Indiva: A Middleware for Managing Distributed Media Environment

Wei Tsang Ooi[a]*, Peter Pletcher[b] and Lawrence A. Rowe[b]
[a] School of Computing, National University of Singapore, Singapore.
[b] EECS - CS Division, University of California, Berkeley, USA.

ABSTRACT

This paper presents a unified set of abstractions and operations for hardware devices, software processes, and media data in a distributed audio and video environment. These abstractions, which are provided through a middleware layer called *Indiva*, use a file system metaphor to access resources and high-level commands to simplify the development of Internet webcast and distributed collaboration control applications. The design and implementation of Indiva are described and examples are presented to illustrate the usefulness of the abstractions.

Keywords: multimedia middleware, webcast production, equipment control, session management, Access Grid

1. INTRODUCTION

Webcast and distributed collaboration are important Internet applications. Many educational and research institutions have begun to webcast lectures and seminars over the Internet, and distributed collaboration systems such as the Access Grid (AG) are increasingly popular. For example, the number of AG nodes has grown from 20 to 182 during the past three years.[1, 2] However, these systems are difficult and expensive to operate – a high quality, multi-stream webcast production requires 3 to 5 skilled equipment operators and an AG conference requires at least one operator at each location.

To reduce the cost and complexity of operation, several research groups, including ours, have developed applications to automate webcast and distributed collaboration production.[3–6] Our group developed two applications, namely, the *Director's Console*[7] and *Broadcast Manager*,[8] to simplify the production of a live webcast. These applications enable a single operator to produce a high-quality live webcast, thereby reducing cost.

However, several problems remain. First, applications that manage a production environment are difficult to write. Detailed knowledge of the environment is required. The problem is that we cannot specify "what we want," rather we must specify "how to implement a request." For example, to add a TV channel off a satellite dish, into a webcast, we must (i) send a command to the satellite receiver to change to the desired channel, (ii) send a command to the routing switcher to switch the output from the satellite dish to a capture computer with available capture card, and (iii) launch an encoder process on the selected capture computer. Second, applications are often tightly coupled with the environment and cannot adapt well to changes. When new equipment is added or when cables are rewired, applications have to be modified. Furthermore, an application written for one environment cannot be easily reused in another environment. These problems constrain the development of innovative control and automation software for webcast and distributed collaboration.

We believe a solution to these problems is to provide a layer of abstraction between the audio/video environment and the applications. Much like relational databases and UNIX file systems, this layer should provide abstractions and an API for managing entities in the environment without revealing the underlying details. As an example, we should be able to say "show me this TV channel", or "move this stream from one session to another", without worrying about how it is actually implemented. This layer of abstraction will allow programmers to focus on more important issues such as automation algorithms rather than low level details such as routing signals and launching processes.

This paper describes the design and implementation of such an **IN**frastructure for **DI**stributed **V**ideo and **A**udio (*Indiva*). The Indiva middleware provides easy to understand abstractions based on a file system metaphor and high-level commands required by application developers. Beside ease of use, Indiva is designed to be extensible and robust. Extensibility allows minimal modification to applications when the environment changes, while robustness ensures that equipment and software failures are handled gracefully. These goals are achieved using the following ideas: (i) separation of logic from data, (ii)

*This work was completed when the first author was a Postdoctoral Research Engineer at University of California, Berkeley

an additional level of indirection between applications and environment, and (iii) soft-state protocols for communication between processes.

The remainder of this paper elaborates on the design and implementation of Indiva. Section 2 describes the abstractions provided by Indiva. To illustrates how these abstractions are used, Section 3 shows two sample applications that can be built using Indiva. Section 4 discusses the implementation of Indiva. We evaluate the performance and ease of use of the system in Section 5. Finally, Indiva is compared with related work in Section 6, and we conclude in Section 7.

2. ABSTRACTIONS IN INDIVA

Before we proceed with a description of Indiva, it is important to understand the type of environment in which it is intended to be used.

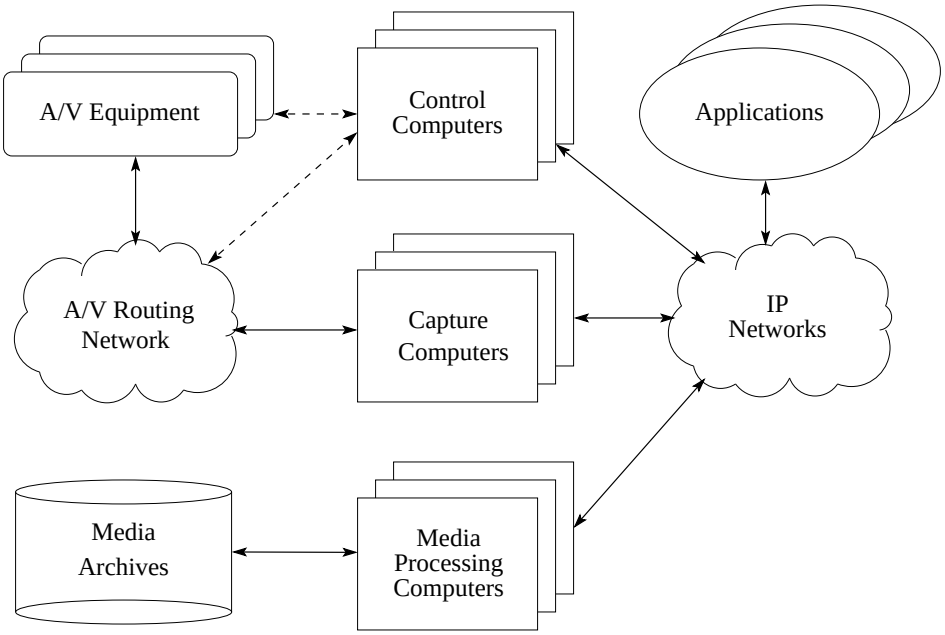

Figure 1. A Distributed Audio and Video Environment

Figure 1 shows a generalized model for a distributed audio and video environment. Such environments are common to many Internet webcast production and distributed collaboration systems. The environment contains *audio/video equipment* such as cameras, video tape recorders, special-effects processors, and scan converters. These equipment can be controlled through an RS232 or IR interface (shown as dotted lines in Figure 1) by *control computers*. The *capture computers* contain hardware and software for translating back-and-forth between *media signals* (generated by conventional audio/video equipments) and *media streams* (sequence of IP packets). The audio/video equipment and capture computers are connected using point-to-point transmission technologies (e.g., fiber, UTP etc.) through an *audio/video routing network*, This routing network is a circuit switch network that interconnects equipment and computers through a series of routing switchers, multiplexers, and demultiplexers that can be remotely controlled by the control computer as well. *Media processing computers* run software processes that manipulate the audio and video streams, such as transcoding,[9] mixing,[10] special-effects processing[11] and recording/playback.[12] Recorded media streams are stored in *media archives*. Finally, *applications* access and control the environment by sending RPC commands to the control, capture, and media processing computers over an IP network. For example, an application might want to pan or tilt a remote camera or change the encoding frame-rate of an encoder. These entities might be physically located in different rooms.

There are several different models for managing media streams in a distributed audio and video environment. In Indiva, we adopt the IETF Conference Model (sometimes called MBone Conferences) as the model for disseminating media streams because of its flexibility and compatibility with the AG. A *conference* is composed of one or more sessions. A *session* is composed of media streams of one type, for example audio or video, being delivered from one or more senders

to one or more receivers via IP multicast. For instance, a lecture webcast might be a conference with a video session and an audio session. The video session might contain two video streams, one shows the lecturer, and the other shows the presentation materials (e.g., slides). Meta-data for conferences and sessions (e.g., descriptions, IP address, owners, format) are stored in SDP format[13] and are periodically announced to a well-known multicast address.

Indiva is a middleware system between a distributed audio and video environment and applications. It provides a set of APIs for accessing and controlling the three types of entities in the environment: *hardware* (e.g., equipment, routing networks and video capture cards), *software processes*, and *data* (e.g., live media streams, archived media, and meta-data for conferences and sessions). The hardware, software processes, and data managed in the environment will be called *Indiva resources* in this paper.

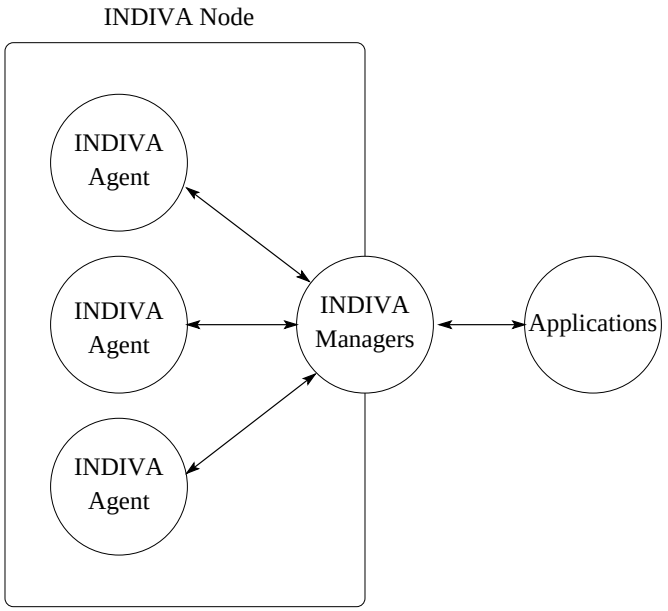

Figure 2. Indiva Software Architecture

Figure 2 shows the software architecture of the system. It is composed of a *manager* and a set of *agents*. The manager is a process that controls resources in an *Indiva node*. A node is an administratively defined domain of equipment, processes and data. For example, an AG node can be an Indiva node. The Indiva agents are processes running on the capture, control and media processing computers. Each process performs a specific task, such as recording media streams or controlling a specific piece of equipment. Each task performed by an agent is called a *service*. An agent may provide multiple services at the same time. The manager acts like a network file server, except that it manages resources in an Indiva node. The manager process is assumed to be up and running all the time, while agents are dynamically launched by the manager when needed. An application that wishes to manipulate resources in the environment invokes RPC commands on the manager. The manager decides how to execute the commands, and invokes one of the standard RPC APIs on the agents to complete the request. This architecture improves the modularity of the system, as new equipments and software processes can be added by writing new agents and reconfiguring the manager so that it can communicate with the agents. No modifications to applications is needed.

The rest of this section describes Indiva abstractions and high-level commands without going into details about how they are actually implemented. The implementation is described in Section 4.

2.1. Resource Naming

In order for an application to manipulate an Indiva resource at a node, the application must name or identify the resource. Indiva provides a hierarchical namespace for naming resources. Names can be grouped into hierarchy based on either logical grouping (e.g., physical location), or based on a “has a” relationship. A resource may appear in multiple places

in the hierarchy using links. Names for static resources, such as devices, are predefined by the Indiva node administrator. Names for software processes, conferences, and media streams are created on-the-fly when processes are launched, when a new SDP announcement is received, or when packets from a new source are detected, respectively. Users can also group resources into a customized hierarchy based on their interest.

For instance, to specify the composite output of a camera located in room 405 Soda Hall, one might use

```
/rooms/405soda/front.cam/composite.out
```

and to specify the audio stream in a lecture webcast, one might use

```
/confs/cs101.con/audio.ses/audio.rtp
```

There are several points to note in the above examples. First, we use UNIX file system syntax to name a resource. This similarity is intentional. As can be seen later, we are using a file system metaphor to manipulate the resources. As such, we also use the term *directories* to refer to resources that can contain other resources, and *files* to refer to resources in general. Second, we use extensions to indicate resource types: ".cam" for a camera, ".out" for an output port, ".con" for a conference, ".ses" for a session, and ".rtp" for an RTP stream. Third, directories not associated with any resources, such as `rooms`, `confs`, and `405soda` can be created for the purpose of organizing the namespace. Figure 3 shows an example of a simple namespace.

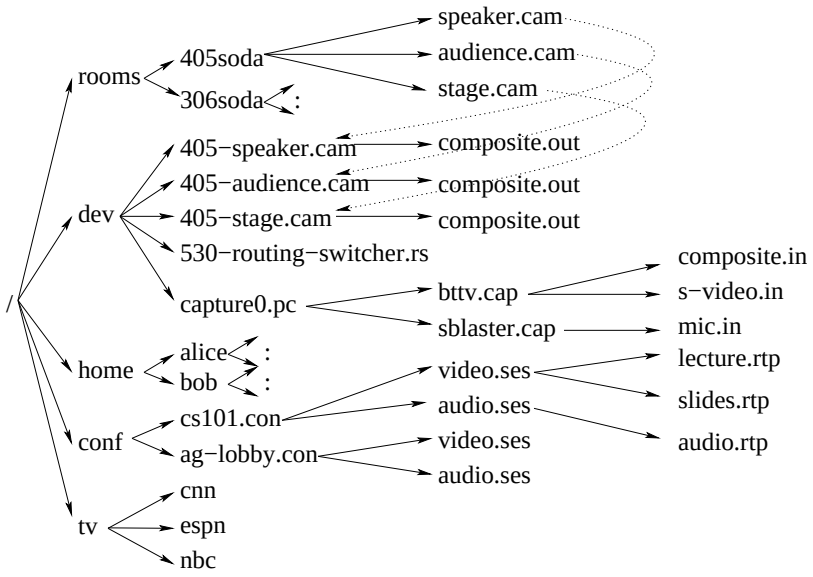

Figure 3. Name space in Indiva

Users of applications written in Indiva can have a "home directory" in the namespace. This directory allows users to define their own resources and link to resources that are of interest to them.

Indiva resources have an associated list of attributes, which is represented by key-value pairs. Different resource types have different sets of required attributes that must be defined. For example, every serial-controllable device must define an attribute with the key `host` that has a host name of its control computer as its value. In addition to the required attributes, customized attributes can be added by users or administrators for convenience.

2.2. Namespace Manipulation

The Indiva manager provides several commands for manipulating the namespace. These commands are being bound to RPC methods in applications. and many of them are similar to those on a UNIX file system. In particular, the `ls` method returns the contents of a directory, and the `find` method returns a list of resources whose attributes match a given list. These are useful methods for discovering resources in an Indiva node. For instance, invoking `ls` on a session returns the list of streams; invoking `ls` on a satellite dish returns the list of available TV channels; and invoking `ls` on a room returns

all devices in a room. To get all cameras looking at the speaker in all rooms, one can use `find` to search for all resources whose name matches "*.cam" and has an attribute `view` whose value is "speaker". Here, `view` is an example of a useful user-defined attribute. Attributes can be modified or added during run-time using the `configure` method. The Indiva manager also provides a method named `info` for inspecting attributes of a resource.

Other namespace manipulation methods include `mkdir`, `ln` and `mv`. `mkdir` creates a directory, `ln` creates an alias of a resource, and `mv` renames a resource. However, as described in the next section, `ln` and `mv` are also overloaded to manage media streams as well.

2.3. Media Streams Management

One novel feature of Indiva is the high-level abstractions for creating, manipulating and deleting media streams in a distributed audio and video environment. The Indiva manager provides the methods `encode`, `decode`, `play`, `record`, `ln`, `cp`, `mv` and `rm` for these purposes.

The `encode` method captures, encodes, and transmits a video or audio signal into a specified multicast session. The `decode` method decodes a specified stream from a multicast session and transmits it as a video or audio signal. Implementing these methods involves more than launching an encoding or a decoding process. In the case of `encode`, before the process can be launched, the manager must select a capture computer and activate the source of the signal. Then, the signal must be routed through the audio/video network to the selected capture computer. `encode` returns the name for the resulting stream in the namespace. By default, `encode` uses the RTP CNAME of the stream as its file name because it is unique among all participants within an RTP session[14] (An example file name is shown in Figure 4).

The properties of the encoded stream can be specified by passing optional arguments to `encode` and modified by using the `configure` method. Recall that `configure` can be used to modify attributes of a resource. In the case of media streams, `configure` may modify the properties of the video source, routing path, and encoding process as well. For instance, `configure` can be used to tilt the camera, switch to another camera or increase the frame-rate.

The `play` and `record` methods convert between live media streams and archived media streams, using RTSP[15] or Mars.[12]

The `mv` method moves a stream from one session to another. In this case, `mv` not only modifies the namespace, but it changes the environment to maintain the semantics of the namespace. Since the moved stream now belongs to a different multicast session, the Indiva manager must contact the encoding process of the stream and tell it to switch to a different destination address. Similarly, the methods `cp` and `ln` modify both the namespace and the environment. When a stream is copied or linked to another session, packets are forwarded to the target session. The difference between `cp` and `ln` is that the attributes of a new stream created by `cp` can be modified without changing the original stream, while changing the attributes of a stream created by `ln` modifies the attribute of the original stream. Consequently, the change is visible to all recipients of the stream. The semantics are analogous to their counterpart in the file system. Finally, a stream can be removed by the `rm` method. `rm` deletes a stream from the namespace, and stops the encoding process.

These methods further illustrate the file system metaphor in Indiva – we are treating media streams as files, and multicast sessions as directories. By manipulating the namespace, we can create, change, forward, and remove media streams in an intuitive way. These are all important operations in production control and automation applications. The operations are also valuable for someone dynamically controlling a desktop interface to a complex distributed audio and video environment. But the operations are difficult to implement without Indiva because so many details are required.

We note that even though Indiva provides high-level abstractions for manipulating resources, all resources in the environment are exposed to the users. A power user can manually select a capture machine (useful if other capture machines are shutting down for maintenance soon), use `route` method to send a video signal to it, and launch the encoder. We expect, however, that most users and applications will use the high-level abstractions because they are convenient and provide the required functionality.

3. INDIVA APPLICATIONS

Before we describe the implementation of Indiva in greater detail in the next section, two applications written using Indiva will be described. The first example is a text-based, interactive shell, called the Indiva shell, or `ish`. The second example is a GUI application for viewing video sources.

3.1. Indiva Shell

We wrote the Indiva shell for two purposes. First, we needed a simple application that calls the APIs provided by the Indiva manager for testing and debugging. Second, shell scripts can be written to automate repetitive tasks in producing a webcast or a collaborative session.

```
$ mount /bmrc otto.bmrc.berkeley.edu:9500
ish$ ls
devices  home  rooms  confs  archives
ish$ cd home/bob
ish$ ls
speaker.cam@  audience.cam@  stage.cam@
ish$ mkcon lecture.con
ish$ mkses lecture.con/v.ses -type video
ish$ cd lecture.con
ish$ configure v.ses -addr 224.4.4.4
ish$ configure v.ses -port 44444 -ttl 16
ish$ encode speaker.cam v.ses
v.ses/ive:8921@capture0.bmrc.berkeley.edu.rtp
ish$ record v.ses /bmrc/archives/today
```

Figure 4. An interactive session with `ish`.

`ish` is implemented as a Tcl shell extension, and provides a command for each of the methods provided by Indiva. In addition, it provides `cd` and `pwd` commands to explore the namespace, and `mount` and `umount` commands to connect and disconnect from an Indiva node. Figure 4 shows an example interactive session in `ish`. In this example, a user named Bob connects himself to an Indiva manager running on a particular machine at Berkeley. Bob then changes the current directory to his home directory `/bmrc/home/bob`, where he has created links to three cameras in a lecture room. To start a webcast, Bob creates a conference and a session under his home directory, configures the session to use multicast address 224.4.4.4 and port 44444. Finally, he uses the `encode` command to send the video from the camera pointing to the speaker into the multicast session and the `record` command to save a copy of the streams for archiving.

The next example, shown in Figure 5, illustrates how `ish` can be used to script repetitive tasks when preparing an AG node for remote collaboration. The script first turns on all three wall projectors. It then sends VGA output from the multi-headed display computer to the projectors and sends video and audio from the local AG node to the AG lobby. Finally, it launches a viewer to display all streams in the AG lobby. The viewer, called `iview`, is another example application built on top of Indiva. We briefly describe `iview` next.

3.2. Indiva Viewer

The Indiva viewer, or `iview`, is intended as a user-friendly GUI for viewing media sources in an Indiva node. `iview` accepts resource names as command line arguments, and renders video and audio streams of the given resources. Resources that can generate media signals (e.g., camera, TV tuner, scan converter), generate media streams (e.g. capture card), or contain media streams (e.g., conference, multicast sessions) can be used as arguments. We find this interface useful for previewing various media sources in our environment. For example, suppose we set up a directory `/tv` in the namespace that contains a list of TV channels available. To watch CNN on our desktop, we execute the command:

```
iview /tv/cnn
```

This causes `iview` to mount the default Indiva node (defined either in a configuration file or registry), call `mkcon` and `mkses` to create temporary multicast sessions, and call `encode` to send the video and audio signals to the temporary multicast sessions. `iview` then joins the sessions to receive, decode, and display the video and audio streams. In this example, the end-users just need to know the resource name `/tv/cnn` and do not need to care about whether the TV

```
#!/bin/ish
mount /ag heart.cs.berkeley.edu:9500
set room /ag/rooms/314soda
set conf /ag/confs/lobby.con
configure /ag/dev/proj1 -on
configure /ag/dev/proj2 -on
configure /ag/dev/proj3 -on
route $room/ctrl.pc/vga1.out /ag/dev/proj1/vga.in
route $room/ctrl.pc/vga2.out /ag/dev/proj2/vga.in
route $room/ctrl.pc/vga3.out /ag/dev/proj3/vga.in
encode $room/wide.cam $conf/video.ses
encode $room/left.cam $conf/video.ses
encode $room/right.cam $conf/video.ses
encode $room/audio.mix $conf/audio.ses
view $conf
```

Figure 5. An ish script for initializing AG environment.

signal is from a satellite dish, a settop box or a TV tuner card, nor do they need to know about signal routing and the encoding process. This command would still work if the environment has changed.

Another useful feature in `iview` is drag-and-drop. Dragging and dropping a video from one session to another executes an RPC call to the appropriate Indiva manager methods to either move, link, or copy the streams. The semantics depends on whether keyboard modifiers CTRL or ALT are being pressed, similar to a traditional desktop file manager. This feature provides an easy-to-use interface for manipulating media streams in MBone sessions. Such manipulations are common in a webcast production. A webcast director can run

```
iview /rooms/soda405/*.cam
iview /confs/lecture.con/video.ses
```

to create two windows, one for the temporary MBone session that contains video streams from all cameras in room 405 Soda Hall, and another for a live video session being broadcast onto the Internet. The director can preview the cameras, and use drag-and-drop to forward interesting streams onto the Internet. Moreover, the director can easily incoorperate sources from different locations (e.g., rooms) in the production.

The two examples shown in this section demonstrate the versatility and usefulness of the abstractions provided by Indiva. In the next section, we will describe the underlying mechanism used to implement the system.

4. IMPLEMENTATION

Indiva is written using Tcl[16] and the OpenMash Toolkit[17] for rapid prototyping. Using OpenMash also allowed us to leverage existing code in the MBone tools. For instance, many of our agents are simply wrappers around existing encoders, packet forwarders, and device drivers. These two themes underscore other implementation decisions as well.

The remainder of this section describes how resources are maintained and how high-level APIs are implemented in Indiva.

4.1. Namespace Implementation

The Indiva namespace is stored in a file system. A directory in the namespace is represented by a directory in the file system, plus a hidden text file in that directory. A text file in the file system represents a file in the namespace. Resource attributes are stored in the text files as a Tcl list. Figure 6 shows two examples of such files.

6 (a)
/usr/lib/indiva/rooms/soda405/speaker.cam/.info

```
camera {
    view speaker
    company Canon
    model VCC-4
    hostname 405ctrl.bmrc.berkeley.edu
    path /dev/cuac00
    defaultout composite.out
    location {Soda 405}
    friendlyname {Audience Camera in 405 Soda}
    norm ntsc
}
```

6 (b)
/usr/lib/indiva/rooms/soda405/speaker.cam/video.out

```
outport {
    type composite
    to ../knox.rs/v07.in
}
```

Figure 6. Example resource configuration files.

Administrators of an Indiva node can add a resource by simply creating a text file with the appropriate attributes. A resource can be removed either by deleting the file on the file system or hiding a file (i.e., prefix it with "."). Hiding a file is useful for "commenting" out a resource in Indiva.

Other implementations are possible: for example, we considered storing the namespace in an LDAP server[18] or in a database. We chose to use a file system for its convinience, simplicity and flexibility in our prototype, as features such as soft links and glob-style pattern matching are readily available. We also considered running discovery protocols so that new resources can be discovered by the Indiva Manager automatically. We decided instead to store all hardware information statically in a centralized location for ease of administration. It was also easier to implement. This decision can be easily changed. Moreover, certain resources such as conference meta-data and archived media are currently being discovered during run-time by listening to an SDP channel or talking to an RTSP server respectively.

One important concept to note in Figure 6(b) is the attribute `to`. The value of this attribute is the name of the resource to which video.out is connected. In this example, it is a port named `v07.in` (i.e., input port number 7) on a routing switcher named `knox.rs`. This attribute is used by the Indiva manager to construct a graph that represents the routing network in the distributed audio and video environment.

4.2. A/V Graph

The Indiva manager maintains information about the routing network as a directed graph called the *A/V Graph*. The vertices in the graph are resources. There is an edge from a vertex u to vertex v, if it is possible to send a media signal or stream from u to v. Hence, there are edges between any pair of ports that are physically connected through cables, and between all input and output ports in a routing switcher. There are also edges from input ports of capture cards to multicast sessions and from multicast sessions to output ports of display cards.

A naive representation of the A/V Graph will result in quadratic number of edges and is neither space nor time efficient. For example, a routing switcher that has 256 inputs and 256 outputs requires 65536 edges if every connection is represented

explicitly. Fortunately, the A/V Graph contains many complete bipartite subgraphs that can be compressed efficiently.[19] The graph compression works as follows: for each complete bipartite subgraph $B = (U, V, E)$, replace it with $B' = (U, V, w, E')$, where w is a new vertex and E' consists only of edges that connects $u \in U$ to w and edges that connects w to $v \in V$. Using this technique, we reduce the number of edges in a complete bipartite subgraph from $|U| \times |V|$ to $|U| + |V|$. However, in the rest of the discussion, we will describe the Indiva manager implementation using an uncompressed graph for clarity.

4.3. Active Service Framework

Indiva adopts the Active Service Framework (AS1)[20] as the framework for managing Indiva agents. The AS1 Framework uses a request/respond, soft-state protocol to allocate a host on a cluster of servers to run a process in a scalable and robust manner. Here is a brief description of how it works: Clients and servers communicate through a multicast channel g. A client c that wishes to launch a process p on the cluster sends a launch(p) message to g. Each server in the cluster runs a process called the *host manager* (HM). HM listens to g for launch messages. Upon receiving a launch message, it sets a random timer T. When T expires, HM executes p and sends an announcement to g indicating that p has been launched. However, if HM receives an announcement that another HM has launched p before its timer expired, it will suppress its own timer to avoid duplicate launches of p. The use of randomized timers distributes processes randomly across hosts in a cluster. By biasing the timer using current CPU load on a host, we can achieve reasonable load balancing among the hosts. Robustness is built into the protocol, as only soft-states are maintain in AS1. c and p periodically exchange heartbeat messages to indicate that c is being served by p. If c stops receiving the heartbeat message from p, c re-launches p. If p stops receiving the heartbeat message from c, p exits.

In our case, the Indiva manager is the AS1 client. The capture, media processing, and control computers form a cluster of hosts for launching Indiva agents. We chose AS1 as the framework for managing agents because it is robust and simple. Moreover, it allows hosts and agents to be added or deleted from the environment without modifications to the Indiva manager source code. However, we made several modifications to AS1. First, not all Indiva agents can run on all hosts. For example, an agent that captures and encodes video signals can only run on hosts with video capture card. An agent that controls a device can only run on hosts with a physical interface to the device. We modified AS1 to include a precondition in the request message. Only hosts that satisfied the specified precondition respond to the request. Second, we decouple the launch message into two new message types, select and launch-now. The select message is exactly the same as launch, except that when the timer expires, the host replies with an acknowledgement without actually launching the agent. This response is used by the Indiva manager to select a host to run the agent. The second message type, launch-now, tells the selected host to launch the agent. These two messages are used when deciding which path to use in the A/V Graph to route a media signal.

4.4. Routing Algorithm

One of the challenges we faced when implementing Indiva is to choose an appropriate model for routing signals and streams through the A/V Graph. Simply selecting a path with the least number of edges from a source to a destination does not work well as there are usually many such paths between them. Factors such as load balancing and sharing of services should be considered in deciding the best path. This often requires distributed states (e.g., loads of the machines) or non-trival cost functions on the edges. This subsection describes the routing algorithm we designed, using the stream management methods `encode`, `mv`, `cp`, `ln` and `rm` as illustrative examples.

To send media signals or streams from a source s to a destination t, Indiva creates an object called a *flow*. A flow $f(s, t)$ consists of a path from s to t in the A/V Graph, and unique identifiers to Indiva agents performing services on edges along the path.

Flow creation is a two-stage process. In the first stage, the Indiva manager searches through the A/V Graph for a set of shortest paths between s and t. A subgraph that contains the shortest paths is returned. The manager traverses the subgraph from t to s. If there is more than one incoming edge to the current vertex, the Indiva manager decides which incoming edge to include in the path by sending an AS1 select message to the hosts, so the HMs can make resource allocation decisions using their local states. At the end of this stage, we obtain a shortest path from s to t. During the second stage of the algorithm, the Indiva manager traverses the selected path from s to t, and sends RPC commands to the Indiva agents (launching them with an AS1 launch-now message if necessary) to send media signals and streams along the edges in the path. In our implementation, we maintain a table of function pointers for each type of edge, pointing to code that causes

data to flow along that edge. As an example, an edge between input port i and output port j of a routing switcher points to a function that queries the port numbers of the ports incident to the edge (i.e., the values of i and j), and invokes RPC commands on the Indiva agent that controls that routing switcher, to send input i to output j. A flow maintains the unique identifier to these agents along the edges, as applications might need to send further commands to these agents later. A flow will be re-created automatically if we stop receiving heartbeat messages from one of these agents.

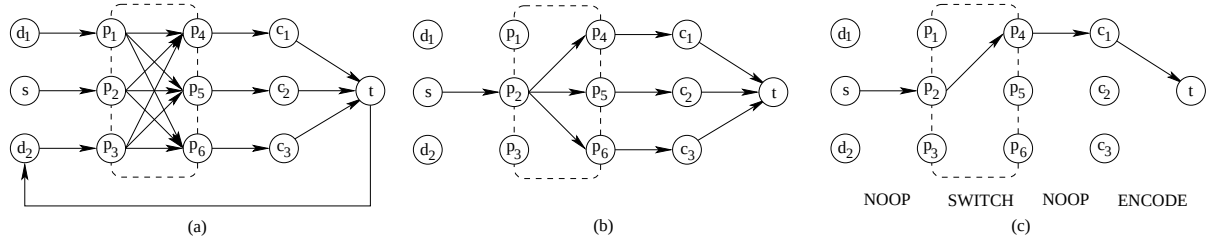

Figure 7. Adding A Flow

Figure 7 illustrates flow creation through an example. Figure 7(a) shows a simple A/V Graph with 3 video sources, denoted d_1, d_2 and s. Each of these video sources sends their output to a 3x3 routing switcher. The 3 outputs from the switcher are each connected to a capture computer, which can send encoded signals to multicast session t. One of the video sources, d_2, is capable of decoding streams from the multicast session. There are three possible shortest path from s to t, as given in Figure 7(b). To pick a path, the Indiva manager traverses the graph in 7(b) from t to s. Since there are three incoming edges into t, namely (c_1, t), (c_2, t), and (c_3, t), the Indiva manager sends an AS1 `select` message to the capture computers c_1, c_2 and c_3. Suppose the manager receives an acknowledgement from c_1, the edge (c_1, t) will be added to the path. The manager subsequently traverses the vertices p_4, p_2 and s, adding the remaining edges on the path from s to c_1 to the flow since they have no more than one incoming edge. To complete flow creation, the Indiva manager traverses the path (s, p_2, p_4, c_1, t) and calls the functions associated with the edges to send media signals or streams along the path. In this example, the Indiva manager sends a request to the Indiva agent that controls the routing switcher to switch input 2 to output 1 when it traverses edge (p_2, p_4), and starts an encoder process that captures a video signal from c_1 and sends it to t as it traverses through (c_1, t).

Other high-level commands can be implementated by manipulating flows. A flow $f(s, t)$ can be split by creating a new flow $f'(s, t')$ with the same source but a different destination. f and f' may share some services if they flow along the same edges. `ln` and `cp` are implemented by flow splitting. The Indiva manager can also redirect a flow (i.e., change the destination of a flow to cause it to flow along a different path). This operation is the basis for implementing `mv`. Finally, a flow can be deleted. Deletion involves traversing a path and sending RPC commands to the agents to stop its service.

When `configure` is called on a media stream, the arguments to the `configure` method are passed to the agents in the flows. Each agent responds to the arguments it recognizes. This approach allows a user to modify the properties of a stream without knowing implementation details.

The behavior of the routing algorithm can be modified by assigning a cost to each edge in the A/V Graph. For example, for access control, we can assign an ∞ cost to edges in a flow after the flow is established. As a result, no other users can share the edges in the flow. However, different situations and applications may require a different policy. In certain cases, sharing between users should be promoted. Consider the case where an encoding process is already running and transmitting video from CNN. If a second user wishes to view the same channel, we should reuse the same encoding process if possible. In this case, we want to assign a zero cost to edges in that flow. Indiva handles these two cases by identifying which type of resources can be shared for reading and writing. For instance, video and audio output ports can be shared for reading, and sessions can be shared for both reading and writing. On the other hand, an RTSP archive stream cannot be shared for writing.

It is possible for an Indiva manager to deadlock attempting to satisfy commands from two clients at the same time because resources are allocated incrementally and the flow creation algorithm might allocate segments in different orders. The Indiva manager uses a priority mechanism to break deadlocks when two commands are attempting to allocate resources at the same time. If the requesting client has a higher priority, the resource is pre-empted and allocated to the requesting

client. The earliest command received aborts later commands if both requests have the same priority. This priority-based mechanism is required, for example, when two users request access on a camera. Higher priority can be assigned to webcast producers over normal users to ensure that the required resources are always available to produce a live webcast when needed.

5. EVALUATION

Indiva is currently in production state. We have setup two Indiva nodes, one for an Internet webcast infrastructure and one for an AG node. We have implemented several agents, including an RTP video encoder, an RTP audio encoder, and controllers for routing switchers, a Miranda Kaleido multi-image display system, and an EchoStar satellite dish. We are currently working on Indiva agents to control pan/tilt cameras and projectors. A web service interface using SOAP is also planned so that Indiva can be used by applications written in languages other than Tcl.

During the implementation of the prototype system, we did not focus on performance. The main reason is that the latency for calling a method is typically dominated by the latency to control the equipment (see Table 1). For example, switching a channel on a satellite dish requires at least 1 second because we need 0.5 seconds interval between sending each digit of a channel number to the dish. Furtheremore, our decision of implementing Indiva using a scripting language for rapid prototyping causes some overhead in the performance. Our measurements show that Indiva can take between 0.5 - 5 seconds to execute a command, depending on the length of the path in A/V graph and the number of equipments involved. While these numbers are not impressive, they are adequate for the applications we have in mind, and they are still order of magnitudes faster than performing the tasks manually.

Step	Relative Time
Finding set of shortest paths	13%
Selecting a path using AS1 select	12%
Switching channels on satellite dish	44%
Switching route on routing switcher	2%
Launching encoder	18%
Starting encoder	11%

Table 1. Time taken to perform the command “encode cnn”

To verify our claim that Indiva simplifies application development in a distributed audio and video environment, we developed several applications using Indiva. `iview`, described in section 3.2, is composed of 1800 lines of Tcl/Tk and OpenMash code. However, most of this code deals with creating the GUI interface and displaying video streams and information about them. There are less than 10 lines of code that actually manages streams and encoding processes.

One of the design goals of Indiva is to make extending an Indiva node simple. There are three cases to consider: adding a new resource of an existing type, adding a new agent, and adding a new resource type. As mentioned, adding a new resource of an existing type requires adding a text file with appropriate attributes. We provide templates for various resource types. An administrator can copy the template to the approrpiate location, rename it, and edit the attributes. Alternatively, we provide a GUI resource editor for creating, modifying and visualizing resources inside Indiva to simplify the process further. Adding a new agent is also relatively easy, as much of the common functionalities of agents are already abstracted into library classes. Writing a new agent involves providing a standard wrapper RPC API for creating a new service, deleting a service and reconfiguring a service. Most of our existing agents are either wrappers around device drivers for controlling devices or existing media stream processing code. Table 2 lists available agents and the number of lines of uncommented wrapper code written. In the rare case that a new resource type needs to be added, modification to the Indiva manager is needed. Again, most common functionalities are abstracted into library classes. To define a new resource type, the developer only needs to create a subclass for the `IndivaResource` class, define its extension, type, and set flags to indicate if the resource can be shared for reading and writing. All current resource types requires less than 50 lines of code to implement. During development of Indiva, we are able to add a new resource type in the order of minutes. This experience illustrates that minimal effort is needed to extend Indiva and is a huge improvement over our previous experience, where we often had to spend up to a few days to adapt existing tools to a new environment.

Agent	Lines of code
video encoder	390
audio encoder	160
Kaleido controller	150
Routing switcher controller	110
Packet forwarder	140

Table 2. Indiva Agents.

6. RELATED WORK

Several research groups have implemented software to control equipment for a specific application (e.g., video conferencing,[21] broadcast automation,[22] and presentation control[23]). These systems are excellent solutions for the particular application, but they are too limited in functionality. In contrast, Indiva is an open, extensible middleware system that provides more functionality and can be easily re-used for many applications.

Specialized audio and video equipment control system, such as AMX, Crestron, and SmartHome, provide APIs for device control. Indiva differs from these system by providing an API not only for controlling devices, but for managing software processes and media streams as well. Furthermore, these specialized control systems require additional hardware and are less flexible. We note that Indiva does not preclude the use of these control systems in the environment. In fact, in our prototype environment, we have an AMX controller that can be controlled through Indiva.

Jini,[24] UPnP,[25] and Salutation[26] are architectures for discovery, access, and control of devices and software services. Indiva differs from this work in a few ways. First, these architectures are meant for networked devices that support their particular protocols. They are not suitable for environment that must intergrate off-the-shelf audio/video equipment, most of which do not currently understand these protocols. Second, the primary contribution of Indiva lies in its easy to use, high-level abstractions for manipulating hardware devices, software services, and media data, rather than the lower-level discovery, event handling, and presentation protocols emphasized by these architectures.

Ninja[27] is an architecture for building a large-scale, secure environment for Internet services. Ninja defines a mechanism for composing Internet services by searching for a path through available services. This concept is very similar to the flow creation process in Indiva, except that Indiva flows can pass through hardware devices and software services, whereas Ninja path creation is restricted to software services only.

HAVi[28] is a proposed standard for networking home entertainment devices defined by several major electronics companies. HAVi provides Java API for stream management and device controls and shares some goals with Indiva. However, HAVi is targeted at consumers home audio/video network and dictates the use of Firewire as a transport mechanism. Indiva is meant for the Internet and supports media transmission through an IP network. It is independent of the point-to-point signal transport mechanism. As a result, Indiva can support many transport mechanisms.

7. CONCLUSION

This paper described a middleware system for a distributed audio and video environment called Indiva. The system provides a single, logical view for manipulating audio/video equipment, media processing services, conference meta-data, live media streams, and archived media. Easy-to-understand abstractions based on a file system metaphor are provided for application developers. The goal is to hide low-level details in the environment from applications, thereby allowing programmers to focus on building high-level functionality such as control automation and end-user interfaces. Indiva also promotes code reuse by minimizing the coupling between application and environment. We believe that Indiva has other applications outside a distributed audio and video environment. For example, similar middleware could be useful in SAN or CDN management.

The discussion in this paper describes manipulation of resources within a single Indiva node. However, applications can connect to multiple Indiva nodes and manage resources across administrative boundries. We are currently looking at issues that deal with inter-node operations, such as establishing flows between multiple Indiva nodes. Another issue we are currently studying is building user-level access control on top of Indiva resources. The current solution for restricting access to a multicast conference is to encrypt the media streams using a key that is known to its intended participants only.

This solution could be done transparently using Indiva since its file system metaphor provides a natural abstraction over access control using file permission.

8. ACKNOWLEDGEMENT

The National Science Foundation Grant ANI-9907994 supported this research.

REFERENCES

1. L. Childers, T. Disz, R. Olson, M. Papka, R. Stevens, and T. Udeshi, "Access Grid: Immersive group-to-group collaborative visualization," in *Proceedings of the 4th International Immersive Projection Technology Workshop, 2000*, (Ames, Iowa), June 2000.
2. Access Grid, "AG nodes." Web Site http://www-fp.mcs.anl.gov/fl/accessgrid/ag-nodes.htm.
3. M. Banchi, "AutoAuditorium: A fully automatic, multi-camera system to televise auditorium presentations," in *Proceedings of Joint DARPA/NIST Smart Spaces Technology Workshop*, (Gaithersburg, MD), July 1998.
4. E. Machnicki and L. A. Rowe, "Virtual director: Automating a webcast," in *Proceedings of the SPIE Multimedia Computing and Networking 2002, Vol. 4673*, (San Jose, CA), Jan. 2002.
5. S. Mukhopadhyay and B. Smith, "Passive capture and structuring of lectures," in *Proceedings of ACM Multimedia 1999*, (Orlando, FL), Oct. 1999.
6. Y. Rui, L. He, A. Gupta, and Q. Liu, "Building an intelligent camera management system," in *Proceedings of ACM Multimedia 2002*, (Ottawa, Canada), Oct. 2002.
7. T. Yu, D. Wu, K. Mayer-Patel, and L. A. Rowe, "dc: A live webcast control system," in *Proceedings of the SPIE Multimedia Computing and Networking 2001, Vol. 4312*, (San Jose, California), Jan. 2001.
8. D. Wu, A. Swan, and L. A. Rowe, "An Internet MBone broadcast management system," in *Proceedings of the SPIE Multimedia Computing and Networking 1999, Vol. 3654*, (San Jose, CA), Jan. 1999.
9. E. Amir, S. McCanne, and Z. Hui, "An application level video gateway," in *Proceedings of ACM Multimedia 1995*, pp. 255–266, (San Francisco, CA), Nov. 1995.
10. W. T. Ooi, R. van Renesse, and B. C. Smith, "The design and implementation of programmable media gateways," in *Proceedings of NOSSDAV'00*, (Chapel Hill, North Carolina), June 2000.
11. K. Mayer-Patel, *Parallel Software-Only Video Effects Processing System*. PhD thesis, Department of Computer Science, University of California, Berkeley, 1999.
12. A. Schuett, R. H. Katz, and S. McCanne, "A distributed recording system for high quality MBone archives," in *Proceedings of Networked Group Communication, NGC'99*, pp. 126–143, (Pisa, Italy), Nov. 1999.
13. M. Handley and V. Jacobson, "RFC 2327: SDP: Session description protocol," Apr. 1998.
14. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RFC 1889: RTP: A transport protocol for real-time applications," Jan. 1996.
15. H. Schulzrinne, A. Rao, and R. Lanphier, "RFC 2326: Real time streaming protocol (RTSP)," Apr. 1998.
16. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.
17. S. McCanne et. al., "Toward a common infrastructure for multimedia-networking middleware," in *Proceedings of NOSSDAV'97*, pp. 39–49, (St. Louis, Missouri), May 1997.
18. M. Wahl, T. Howes, and S. Kille, "RFC 2251: Lightweight directory access protocol (v3)," Dec. 1997.
19. T. Feder and R. Motwani, "Clique partitions, graph compression and speeding-up algorithms," in *Proceedings of 23rd ACM Symposium on Theory of Computing*, pp. 123–133, (New Orleans, Louisiana), May 1991.
20. E. Amir, S. McCanne, and R. Katz, "An Active Service framework and its application to real-time multimedia transcoding," in *Proceedings of ACM SIGCOMM*, pp. 178–189, (Vancouver, Canada), Aug. 1998.
21. M. Perry and D. Agarwal, "Remote control for videoconferencing," in *Proceedings of the 11th International Conference of the Information Resources Management Association*, (Anchorage, AK), May 2000.
22. S. J. Angelovich, K. B. Kenny, and B. D. Sarachan, "NBC's genesis broadcast automation system," in *Proceedings of the 6th Annual Tcl/Tk Workshop*, (San Diego, CA), Sept. 1998.
23. T. Hodes and R. Katz, "Composable ad hoc location-based services for heterogeneous mobile clients," *ACM Wireless Networks Journal* **5**, pp. 411–427, Oct. 1999.
24. Sun Microsystems, "Jini technology specification." http://www.sun.com/jini/specs/.

25. Microsoft Corporation, “Universal plug and play device architecture.” http://www.upnp.org/.
26. Salutation Consortium, “Salutation architecture specification v2.0.” http://www.salutation.org/.
27. S. D. Gribble et. al., “The Ninja architecture for robust internet-scale systems and services,” *Computer Networks (Special Issue on Pervasive Computing)* **35**, pp. 473–497, Mar. 2001.
28. R. Baier, C. Gran, A. Scheller, and A. Zisowsky, “Multimedia middleware for the future home,” in *Proceedings of International Workshop for Multimedia Middleware (M3W’01)*, (Ottawa, CA), Oct. 2001.