses on a single machine.
- *Event*: forms the basis of the communication protocol and data transfer methods by generalizing all communications to events. Both applications and drones have

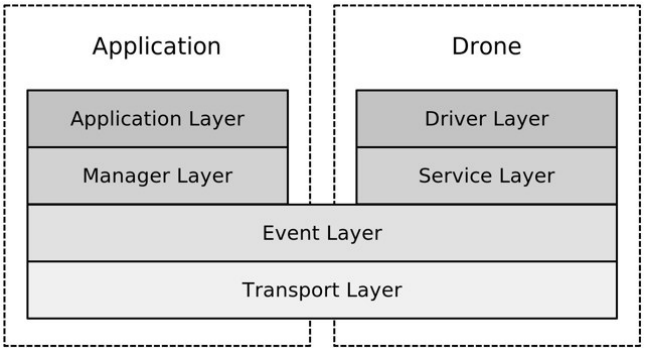

Fig. 2. The layered architecture of Hive for both applications and drones. They share transport and event functionality, but diverge above these.

an event handling and callback system.

- *Service*: is a drone specific layer, supplying the interface to Hive for user-written drivers to specific functionality such as getting camera data or processing a video stream. It also handles callback queueing, resource allocation and registration methods for drone and swarm configuration.

- *Manager*: is an application specific layer, providing the functionality required to control and configure drones and swarms. It also contains all the functionality of the service layer except it does not provide configuration methods (since applications are not configurable by drones).

- *Application* / *Driver*: are the application programmer defined sections which make up a Hive module. The driver is the interface to Hive from a software implementation or from a hardware device, and the application is the controller for drones.

We describe each of these in detail in the next subsections.

3.2. Hive Drones

A Hive drone is a reusable module with a clearly defined interface which indicates what it expects to receive as input and what it will produce as output. Drones can be linked together to form swarms capable of performing complicated vision tasks or distributing an intensive computation across multiple processes. Drones must provide methods which allow configuration of their internal state. A typical drone may be a camera or a specific algorithm such as a face-recognizer as we demonstrate below.

As described in Section 3.4, applications are in effect a special kind of drone, as there can be more than one (although they do not have direct control over each other) and communication is identical. However they have a superset of functionality comprised of control and configure commands for drones. Unlike drones, they do not have to provide methods to customize their behaviour.

Error reporting is accomplished using an exception model built into the Hive service layer. If an error occurs which a drone is not able to recover from, it notifies the application by sending an exception. Drones can be pre-configured to halt all processing when an exception occurs, and resume when told to by the application. Alternatively a drone can notify the application, abandon the current state of operation and resume from the next set of input data it receives.

3.3. Hive Swarms

One of the novel contributions of this work is the ability to connect different drones together into swarms, demonstrating the reusability of drones and the flexibility of the architecture.

Drones are designed as encapsulated processes (much like a class in an object-oriented programming language) so that they can be connected together to perform arbitrary tasks. For example, if different drones are created for a camera, background subtraction, object detection and gesture recognition, a gesture recognition system swarm can be constructed through an aggregate of these drones. The same camera and background subtraction drones could be used for an object tracking swarm.

Drones are connected together by applications using *control* events. These are events which are processed inside the event layer upon arrival and hidden from the user. Control events can be one of two types: `register` and `connect` (including their counterparts `deregister` and `disconnect`).

A `register` event is sent by a drone (R) to request certain data from another drone (S). The `register` event is processed by S by adding R to the table of receivers for that event. This means that whenever S produces data corresponding to this event type, R will be sent that event (with the data or just as a notification). Registration can be done once for streaming, or repeatedly using a synchronized model. If using the synchronized model, every time new data is sent the receiving drone is removed from the list of registered receiving drones. The data transfer models are discussed in more detail in Section 3.5.

A `connect` event is sent by an application to a drone, instructing it to connect itself to another drone (which is done via a `register` event). This is the mechanism used to connect two drones together. By issuing multiple connect commands to various drones, applications can construct swarms built from two or more drones. Swarms can also be initialized to work from the same source in a one-to-many scenario: a single camera can be the source for many different processing drones e.g. background subtraction and object detection

can execute simultaneously on the same image.

Control events are dealt with at the event layer so they remain hidden from the application programmer. As well, they may be implemented peer-to-peer or through a single point manager depending upon how Hive is implemented.

3.4. Hive Applications

Hive applications are the command and control centres for drones and the swarms to which they belong. Applications use the same transport and event handling mechanisms as drones, but have additional methods for setting up swarms and configuring individual drones.

An application connects receiver and sender drones together using the process described above. Applications can also connect themselves to drones using the same process, allowing them to receive final results and monitor progress.

Drone configuration is achieved via remote procedure calls (RPC). The RPC sends a control event to the drone for parameter change with the parameter list, to which the drone responds with a confirmation event signifying success or failure. The RPC called by the application blocks until the confirmation message is received. Hive will be extended in future to allow simultaneous configuration of identical drones (or drones which accept the same settings). A similar approach is used for getting parameter settings of a drone; such as camera parameters.

3.5. Hive Data Transfer Models

Data transfer is supported under two separate models, *synchronized* and *streaming*.

Synchronized data transfer has been included to ensure a processing pipeline operates at its capacity and does not waste unnecessary bandwidth. Once a drone has been set up to receive data from another, it will request the next available data from the source when it is ready. After receiving the data the drone enters a 'busy' state for processing, and will automatically request the next piece of data when it is finished. Any data produced by the source in the interim is not sent to the drone for processing.

As a simple example of synchronized data transfer, if a camera is the source drone running at 30fps and an object tracker is the processing drone running at 10fps, using synchronized transfer ensures bandwidth is not wasted by sending images across the network to the tracker when it cannot process them. If more drones are included in the swarm, this method slows the overall speed of the swarm to that of its slowest drone, conserving bandwidth and reducing overload on individual drones. This does not affect the speed of other swarms, and does not exclude the possibility of attaching higher performance drones to other swarms to increase overall efficiency.

There is some small overhead associated with this method, since every new piece of data must be registered for by the receiving drone. However, the registration is typically only a single packet of data which is negligible compared to the size of an image.

Streaming data transfer is a low-overhead high-bandwidth method, which requires a single registration at the beginning of operation. In this model, a source drone continuously sends data to the processing drone regardless of its state. This model is designed for systems known to be capable of processing at the required speed, but also for different forms of processing, e.g. image-based systems benefit from the synchronized method, but if pixel or scanline data is needed, this could easily be streamed across the network.

Both models allow for two possible callbacks types: one where the transferred data is included in the call and another that excludes data when being notified of an event, basically creating a notification system. Notification is useful for applications wishing to monitor progress but not receive the data itself. Both models are useful in different contexts and are supported.

3.6. Hive Communication

All communication in Hive is event-based (including the implementation of the RPC for configuration) and event handlers are built into both drones and applications. There are two separate data flows associated with the architecture, as shown in Figure 3: receiving data to be processed, and sending data produced by a drone. This section covers the process associated with receiving and sending data.

3.6.1. Receiving

The path of incoming data in a drone begins with the Hive Listener running in the transport layer, waiting for new data from other drones or commands from applications. The entire packet is read from the network and subsequently passed up to the event layer. This is outlined in Figure 3(b). This is the server-side operation of the module: the server only receives data, never sends, because established connections from a client interface do not connect to the client module's Listener. Therefore modules operate in client and server mode concurrently

When the packet arrives at the event layer, the header is inspected to discover which type of event is contained within: events can either be *control* or *user* events. Control events are used by Hive to set up swarms and configure drones, and are hidden from the user (see Section 3.3 for more information). User events are defined in the drone interfaces, and are used to connect two drones together.

If the packet does contain a control event, the command is processed in the event layer. If the event is classified as user, the packet is passed up to the service/manager layer. As in most event handling systems, callbacks are used to process incoming events. Hive modules register callbacks at the start

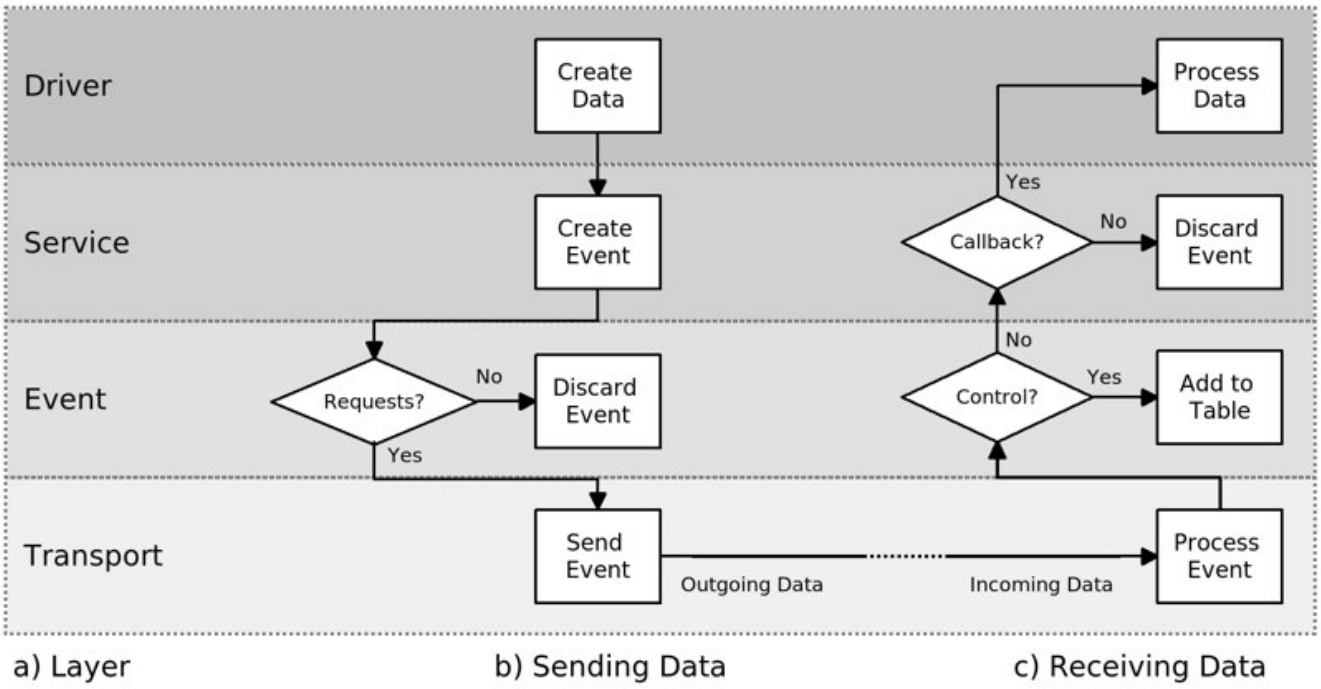

Fig. 3. The flow across layers associated with sending (b) and receiving (c) data across the network.

of execution for specific events they expect to receive. These are registered with the service layer which maintains a callback table. When an event arrives at the service layer from the event layer, the event type is looked up in the callback table to check the drone has registered for it (this is more of a double check, since if an application connects two drones, the event received should have a callback registered). The data is put in a callback queue to be processed once the drone has finished its current processing; this keeps the user-defined functionality in a single thread and avoids data synchronization issues.

3.6.2. Sending

The process of sending data is relatively less complicated since all the receivers have registered for events. The data flow is shown in Figure 3(c). When a drone produces data of a specific kind, it passes it to the service layer where it is wrapped as an event and passed to the event layer. The event layer maintains a table of requests from other drones or applications for each kind of data the current drone produces. The table of requests for this event type is checked by the current drone to see which Hive modules have requested this data, and the event layer sends the data to each one using the client interface of the transport layer.

```
Hive::DeviceID dev_id(port);
Hive::RegisterCallback(Events::SET,
                       &Set);
Hive::RegisterCallback(Events::GET,
                       &Get);
Hive::RegisterCallback(Events::IMAGE,
                       &ProcessImage);
Hive::SetMain(&DroneMain);
Hive::StartDrone(dev_id);
```

Fig. 4. This C++ code snippet shows the code required to set up a Hive drone. The code: sets up the device on a specific port (for the Hive Listener); registers functions to deal with requests for set and get parameters; registers a function to call when an image arrives; registers the main drone function which is called repeatedly by the Hive service layer; and finally the drone is started with the device id.

3.7. Hive Layer Interface

The Hive API has been kept simple while maintaining flexibility for general applications. Hive is accessed by supplying a set of functions which are called when applicable by the managing routines. Drones behave as stand-alone processes and provide a `main` function (called `DroneMain` in Figure 4), `set` and `get` parameter functions, and define functions as callbacks appropriate for specific data types (which will arrive and be processed by the event handling system). Applications provide a `main` function and a set of callbacks to deal with exceptions and data from the drones that they are interested in.

Each Hive module contains two active threads: one executes the user-based functionality (within `DroneMain` and `ProcessImage` in Figure 4), and the other operates the Hive Listener and event handling system. Thread-safety is maintained in user code by executing event callbacks in the same thread as the `main` routine: events are queued as they arrive, and the queue is processed when the `main` routine finishes its current processing.

3.8. Additional Language Support

Hive has been designed to be cross-platform, and where possible, work with other languages. The architecture contains the language layer, an additional layer above the Manager / Service layer which acts as a translator to other programming languages. Hive has been implemented natively in C++, but also contains a C interface and python bindings so that applications and drones can interoperate using pre-built executables or the python interpreter. It would be possible to change the native implementation if necessary. Additional languages can be added, with minimal development effort, through bindings of the service layer.

4. PROOF-OF-CONCEPT FACE DETECTION APPLICATION USING HIVE

This section describes the implementation of a face detection application that has been developed using Hive. The face detection Hive application uses two drones: a camera and a face detector as described below. The computational complexity of the face detection algorithm makes it a suitable candidate for demonstrating the modularity and distribution properties of the Hive system. The implementation of the Hive layered architecture is in C++ and uses the TCP/IP suite of protocols for communication, allowing devices to reside anywhere on the network.

4.1. Axis Camera Drone

The source drone for the face tracking application is designed to use the Axis 207 network camera. This camera is highly configurable and exposes its services through an HTTP based API called VAPIX[16]. Using this API, the user is able to receive image data and configure the camera's many parameters such as resolution, exposure, image format, compression level etc. The Hive drone wraps the VAPIX API, providing a convenient abstraction for the camera's functionality.

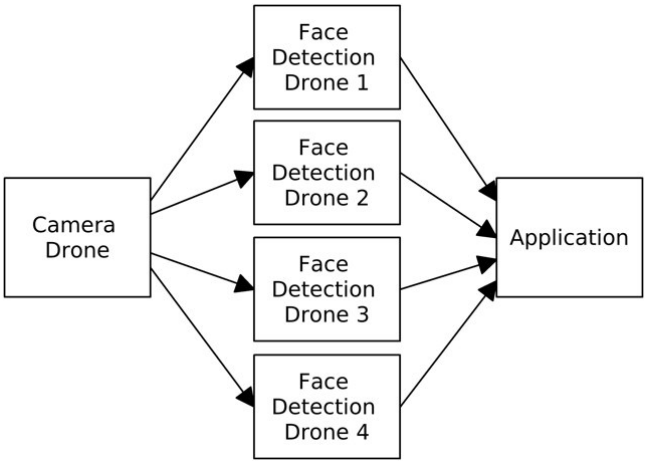

Fig. 5. Data flow between the different drones and the application in the face detection system

In order to provide abstraction, the Hive module exposes the functionality of the device to the service layer. The functionality can be classified into two categories: configuring parameters (including parameter retrieval) and getting image data. The request to set or get parameters is initiated by a Hive application drone through the manager layer, and the requests are dealt with by the drone's service layer. The routines to handle these requests are registered as callbacks on the service layer. The availability of image data is dictated by the device, so the 'Main' method of the drone polls the camera for new data.

4.2. Face Detection Drone

The face detection drone is a software routine that finds face like patterns in images. The algorithm used for this module is OpenCV's haar-like feature detection[17]. This algorithm scans sub-regions of the frame sequentially and at various scales using the haar descriptors in order to find areas that corresponds to a face. For this application high accuracy is required and therefore a low scale factor is used, making this operation computationally expensive.

The face detection module provides the following services: configuring the haar-like features; accepting images from the source; finally, processing the images to produce the size and location of the faces in view. In addition to the parameter configuration callbacks, the face detection drone registers an additional callback to deal with incoming data from a source. This callback routine decodes the incoming jpg_image and

sets a new_frame flag indicating that there is new data to be processed. The face detection routine operates in the main method, which checks the new_frame flag and starts processing if it is set. Once the routine completes, it creates a new event with the annotated image. The event is propagated to the event layer and sent to any drone or application that has requested the data.

4.3. Face Detection Application

The goal of the Hive Face Detection Application is simply to display the face detection results. To accomplish this task the application needs to: declare and configure the drones; establish data flow amongst the drones; receive and display the results from the drones.

The raw data is generated by the Axis camera drone, which is sent to the face detection drone for processing. Results are sent to the application for viewing. This data flow is established by the application, which connects the face detector drone (the receiver) input to the camera drone (the sender) 'new image' event. It then connects itself to the 'processed frame' event of the face detector drone. In our proof-of-concept system, the face detection drone is set to receive raw images from the Axis 207 camera synchronously (request per frame). The application receives data via a user-registered callback once the processed frames from the face detection drone are available. The application callback fills in a display buffer and calls the display function that displays the processed image using OpenGL and the Graphic Library Utility Kit (GLUT). Figure 6 shows the output of the Hive face tracking system. For this proof-of-concept, the Axis Camera Drone was running on Windows XP, and the Face Detection Drone and Face Detection Application were running on a PC with Linux operating system (Ubuntu 7.10).

4.4. Results

The images in Figure 6 show the results of the face detection system running on Hive. The scene conditions (the background and subjects) were kept as constant as possible in order to minimize the variability in performance due to external factors. Our interest was not to optimize the face-detection algorithm, but to see how Hive can be reconfigured simply and provide the ability to easily distribute processing. Using a single drone to process the images on average gave a rate of 2.3 frames per second. Adding a second drone, and moving the processing drones to a different machine from the application and source drone, resulted in an improved rate of 4.4 frames per second. Spawning additional drones on other machines would increase the throughput of the system, and the minimal overhead involved requires augmenting the application to use the new drones.

5. CONCLUSIONS AND FUTURE WORK

We have presented Hive, a novel middleware for vision processing systems. We have discussed the architecture of Hive and its ability to provide mechanisms and abstractions to enable capture and processing of vision data through a modular, flexible and plug-in architecture. Using our proof-of-concept application, we have demonstrated that a single vision system based on the Hive architecture can run on multiple PCs with non-homogeneous platforms. Although the current implementation of Hive is designed for large and distributed systems we are working on a light version of Hive that is more suited for single machine use by modifying the transport layer to support inter-process communication by using the internal high speed buses on a single computer.

6. REFERENCES

[1] D. Arita, Y. Hamada, and R. Taniguchi, "A real-time distributed video image processing system on pc-cluster," in *Proceedings of International Conference of the Austrian Center for Parallel Computation(ACPC)*, 1999, pp. 296–305.

[2] J. Fung and S. Mann, "Openvidia: parallel gpu computer vision," in *Proceedings of ACM Multimedia*, 2005, pp. 849–852.

[3] Michael H. Schimek, Bill Dirks, Hans Verkuil, and Martin Rubli, "Video for linux v4.12: http://v4l2spec.bytesex.org/v4l2spec/v4l2.pdf," Tech. Rep. 0.24, Linux, 2008.

[4] 1394 Trade Association, "1394-based digital camera specification version 1.20.," Tech. Rep., IEEE, 2007.

[5] Intel Corporation, *Open Source Computer Vision Library: Reference Manual*, Intel Corporation, 2001.

[6] Jeffrey M. Squyres, Andrew Lumsdaine, and Robert L. Stevenson, "A cluster-based parallel image processing toolkit," in *Proceedings of the IS&T Conference on Image and Video Processing*, 1995.

[7] Klas Nordberg, Per-Erik Forssen, Johan Wiklund, Patrick Doherty, and Per Andersson, "A flexible runtime system for image processing in a distributed computational environment for an unmanned aerial vehicle," in *In Proceedings of IWSSIP*, 2002.

[8] M. Jovanovic and B. Rinner, "Middleware for dynamic reconfiguration in distributed camera systems," in *Intelligent Solutions in Embedded Systems, 2007 Fifth*, 2007, pp. 139–150.

[9] H. Detmold, A. Hengel, A. Dick, K. Falkner, D. S. Munro, and R. Morrison, "Middleware for distributed

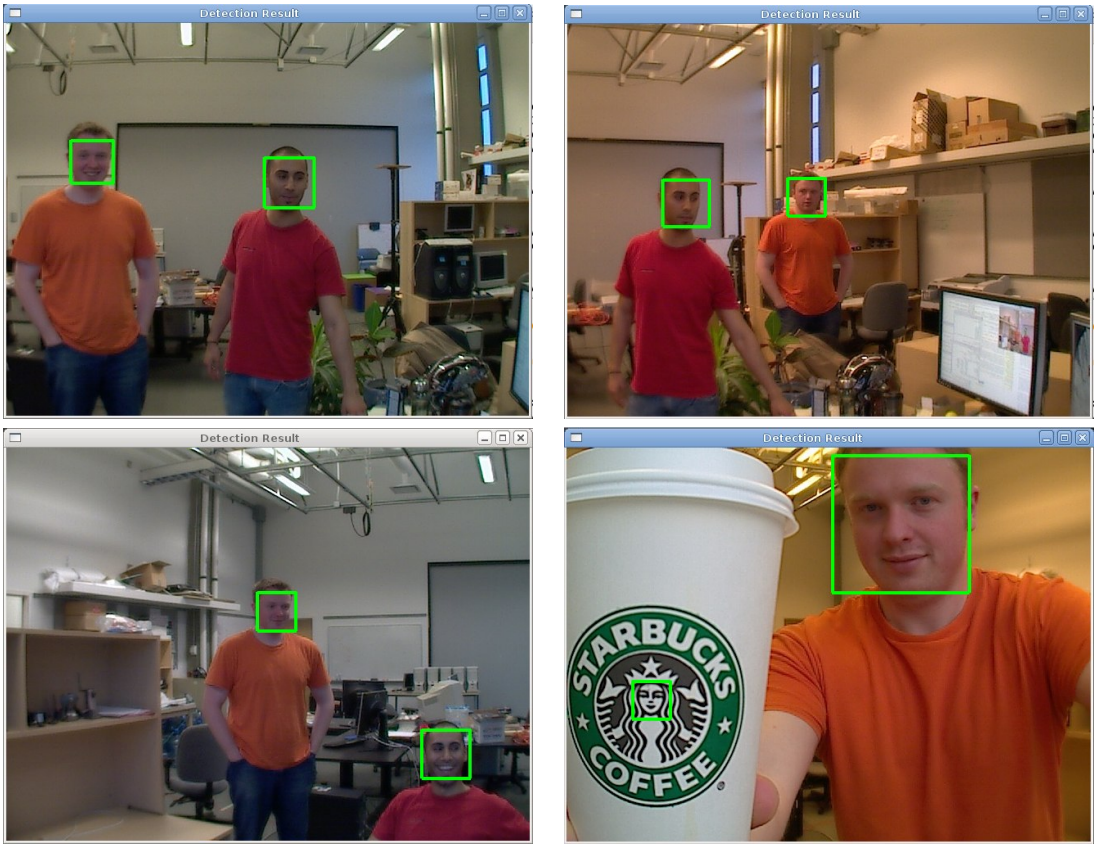

Fig. 6. Results of the face detection application running on a Hive system distributed across four drones, with one application in control which displays the result (shown).

video surveillance," in *Distributed Systems Online, IEEE*, 2008, pp. 1–1.

[10] D. Arita, Y. Hamada, S. Yonemoto, and R. Taniguchi, "Rpv: a programming environment for real-time parallel vision - specification and programming methodology," in *Proceedings of 15th International Parallel and Distributed Processing Symposium*, 2000, pp. 218–225.

[11] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: yet another robot platform," in *International Journal of Advanced Robotics Systems*, 2006.

[12] B.P. Gerkey, R.T. Vaughan, K. Stoy, A. Howard, G.S. Sukhatme, and M.J. Mataric, "Most valuable player: a robot device server for distributed control," in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, 2001, pp. 1226–1231.

[13] George Pava and Karon E. MacLean, "Real time platform middleware for transparent prototyping of haptic applications," in *Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2004. HAPTICS '04. Proceedings. 12th International Symposium on*, 2004, pp. 383–390.

[14] H. Zimmerman, "Os1 reference model-the is0 model of architecture for open systems interconnection," in *IEEE Transactions on Communications*, 1980, pp. 425–432.

[15] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes, "Hive: Distributed agents for networking things," in *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

[16] VAPIX, "Axis communication application programming interface: http://www.axis.com," Tech. Rep., AXIS, 2008.

[17] Rainer Lienhart and Jochen Maydt, "An extended set of haar-like features for rapid object detection," in *Proceedings of International Conference on Image Processing*, September 2002, vol. 1, pp. 900–903.