Lightweight Agent Framework for Camera Array Applications

Lee Middleton, Sylvia C. Wong, Michael O. Jewell, John N. Carter, Mark S. Nixon

School of Electronics and Computer Science, University of Southampton, UK
{ljm,sw2,moj,jnc,msn}@ecs.soton.ac.uk

Abstract. This paper describes a lightweight middleware agent framework (LAF) for coordinating a large array of computers with attached cameras to construct high resolution video-rate image sequences. Compared to existing camera middleware, LAF provides more than a remote sensor access API. The use of an agent framework allows reconfigurable and transparent access to cameras, as well as software agents capable of intelligent processing. It also eases maintenance by encouraging code reuse. Other features include an automatic discovery mechanism at startup, and multiple language bindings. Performance tests showed the lightweight nature of the framework while validating its correctness and scalability. Two different camera agents were implemented to provide access to a large array of distributed cameras. Correct operation of these camera agents was confirmed via several image processing agents.

1 Introduction

Increasingly, large arrays of high quality cameras are used for capture and analysis of motion. Kanade et al. mounted 49 cameras in a room to capture motion for virtualised reality [1]. Wilburn et al. built a dense CMOS camera array to achieve high resolution and framerate capturing of image data [2]. Zhang et al. built a large self configuring camera array capable of rendering novel views of scenes in near real time [3]. In all these works multiple cameras are employed to produce a higher quality image than would be possible with a single camera. Specifically, we seek to deploy high resolution video-rate data captured from multiple cameras for the analysis of human gait.

In this paper we propose a middleware framework for a camera data acquisition system. The aim is to allow transparent and reconfigurable access to visual data while minimising maintenance by encouraging code reuse. Real time streaming is not required because gait analysis requires high resolution data. Additionally, the complexity of the algorithms employed make it impossible to do on-the-fly processing. The middleware facilitates communication between dedicated camera computers and image/gait processing software. It has the following features: (i) *Zeroconf*: This allows agents to automatically locate middleware components in a network. (ii) *Multi language support*: This allows users to exploit the benefits of the different languages. (iii) *Lightweight*: The algorithms used in gait analysis are CPU intensive, thus the middleware must not be an additional adverse drain on resources. (iv) *Service discovery*: Agents can query the middleware to discover and utilise services provided by other agents. (v) *Locking*: Cameras are stateful devices. It is important that processes cannot be interrupted mid-session.

Hori et al. [4] also implemented a middleware for networks of computers with attached cameras. Here, cameras were accessed as if they were a local device. This is similar to player/stage [5], which provides software abstraction for robot sensors. However, these systems are not suitable for our application as their goals are to provide direct access to sensor data over a network. In contrast, we want intelligent agents in our framework, where researchers can provide agents that perform complex algorithms. Multi-camera tracking systems [6, 7] also study the same problem. Here, camera agents not only act as capture devices, but also perform processing on the image data. The middleware is highly focused on the task of tracking. Thus the messages are high level commands like location of objects. The middleware is also responsible for coordinating the movement and focus of the cameras to achieve a goal. In comparison, the middleware in our application has to be more general purpose. This is because researchers have different requirements from the image data. For example, in our research group, there are people who work with raw image data, silhouettes [8], and 2D and 3D models [9]. A final approach to this camera coordination problem is to leverage an existing middleware such as CORBA [10] or XML-RPC [11]. However, both systems require lot of resources to run. Many features not required in our application are included by default, and they cannot be optionally switched off. Also, there is a steep learning curve before researchers can add their existing code to this framework.

Our proposed solution contains some of the features from all the approaches outlined above, while being easy to use and lightweight. The paper is organised as follows: Section 2 provides an overview of the Lightweight Agent Framework (LAF). Section 3 introduces the two camera agents essential to our camera data acquisition system. Section 4 presents results from performance tests. Finally, Section 5 describes application agents that employ the services provided by the camera agents.

2 System Overview

Figure 1 shows an overview of LAF. The system has been simultaneously developed on C++, Java, and Python. Central to the system is the *router*. It is the main point of communication and coordination. All *messages* between components (except streamers) are sent via the router. Also, it acts as a broker for agents providing and requiring services. Agents are providers of services. Remote agents are clients of services provided by agents. To use the service provided by an agent, a remote agent requests a *lock* on the agent from the router. However, LAF is not restricted to a simple model of clients (remote agents) and servers (agents). An agent can contain one or more remote agents, thus allowing it to be both a client and a server at the same time. Ports are inputs and outputs of agents. Streamers are direct socket connections, mediated via the router. This allows video information to be sent directly between agents without the traffic passing through the router.

Ports and Streamers There are two types of ports in LAF: input and output. Input ports are used to pass data to an agent. They can be optional or non-optional. Processing cannot proceed until all non-optional ports are *set*. Output ports are used to pass data to remote agents. In applications involving a large array of cameras, a large amount of data is generated. If this data is to be sent via the router, the router may fail. For this reason, streamers were implemented. A streamer is a direct socket connection between an agent and a remote agent. The router is responsible for instigating this connection, so address information is not required ahead of time.

Messages LAF employs XML messages for communication between the router, agents and remote agents. Broadly, there are three classes of messages – status, router and agent. Status messages give feedback about the success of an action. Router messages are actions that only the router can perform. Agent messages are communications, via the router, between agents and remote agents. They deal with control and communication.

Router The router, agent and remote agent are implemented via a common class hierarchy, as shown in Figure 2. The router is responsible for agent subscription, message re-direction, and agent selection. It employs a plugin system which makes it simple to extend its functionality. Upon starting, the router registers itself as a multicast DNS service (mDNS) in zeroconf. The mDNS service allows information to be passed to any subscriber of the service. In this case the port and IP address of the router are passed via mDNS. Essentially this means that connection to the router by any agent is potentially an automatic process.

The subscription process of an agent from the perspective of the router begins when a `SUBSCRIBE` message is received. This message contains the *type* of agent which is being subscribed. The type of an agent is purely a descriptive name describing the service it provides. When the subscription is received the router assigns a unique name for the agent. This is made up of the type and a unique id number. Once an agent is subscribed, it is added to two lists (connected agents and free agents) which are maintained in the router. If an agent unsubscribes or dies, the router removes it from both lists. All messages pass through the router. As a result of this the router can perform filtering of the messages. For example, some messages are permitted only if the sender has a lock on the target. If the sender does not have the required lock, the router returns an `NOK` message to the sender. In most cases however the extent of the routers manipulation of the message is handling acknowledgements and passing it onward to the appropriate target. The last function of the router is the agent selection process. This is used when a remote agent attempts to lock an agent. The selection mechanism currently employed is a naïve one. A handle to the first agent of the correct type on the free agent list is returned to the remote agent. The selected agent is also removed from the free agent list.

Agents and Remote agents Users writing agents for LAF will need to create a derived class of `Agent`. The `Agent` class provides the underlying networking and messaging required for all agents. The derived class will minimally need to (a) provide

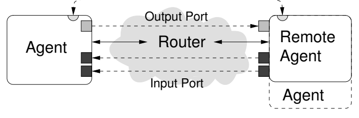


Fig. 1. An overview of how the middleware and agents fit together.

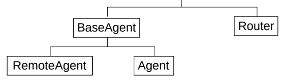


Fig. 2. Diagram showing the interrelation of the software objects.

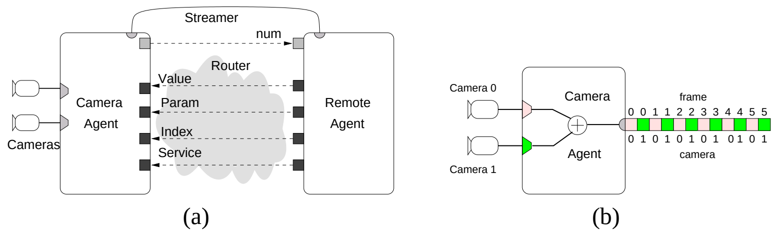


Fig. 3. (a) Configuration of camera agent. (b) Sending image data with a streamer.

type, ports, and streamers in the constructor, and (b) overwrite the `action()` method. Type is a descriptive name of the service provided. Ports and streamers provide the inputs and outputs of this agent. The `action` method is the engine of the agent. The user writes their own action method to provide the agents' functionality. Users writing remote agents will need to create a derived class of `RemoteAgent`. The derived class will need to specify the name(s) of the agent(s) it wishes to lock, set the agent's input ports and call the `action()` method. Agents and remote agents can connect to LAF either automatically or manually. Automatic connection is performed using zeroconf while manual connection uses environment variables.

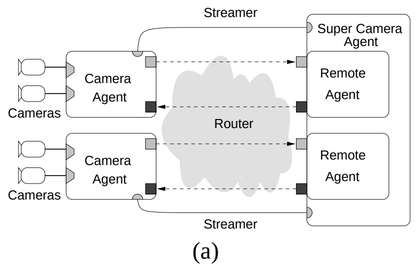
3 Camera Agents

Figure 3(a) illustrates the specific ports and general configuration for a camera agent. Only the *service* port is compulsory. The service port exposes the features of the cameras controlled by the agent, such as grab image and set shutter speed. The other input ports were employed to set required parameters. The *streamer* is used to transmit image data over the network directly to remote agents. The video data is sent a frame at a time with the frame from each camera interleaved as shown in figure 3(b).

The camera agent provides access to the camera array on a per PC basis. However, accessing and controlling a large number of cameras can be cumbersome using camera agents alone. For instance to access 6 cameras which are connected in pairs to each of three PCs the user needs to maintain three separate camera agents. As a result, a super camera agent was written to provide a single collated image data stream to image processing applications. The configuration for a super camera agent is shown in Figure 4(a). It contains a number of remote agents for locking camera agents connected to the router. Thus, the super camera agent is both a remote agent and an agent. Image data is transmitted to the super camera agent via a number of streamers. These are demangled into a single output stream for other other remote agents to consume (Figure 4(b)).

4 Performance Testing

Four tests were carried out to evaluate the performance of LAF. Figure 5 shows that connection and disconnection times to LAF do not increase with an increasing number of registered agents. The second test measures the overhead of messaging. One hundred



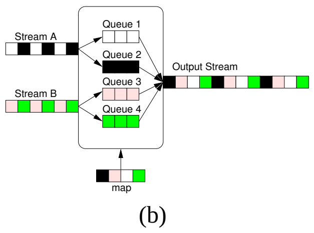


Fig. 4. (a) Configuration of super camera agent. (b) Demangling of image streams.

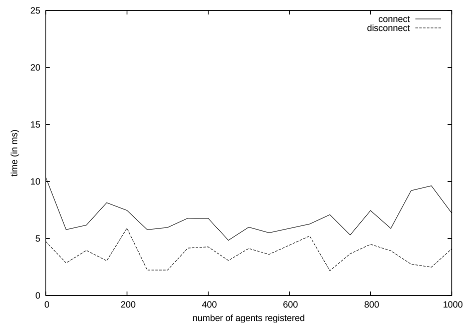


Fig. 5. Connection and disconnection time against number of agents registered.

	remote agents	
agents	C++	Python
C++	943	1771
Python	1596	2158

Table 1. Time (in ms) to perform 100 string concatenation operations.

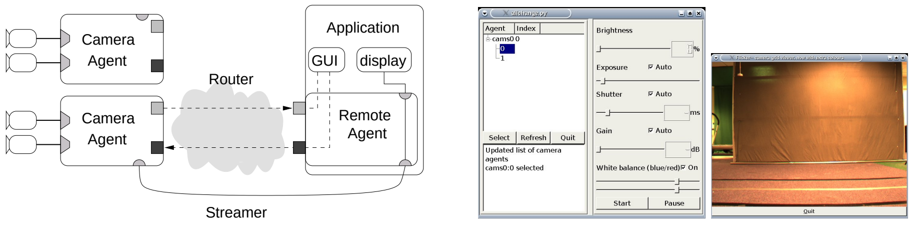
	time (ms)
C++ agent	263
C++ remote agent	291
Python agent	512
Python remote agent	604

Table 2. Startup time.

separate operations were invoked against an agent with two mandatory inputs and a single output. The result is shown in Table 1. The third test measures the average startup times of agents and is illustrated in Table 2. The slower start up times of the remote agents is the overhead due to locking. The messaging test and the startup time test demonstrate the lightweight nature of LAF. The fourth test examined the streaming performance. For a camera agent we achieved an average of 661 Mbit/s with a gigabit network. This means we can directly (640×480, 30fps) stream video data from 9 camera.

5 Application Agents

As an example of the system in operation three application agents are described here. Firstly, an agent was designed to allow remote configuration of cameras (see Figure 6). Secondly, an image mosaicing agent was designed. Figure 7(a) shows a composite image created by this agent. Finally, a background subtraction agent was developed. It locks a single camera and statistically computes a reference background when no subject is in view. When the subject is in the scene, the reference is used to find portions of the image that have changed. Figure 7(b) shows an example output frame.



cams0 0
0
1

Updated list of camera agents
cams0:0 selected

Brightness %
Exposure ☑ Auto
Shutter ☑ Auto ms
Gain ☑ Auto dB
White balance (blue/red) ☑ On

Fig. 6. Camera control application agent.

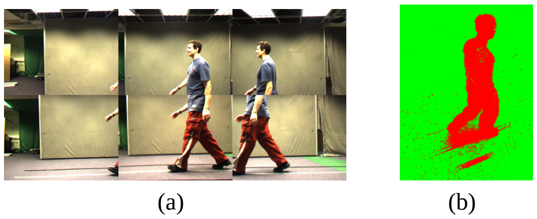


Fig. 7. (a) Creation of an uncalibrated image mosaic from six cameras on three PCs using the image mosaicing agent. (b) Output from the background subtraction agent.

6 Conclusions

This paper described the development of a middleware, LAF, for the control of a large array of cameras. LAF consists of a router, and superclasses for agents (service providers) and remote agents (clients). Since agents can contain remote agents, LAF is not limited to a simple client/server model. LAF provides transparent access to camera control. Additionally, it facilitates access and reuse of intelligent agents that provide a variety of image processing operations. Other features of LAF include minimal computational overhead, zeroconf for automatic discovery of components in the framework, locking, and multiple language support. This paper also presented performance tests and applications of LAF. By measuring the connection and disconnection times of agents on an increasingly loaded router, scalability was demonstrated. Messaging tests showed the overhead of communication, in either C++ or Python, was small. Measured startup times showed the registration process to be short. Video streaming tests validated that the throughput in the designed system was sufficient to stream 9 cameras. Three application agents were also demonstrated in this paper. These applications validated the correctness of the design and demonstrated the ease of implementation of image processing applications.

References

1. Kanade, T., Saito, H., Vedula, S.: The 3d-room: Digitizing time-varying 3d events by synchronized multiple video streams. Technical report, Carnegie Mellon University (1998)

2. Wilburn, B., Joshi, N., Vaish, V., Levoy, M., Horowitz, M.: High speed video using a dense camera array. In: Proceedings International Conference on Computer Vision and Pattern Recognition. (2004)
3. Zhang, C., Chen, T.: A self-reconfigurable camera array. In: Eurographics Symposium on Rendering. (2004)
4. Hori, T., Nishada, Y., Yamasaki, N., Aizawa, H.: Design and implementation of a reconfigurable middleware for sensorized environments. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems. Volume 2. (2003) 1845–1850
5. Gerkey, B., Vaungan, R.T., Howard, A.: The player/stage project: Tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th International Conference on Advanced Robotics. (2003) 317–323
6. Sato, K., Maeda, T., Kato, H., Inokuchi, S.: CAD-based object tracking with distributed monocular camera for security monitoring. In: Proceedings of the Second CAD-Based Vision Workshop. (1994) 291–297
7. Orwell, J., Massey, S., Remagnino, P., Greenhill, D., Jones, G.A.: A multi-agent framework for visual surveillance. In: International Conference on Image Analysis and Processing. (1999) 1104–1107
8. Veres, G., Gordon, L., Carter, J., Nixon, M.: What information is important in silhouette-based gait recognition. In: Proceedings of IEEE Computer Vision and Pattern Recognition conference. (2004)
9. Wagg, D.K., Nixon, M.S.: On automated model-based extraction and analysis of gait. In: Proceedings of 6th International Conference on Automatic Face and Gesture Recognition. (2004) 11–16
10. Henning, M., Vinoski, S.: Advanced CORBA Programming with C++. Addison Wesley (1999)
11. Tang, J., Tong, W., Ding, J., Cai, L.: MOM-G: message-oriented middleware on grid environment based on OGSA. In: International Conference on Computer Networks and Mobile Computing. (2003) 424–427