MediaFlow,
A Framework for
Distributed Integrated Media

Conal Elliott, Greg Schechter, and Salim Abi-Ezzi
SunSoft, Inc.

March, 1994

1. Introduction

We use the term *integrated media* to encompass both modeled media, and recorded and live natural media. We believe that an integrated media framework is important because natural and modeled media are complementary and are most useful when integrated. Natural has the property of vividness (since nature is so rich), while modeled has the property of compactness and malleability (since computers are so flexible). While existing multimedia systems have little, if any, support for modeled media, omitting it is abandoning what computers do best, namely computing, mixing, and enhancing. Examples of the integration of natural and modeled media include: mixing of multiple audio streams, providing volume control to an audio stream, giving a varying spatial locality to an audio stream, and synchronizing and overlaying a geometric animation over a video stream.

MediaFlow supports modeled and natural media and the spectrum in between. It is based on the concepts of high-level data types and of omni-grain continuous dataflow. (By “continuous”, we mean simply that time is indefinitely refinable.) We treat visual, audible, and gestural flows uniformly. The result is a design that provides uniformity, minimality, and orthogonality and hence the ease of programming integrated media applications.

MediaFlow provides the programmer with three distinct advantages. First we have hidden from the programmer details concerning the representation (data formats) of the media types. Sound to a programer is an abstract data type which, when sent to a sound presenter, will be played. Second, the programmer view of time is a continuous one. Hence two separate media flows can be coordinated using real arithmetic and without regard to their specific rates, which provides for rate decoupling between different media components, simplifying the problem of synchronization. Third, media distribution is supported by a conceptually simple mechanism of exporting and importing media flows between processes, leaving choice of external representation and transport up to the implementation.

A fundamental principle of the design of MediaFlow is a strong separation between the conceptual model, as presented to the programmer, and the implementation model. The conceptual model is designed for ease of use, emphasizing simplicity, uniformity, and power of expression. The implementation model, on the other hand, deals with issues like efficiency, data representation, transport, frame rates, etc.

This paper contains a discussion of the conceptual and implementation models and illustrates our design with numerous examples that we have selected as representatives of the needs in distributed integrated media applications.

MediaFlow is an evolution of a system called TBAG for supporting high-level specification of interactive 3D graphics applications, developed at SunSoft and described in [Elli94, Sche94], together with ideas from previous multimedia framework designs, such as [HIS93].

2. Programmer's Model

2.1 Metaphors in Software Systems

Metaphors help us to solve computational problems by allowing us to predict the behavior of one system, based on the behavior of some other idealized system more compatible with our habits of thought. A computer user gets her computational needs done as though she were manipulating and using objects on a desktop. Her desktop model helps her to remember how to access basic operating system services. A financial planner is accustomed to working with tabular information. As a consequence, he finds the spreadsheet metaphor helpful to his problem solving. Therefore, having a rich metaphorical content is an important characteristic of effective software systems.

We are interested in helping programmers develop computerized multimedia software. It seems therefore helpful to examine the mental models of people who deal with media of various kinds, and to extend these models to ones that take advantage of the flexibility of computerized media. This approach carries on a tradition that has led to spreadsheets, WYSIWYG text processors, and desktop GUIs.

2.2 The Basic Model

MediaFlow's underlying metaphor is based on *continuous flow* of media among *components* via *station*s. The components include information gathering devices (microphones, cameras, etc.), information presenting devices (projectors, loud speakers), recording devices (vcr, tape recorder), playback devices (turntables, tape decks), and information-transforming devices (amplifiers, clippers, squelchers). In addition, a number of control devices (knobs, dials, etc.) help to dynamically alter certain characteristics of the other devices (volume, gain, sensitivity, etc.). A transmitter/receiver paradigm is used for communication between *components*. *Stations* decouple transmitter and receiver components from each other.

Stations are specific to the type of media data that they encapsulate, so that it is not possible to connect a component to a station emitting an incorrect type. This means that fewer errors occur when connecting components.

Once interconnected, a set of components can be manipulated in certain ways. For example, the gain of an amplifier can be adjusted, the pitch of a music synthesizer can be increased, and color intensity can be modified. Also, in some systems it is possible to scale time as is done through slow motion and fast forward controls on a VCR.

The general notion of time manipulation is very useful in developing media systems. For example, in a slide presentation with an audio narration, it becomes straightforward to speed up the presentation and to deliver it in a fraction of its nominal time. Each slide is projected for a proportionally reduced period of time and the audio narration is sped up (scaled down) accordingly. We take a special note of the fact that time manipulation is easier to think about if the flow of time is continuous rather than discrete samples per second. In the above example, it is easier to express continuous scales for the audio narrations, than to express them in terms of integer arithmetic. Also, in a world of different media devices that operate on discrete streams of different rates, a continuous time model becomes a unifying principle that simplifies integration and synchronization.

Finally, we observe that media computations are intrinsically distributed, simply because media is about communication. From the point of view of configuring multimedia systems, the media specialist is not concerned with the physical location of devices in a multimedia system — only in the topology (i.e., what is connected to what).

2.3 Supporting the Basic Model

MediaFlow adopts a dataflow paradigm to support the multimedia metaphor presented above. An application program constructs networks of components and stations. The components continuously gather input from stations, perform computations, and write output to other stations. The approach has the following specific features:

- The conceptual model takes a fundamentally *continuous* approach to media, in accordance with the metaphorical model in which time is indefinitely refinable. This continuous approach considerably simplifies the programmer's conceptual model, freeing her from the complexities imposed by the many different sampling rates that occur in the hardware devices and drivers involved in an integrated media configuration. Moreover, as described in the next chapter, the continuous approach can be efficiently implemented.
- A single dataflow network may involve physical devices attached to more than one computer, and thus the implementation will be distributed.
- It is easy to incorporate existing software libraries into dataflow computations. A tool provided with MediaFlow converts existing interface specifications (currently, C / C++ header files) into classes and functions that construct and connect dataflow networks. Execution of networks results in calls to the original functions. Note that *the original library functions are written to operate on basic values and know nothing at all of dataflow, let alone continuousness.*
- Values of any type may participate in dataflow computations, e.g., sound, images, real numbers, colors, points, and geometric shapes. Moreover, it is not just *possible*, but *easy*, to make pre-existing types and operations available in dataflow networks.
- The dataflow paradigm is applicable at all levels of granularity, from adding two numbers or doing a single image operation, to connecting complex functions implemented as large system modules. MediaFlow is intended to efficiently and conveniently support these extremes as well as everything in between. This uniformity simplifies the programmer's job, since it allows an application to be constructed using a single paradigm and API.

- Purely computational (algorithmic) nodes have no inherent notion of *location*. The implementation can thus transparently map the specified dataflow network onto actual networks of machines, processes, threads, and communication channels, reducing communication costs where possible.

The notion of *station* is an abstraction inspired by radio or television stations. At any moment, one component (at most) may be *sending* to a station, while any number of other components may be *receiving* from it. We draw stations as circles with connecting arrows, as shown in Figure 2-1 below.

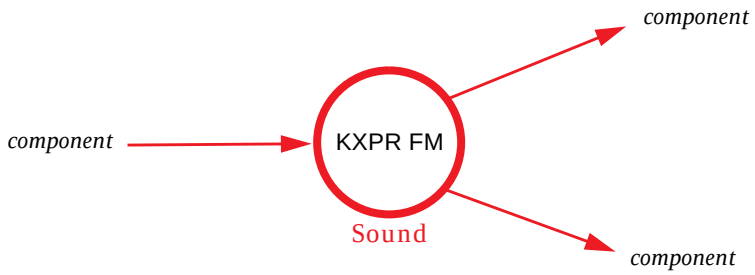

Figure 2-1 Sound-valued station

The sender and receivers may be completely unaware of the others' location, or even existence. From time to time, a station's sender may change without disruption to the receivers.

A *component* is an object having some number of input and output stations, generally shared with other components. Components get values from their input stations, compute functions of these values, and pass the results to their output stations for use by other components. As usual with dataflow, the functions do not modify their input values. This discipline of avoiding side-effects lends itself to easier multithreading and multiprocessing, as well as keeping the semantics straightforward.

2.4 Supporting Media Distribution

MediaFlow provides support for distribution of media to the application programmer by allowing stations to be *exported* from one process and *imported* into another process. Conceptually, the resulting station is identical to the original station. As we will see in Section 3.2.4, "Station Cloning", the exportation/importation of a station results in a *cloning* of the original station as a new station in the other process.

Exporting and importing of stations are managed by two functions, `export_station()` and `import_station()`. `export_station()` takes a station in the local process and returns a machine-independent token that serves as a handle for the station. `import_station()` takes such a token and turns it back into a corresponding clone station in the process in which it is executing.

Applications use this facility by identifying a set of stations that they want to make available to remote processes, creating tokens for these stations via `export_station()`, and distributing these tokens to other processes via any number of communication mechanisms, such as ToolTalk®, CORBA, NCSA's Mosaic, etc.

Other processes then find these tokens and use `import_station()` to create station clones in their local process. The application will then use these new stations during the construction of a MediaFlow network. It is the MediaFlow implementation's responsibility to establish media transport between a station and its clone.

3. Integrated Media Examples

The following examples briefly illustrate our basic approach.

3.1 Simple Connection

The first example simply connects a microphone on a remote machine to a speaker on a local machine.

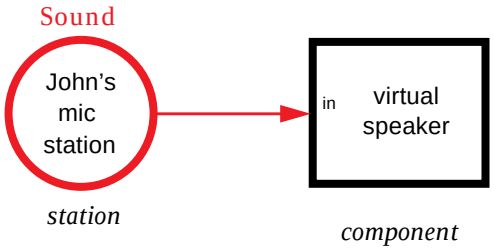

Figure 3-1 Simple sound connection

This simple network can be constructed as follows:

```
new virtual_speaker_Component(import_station(johns_mic_token));
```

Some comments:

- `johns_mic_token` is a sound-valued-station token exported from John's machine, which is connected to a microphone component. It is the responsibility of the remote application to provide the reference to the sound-valued station.
- `import_station()` converts a station token into a station, as described above.
- The class `virtual_speaker_Component` has a constructor that takes a sound station and makes a new virtual speaker component. The new virtual speaker component shares the physical speaker with other virtual speakers by mixing.
- Media flow begins immediately.

3.2 Nonphysical Components

In Figure 3-2, we add in a volume adjuster component that takes sound flow from a microphone, and a real number flow from a dial, and produces a sound output flow, which is the input sound made quieter or louder, according to the input number flow.

Some comments:

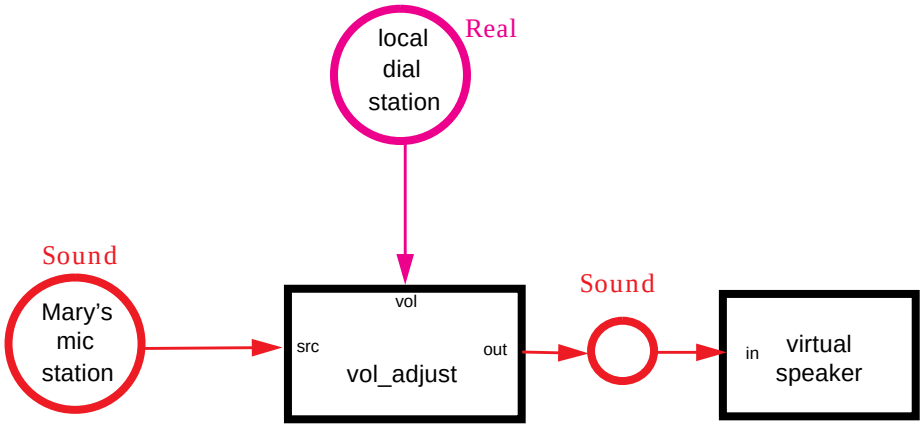

Figure 3-2 Volume adjuster

- In this example and in the ones below, one can replace the dial device with any source of real-valued flow, e.g., a clock, the distance from the mouse's position to the center of a window, or some aspect of an animation like the distance of an ambulance to the viewer.
- Motion is media, i.e., n-dimensional position flows are treated exactly like sound and image flows. *All types* are supported in this way.
- Components can be purely algorithmic, as in the volume adjuster. The functions on which these purely algorithmic components are based are programmed to operate on basic values (in this case, sound and numbers), without any knowledge of dataflow.

Code for constructing this example goes much as in the previous example. The nonphysical component is created by instantiating a class that is automatically constructed, as discussed in Section 3.5.

```
Station<Sound> adjusted_sound;    // to hold volume-adjusted sound

// Make new volume adjuster component
vol_adjust_Component *voladj =
    new vol_adjust_Component(marys_mic_station,
                             local_dial_station,
                             adjusted_sound);

// Plug into a new virtual speaker.
new virtual_speaker_Component(adjusted_sound);
```

Note that dataflow components are represented by objects that have statically typed input and output ports as instance variables.

3.3 Dataflow Expressions

As dataflow networks grow, it becomes cumbersome to create all the components involved and make explicit connections among them. As a short-hand, we will automatically generate function overloadings that take input stations as arguments, create new components, make connections to input stations, and return an output station. (See Section 3.5 for details.) The code given for the previous example may then be replaced by the following:

```
virtual_speaker(vol_adjust(marys_mic_station,local_dial_station));
```

Note that these “dataflow expressions” may be arbitrarily nested, as can conventional programming language expressions. From here on, figures will leave intermediate stations implicit for the sake of clarity.

3.4 Fade

In Figure 3-3, a sound flow is split into a pair of channels. The loudness of each channel is modulated by a proportion value that comes from a real-valued slider, with one channel being controlled directly, and the other being the complement of the slider.

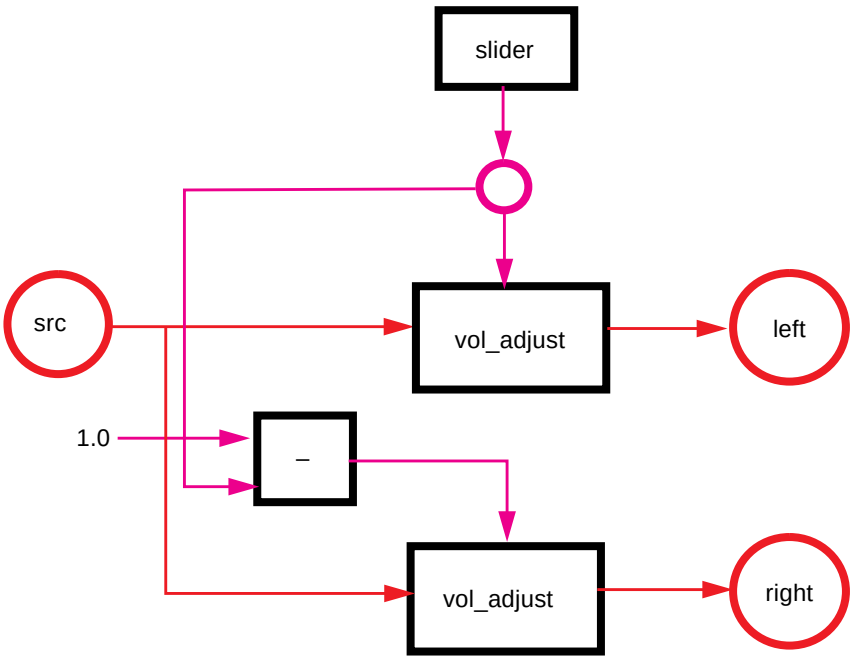

Figure 3-3 Sound fade

Note that (a) GUI sliders are supported as continuous “media”, and (b) the dataflow paradigm is being applied here even at the level of a single arithmetic operation.

Code fragment:

```
left_sound = vol_adjust(src, slider);
right_sound = vol_adjust(src, 1.0 - slider);
```

3.4.1 *Encapsulation and Reuse*

In order to be scalable, a specification or programming formalism must support encapsulation of a created structure into reusable components with well-defined interfaces. When speaking of encapsulation with respect to the dataflow paradigm, we mean the ability to take a network of dataflow nodes and encapsulate it as a single dataflow node. These compound nodes are then indistinguishable from primitive nodes, and may be replicated and further combined into higher level dataflow nodes. Designers of dataflow systems have sometimes ignored this basic principle, but we consider it to be of crucial importance.

In the previous example, we would like to abstract out the particular choice of input and output devices, defining a new component called `stereo_split` that has a sound and a numeric input, as well as two sound outputs.

Figure 3-4 and Figure 3-5 show an inner view and an outer view of the new component.

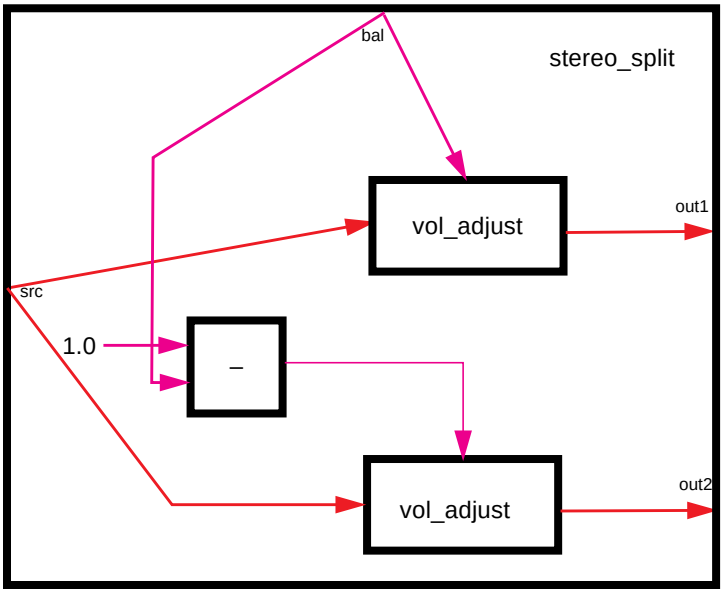

Figure 3-4 stereo_split inner view

MediaFlow's embedding in C++ allows it to inherit C++'s own encapsulation and reuse mechanisms — namely, procedure definition and invocation. For instance, a programmer could define `stereo_split` as follows:

```
void
stereo_split(Sound src, float bal, Sound &out1, Sound &out2)
{
    out1 = vol_adjust(src, bal);
    out2 = vol_adjust(src, 1.0-bal);
}
```

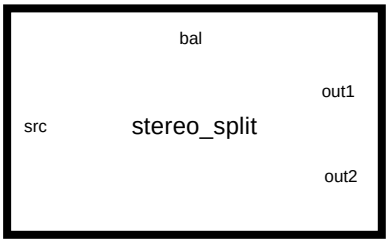

Figure 3-5 stereo_split outer view

The overloading tool would then generate a class `stereo_split_Component` and an overloading of `stereo_split` with the following interface:

```
void
stereo_split(Station<Sound> src, Station<float> bal,
             Station<Sound> &out1, Station<Sound> &out2);
```

Another use of `stereo_split` would be to create a four-way splitter based on two numerical inputs, which might correspond to left/right and front/back balance. Figure 3-6 shows a new, reusable component definition named `quad_split` that performs this task. The input sound flow is split into two, according to the left/right balance, and each of the two resulting flows is further split into two more, according to the front/back balance.

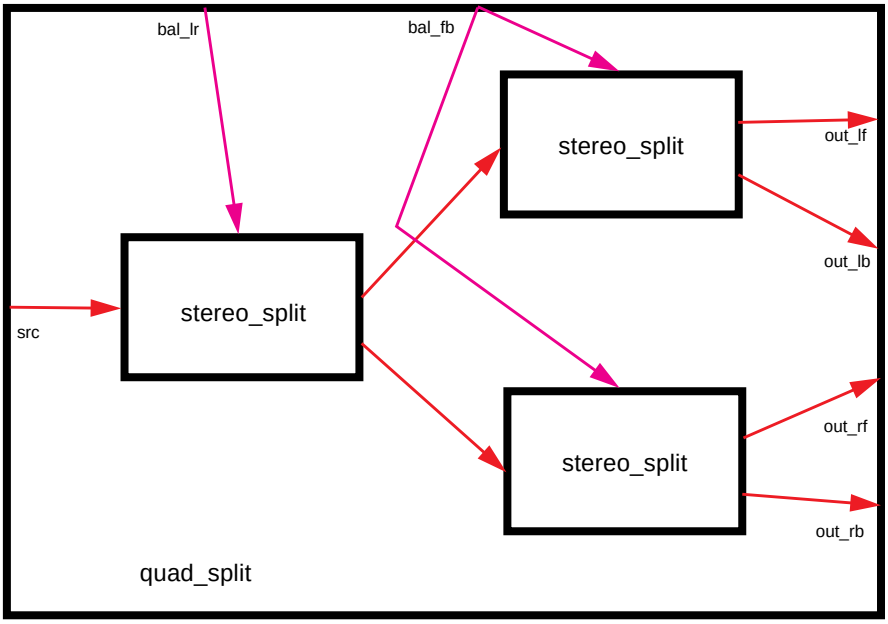

Figure 3-6 Quadraphonic split

Expressed in code:

```
void
quad_split(Sound src, float bal_lr, float bal_fb,
           Sound &out_lf, Sound &out_lb,
           Sound &out_rf, Sound &out_rb)
{
    Sound &left; Sound &right;   // temps for first split
    stereo_split(src,bal_lr,left,right);
    stereo_split(left,bal_fb,out_lf,out_lb);
    stereo_split(right,bal_fb,out_rf,out_rb);
}
```

3.5 Media Players

Our next example, Figure 3-7, introduces our formulation of a *media player.*

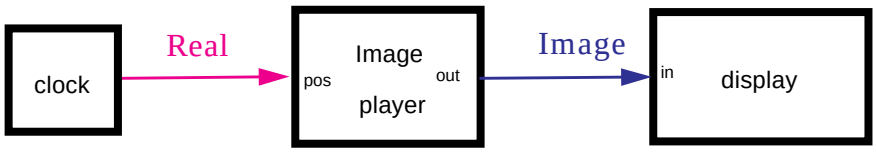

Figure 3-7 Media player

A media player for a type `T` (in this case `Image`), is a component that has one input of type `Real`, and one output station of type `T`. The idea here is that the numerical input flows continuously and tells the player what position to play. For instance, an input flow of numbers that increases over time causes the player to play forward, while a decreasing flow causes backwards play, and a constant-valued flow causes a pause. In this example, the position is being generated by a clock, which presumably would have controls like `pause` and `resume`. Note that the clock could be replaced by a dial or slider component, allowing the user to control the player's position directly.

3.6 Models as Media Sources

We treat modeled media (e.g., text-to-speech, MIDI-based music, sound effects, and interactive, animated 2D & 3D graphics) not as alternatives to the sound and image types, but rather as sources of sounds and images. For example, Figure 3-8 shows the specification of a chair rotating under control of a clock, filtered by the sine function. (Due to the sine function, the chair turns clockwise, slows down,

reverses direction to counterclockwise, slows down, reverses, etc.) The rotating chair is represented as a `Geometry` flow coming out of the `rotate` component. This `Geometry` flow and a viewing `Transform` flow enter the `render` component, which uses them to produce an image flow.

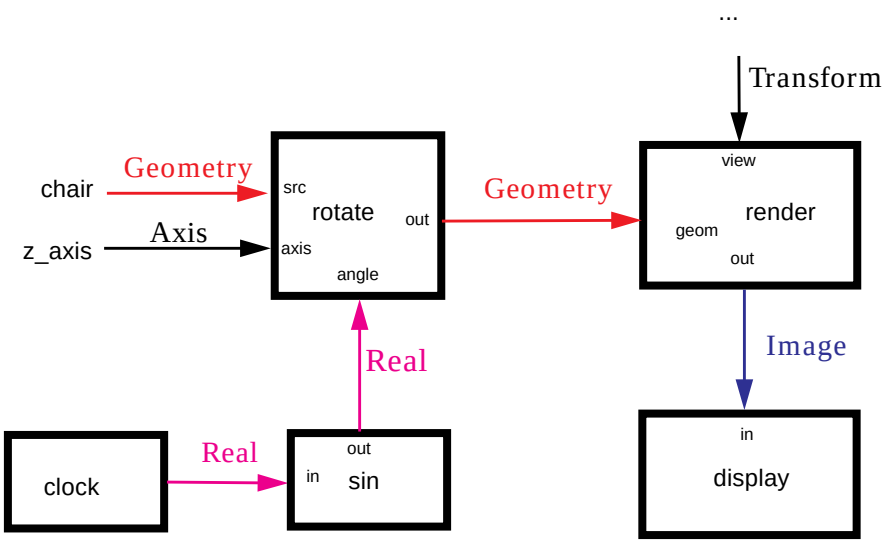

Figure 3-8 Media from models

Note that the modeled image flow generated in this example has exactly the same type as image flows in the previous examples, and is presented by the same kind of display device. We consider this uniformity to be very useful, since it allows the consumer of a flow to be insulated from the model choices made in providing the flow. (As will be discussed in Section 6.1, “What’s a value?”, this conceptual simplicity does not lead to loss of efficiency.) Another advantage of this uniformity is that we foresee some useful dataflow nodes (operations) that combine geometry with images and produce images as a result. Future interactive digital HDTV is likely to be based on such a synergy. Figure 3-9 shows an example of such a synergy. We have extended the previous example by adding in a live background image supplied by a video camera. The live background is filtered through a contrast modifier component, with the contrast ratio provided interactively by means of a slider widget.

4. *Advantages of the Continuous Approach to Media*

Media-rich applications are largely characterized by presenting and modeling real-world phenomena that undergo *continuous*, rather than discrete, changes, i.e., they are based on a continuous notion of time. (Consider sound, light, and motion.) In spite of the continuous underlying reality or model, multimedia systems tend to be based on temporally discrete foundations, as seen in the fundamental notion of a “stream”, which is some sort of a sequence of samples. In our experience, temporal discreteness imposes an artificial-feeling conceptual model on media applications.

In contrast, MediaFlow’s conceptual model takes a fundamentally *continuous* approach to media. (Unfortunately, the term “continuous media” is often used informally to mean discrete streams with high enough sample rates to give the illusion of continuous change.) Of course, this conceptual model raises non-trivial implementation challenges, but these challenges are well understood and have

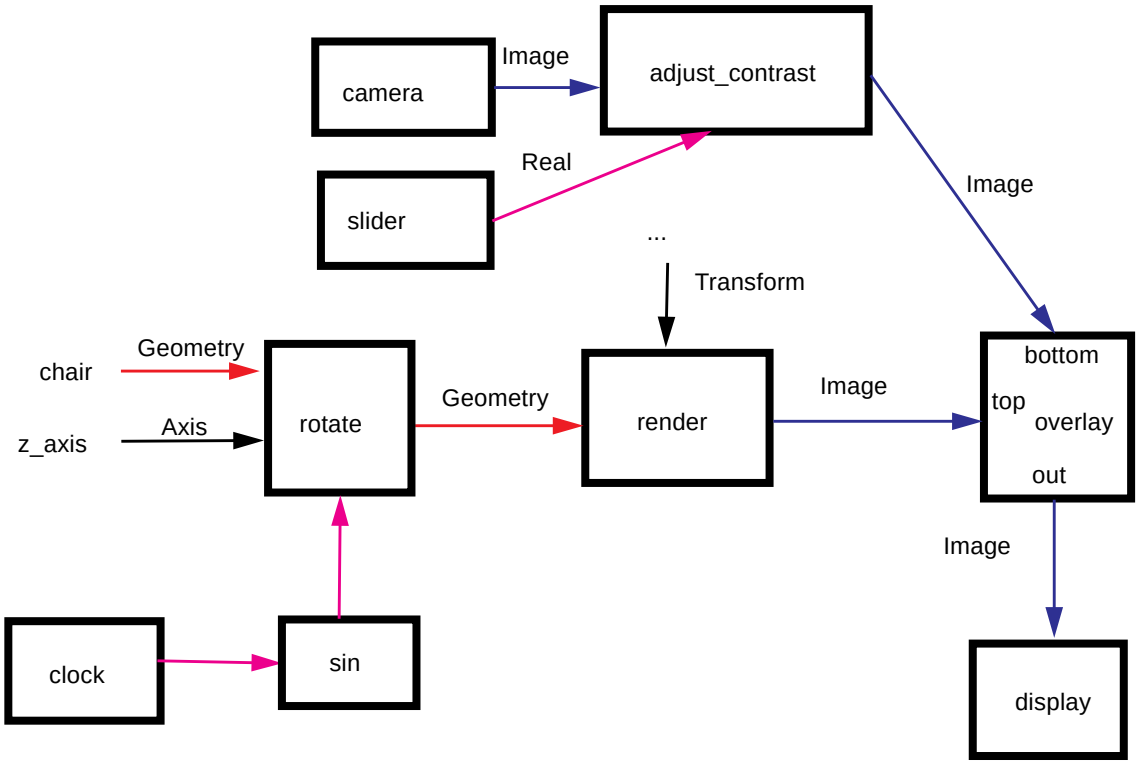

Figure 3-9 Modeled/live media synergy

effective answers. Furthermore, we strongly believe that the implementation effort required is very much worth the benefit of giving the programmer a conceptual model tailored to fit the phenomena being modeled, rather than the machines executing the model. As illustrated by the examples above, these benefits include making the following traditionally difficult tasks easy for the programmer or end-user to specify:

- Combination and synchronization of multiple pre-sampled sources, or sources and sinks, that have inconsistent sample rates
- Support of modeled media (which has no inherent sample rate)
- Seamless integration of natural and modeled media (e.g., music, sound effects, and interactive, animated graphics)

Note that while different flows may be approximated by sample streams made at different rates, programmers do not want to try to keep track of and reconcile all of these sample rates. The continuous model frees them from doing so.

5. Implementation Model

5.1 Continuous Dataflow

In order to make MediaFlow's continuous approach more concrete, consider again the example in Figure 3-9, and in particular three flows: (a) the geometry flow coming from the `rotate` component, (b) the real-valued flow coming from the slider, and (c) the image flow coming from the camera. Conceptually, i.e., from the programmer's point of view, these three flows are continuous (as they would be in nature), and the output image flow is continuous as well, as shown in the thick curves in Figure 5-1. (The four flows shown in the diagram are to be taken very figuratively.) MediaFlow will implement an *approximation* of this continuous ideal. In the slider case, the approximation might be a piecewise linear curve that interpolates between the discrete mouse locations provided by the window system. For the camera, the approximation will most likely be piecewise constant, since there is currently no real-time interpolation technique for images. (Section 3.2 discusses these issues in more detail.) In the geometry case, the flow is completely algorithmic, and so can generate an accurate value for *any* desired time. These approximations are illustrated as the thinner curves in the figure.

Although MediaFlow can construct an internal representation of the approximate continuous composite image flow shown in the figure, a display device can only present a discrete sequence of *samples* of that flow. In the figure, the vertical lines and their intersections with the approximation curves show the samples that might be selected in each of the three input flows and the resulting composite image flow. Note that the sample rate may change occasionally, and is completely independent from the rates of the low-level input devices on which our continuous input flows are based. For instance, in this case, more CPU resources may have gotten freed, allowing the sample rate to increase. Note also that other display devices may be attached to this same composite image flow. In that case, the same continuous approximate flow shown in the figure would be sampled at rates independent from that of the first display device.

5.2 Station Cloning

Our approach to supporting distribution is based on a very simple, low-tech initial idea: do a purely local design and implementation, and then just *use* it in a distributed way, by hooking up a display on one machine and a capture device on another machine and pointing them at each other. Rather than transmitting the media through the physical ether, however, transmit across the electronic ether (e.g., shared memory or network transport). (On the other hand, one could also use, e.g., FM radio transmission for practical, wireless media connections.) Implementing this idea requires the introduction of what we call "transport display" and "transport capture" components. These transport components look to the framework like regular display and capture components, but their implementation involves encoding of media values into external (process-independent) data streams, transmission to other processes on the same machine or different machine, and decoding into internal representations of media values.

As an example, consider the simple case of connecting a remote microphone to a virtual speaker, as shown in Figure 5-2.

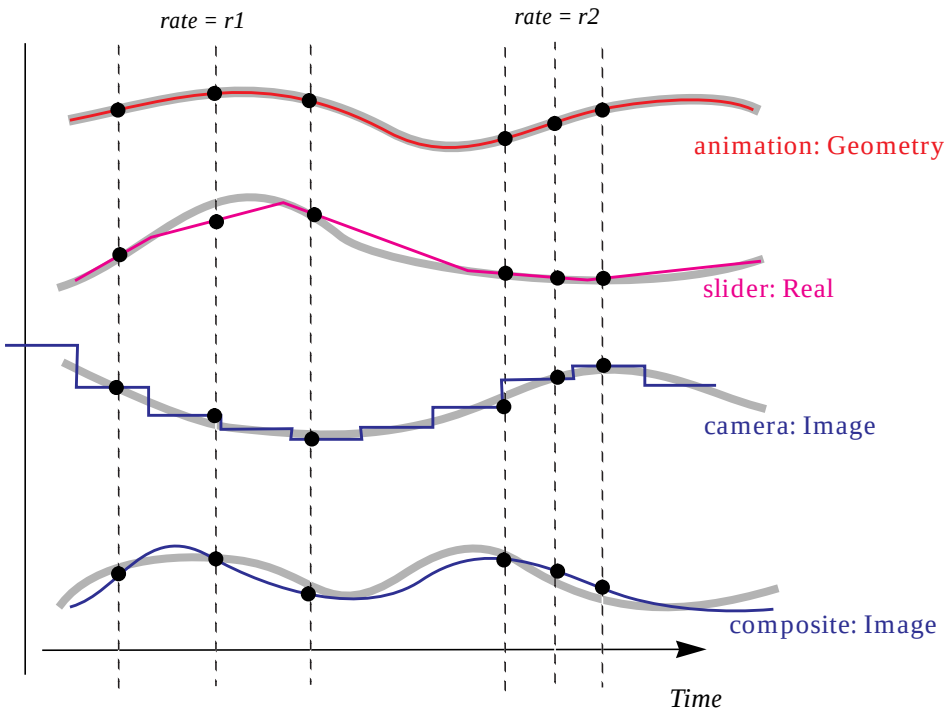

Figure 5-1 Sampled implementation input and result

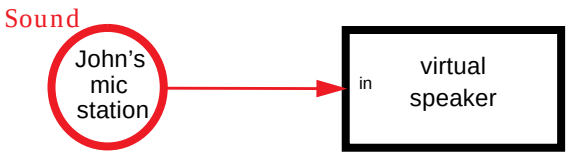

Figure 5-2 Remote microphone, conceptual

As described in Section 2.4, the process managing John's microphone station must *export* it via `export_station`, resulting in a station token. Then the process wishing to create a virtual speaker listening to it must obtain and *import* that token via `import_station`. The result is the creation of (a) a new "clone" local station, (b) a remote *transport display component*, and (c) a local *transport capture component*. The clone is then used in the creation of the new virtual speaker, as in the purely local case. By means described in the next section, the remote transport display component encodes its media value input flow to a stream of external representations that are transmitted to the local transport capture component. The result is shown in Figure 5-2.

The idea of exported stations can apply more generally than to connecting two MediaFlow processes. We also envision them being used to connect a MediaFlow process to non-MediaFlow processes, such as Internet Talk Radio or to a process using Microsoft's MCI. In this case, the application would attach a different station subclass to the network that "tunes in" the non-MediaFlow process.

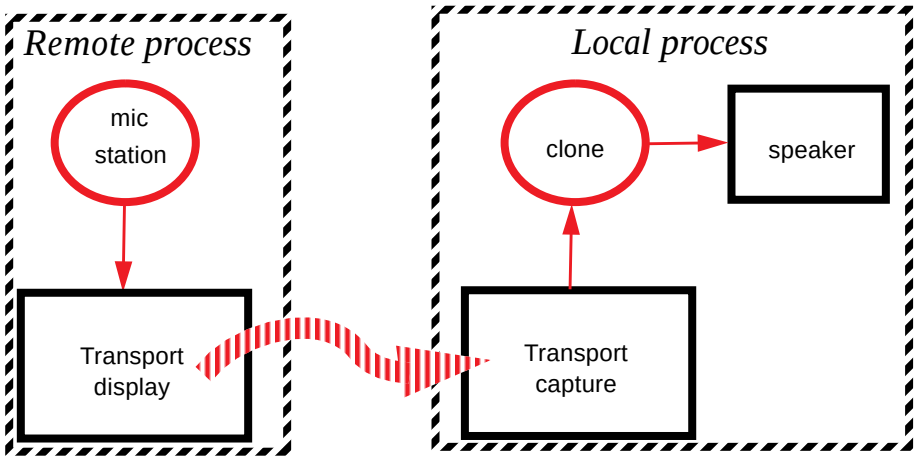

Figure 5-3 Remote microphone, implementation

6. Evaluation Strategy

Within a single process, the dataflow graphs are evaluated in a purely demand-driven manner. Every sink component has a thread with which it issues a succession of `value_at(t)` calls for increasing values of `t`. These "demands" propagate backwards to cause a cascade of `value_at` calls, ending at source components. Components maintain buffers of time-stamped values, and respond to `value_at(t)` by some form of interpolation, possibly degenerate (clamping to nearest value). This use of interpolation is what insulates the rate at which input samples appear from the rate at which they are used.

As a special case of this evaluation strategy, consider *transport* display and capture components, as in Figure 6-1. While in the programmer's model there is only one capture and one display device in this distributed dataflow graph, the implementation model provides two of each, with each process containing both a capture and a display component. In both cases, the display device drives evaluation within the process, and the capture device receives data from "outside" and buffers it, in order to respond to demands. However, the transparently-created transport capture component is receiving its data from the transparently-created transport display component. The result is a hybrid data-/demand-driven evaluation of distributed dataflow networks. Note that a simpler alternative, namely pure demand evaluation across a network would incur prohibitively expensive network round trips per sample displayed.

6.1 What's a value?

MediaFlow implements "abstract media types" as C++ abstract classes with several concrete subclasses that implement the common interface. Hence, "abstract media values" are instances of these concrete subclasses. In contrast to many uses of objects, media values are immutable and often short-lived. Some of these subclasses will correspond closely to existing external formats, e.g., the `JPEGImage` subclass of `Image`, whose internal representation contains a JPEG-compressed image. In other cases,

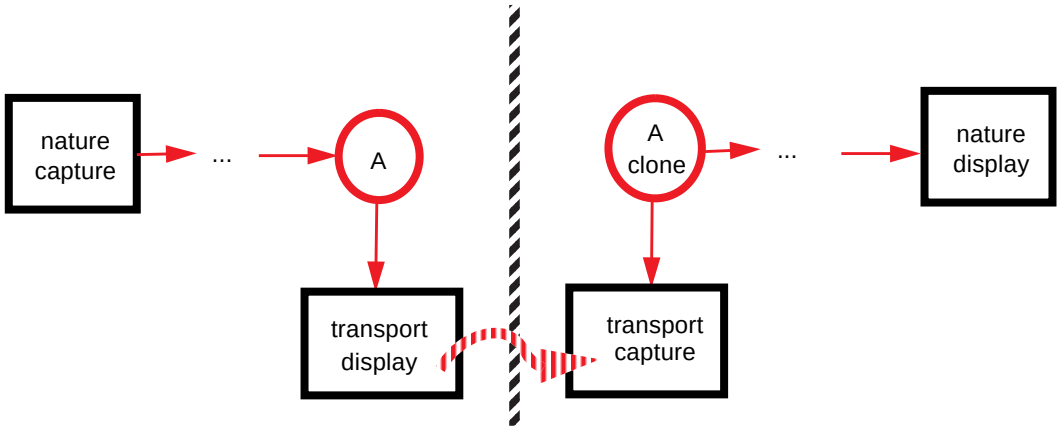

Figure 6-1 Distributed dataflow evaluation

subclasses correspond to *structured values*. As an example of this latter kind of subclass, consider a `mix` function that takes two sounds and produces a sound. A simple implementation of that function creates a sound object that contains the two argument sound values. The class and function might be defined as shown below.

```
class MixedSound : public Sound {
  public:
    MixedSound(Sound&, Sound&);
    // .. Sound methods ..
  private:
    Sound& sound1;
    Sound& sound2;
};

Sound& mix(Sound& s1, Sound& s2)
{ return *new MixedSound(s1, s2); }
```

One benefit of this sort of structural representation is that it allows deferral of actual execution of a function so that it can occur in hardware where supported. This benefit is particularly important for images based on geometry, as in Figure 3-8. The `render` function in that example is similar to `mix` above, producing an abstract image of subclass `GeometricImage`, which is simply a wrapper around the geometry and viewing transform. The actual rendering is deferred to the `display` method on the `GeometricImage`, which makes calls to an efficient, hardware-accelerated low-level graphics rendering library.

One of the fundamental principles of the MediaFlow architecture is that abstract media values are always self-contained. This principle enables filtering (e.g., any value of type `Image` can be displayed, cropped, blended, etc.), as well as arbitrary resampling by stations. In contrast, for MPEG, and other temporally compressed formats, the elements of the format stream are *not* self-contained external representations of abstract values. Fortunately, however, even these compressions schemes can be made into streams of self-contained internal value representations by having several successive values share pointers to the same key frame, i.e., just repeat whatever it takes to make the samples self-

contained. In the case of MPEG, there are three types of Images: I frames which are self contained, P frames which are deltas from an I frame, and B frames which are interpolations between frames (either P frames or I frames). There would likely be a few Image subclasses: `MpegIImage` which would point to an I frame, `MpegPImage` which would point to a P frame and the I frame it depends on, and `MpegBImage` which would point to a B frame plus the previous I or P frames plus the following I or P frames. When such a sequence has to be put into a MPEG external format, a kind of dual transformation must be done. Each I image sends its I frame. A P image causes its I to be sent only if it's different from the previously sent I, and in any case, the P (a delta to the I) is sent. Similarly for a B image which would send the frames necessary to create its previous and following frames. Aside from maintaining the simplicity of the MediaFlow conceptual model, note how this trick allows the general resampling ability of stations to do the right thing even to representations of MPEG streams.

7. Conclusions

We have described some of the aspects of the design and implementation of MediaFlow, a powerful and high-level system for supporting distributed integrated media applications. Efficient implementation of such a system is no straightforward task, but our previous experience described in [Sche94] and [Elli94] has led to several novel and practical techniques, as outlined in this paper.

8. References

[Elli94] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. To appear in SIGGRAPH '94.

[HIS93] HP, IBM, SunSoft. Multimedia System Services. Requested by Interactive Multimedia Association. June, 1993.

[Sche94] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. Functional 3D Graphics in C++ – with an Object-Oriented, Multiple Dispatching Implementation. To appear in the proceedings of the Fourth Eurographics Workshop on Object-Oriented Graphics (EOOG '94).