Toward a Common Infrastructure for Multimedia-Networking Middleware

By
Steven McCanne, Eric Brewer, Randy Katz, Lawrence Rowe,
Elan Amir, Yatin Chawathe, Alan Coopersmith, Ketan Mayer-Patel, Suchitra Raman, *Angela Schuett, David Simpson, Andrew Swan, Teck-Lee Tung, David Wu*
University of California, Berkeley
and
Brian Smith
Cornell University

Abstract

Real-time multimedia streams like audio and video are now integral data types in modern programming environments. Although a great deal of research has investigated effective and efficient programming support for manipulating such streams and although the design of digital media "middleware" is fairly well understood, no widely available or commonly accepted programming model exists within the research community. We believe this lack of common practice impedes our collective progress because it prevents disparate research groups from easily leveraging each other's work. In this paper, we propose a solution to this problem that combines the best features of a number of existing multimedia toolkits --- Berkeley's Continuous Media Toolkit, MIT's VuSystem, and the LBL/UCB MBone tools --- into a fine-grained, extensible, and high-performance toolkit. We describe the convergence of these three toolkits into a common programming infrastructure and argue that the availability and acceptance of our middleware could potentially facilitate and accelerate breakthroughs in multimedia networking.

1.0 Introduction

Ongoing improvements in workstation and PC performance have steadily enriched the variety of compute-intensive, real-time multimedia data types that can be processed and "programmed" in standard software. In the early eighties, we saw the emergence of high-resolution graphical displays, in the the late eighties, digital audio became standard on Unix workstations and PCs, and in the early nineties, low-quality digital video hit the desktop. Now, in the late nineties, we are witness to the development of high-definition video and real-time rendering of complex 3D virtual worlds.

To complement these technological developments, the research community has experimented with and developed programming environments, toolkits, and protocols for manipulating digital media within computer systems and across communication networks. This work has led to a solid understanding of many multimedia-system abstractions like cross-media synchronization [10, 27], modular and extensible manipulation of streams [20, 24, 12, 3], network bandwidth and delay adaptation [27], and scalable multimedia-networking protocols [19, 4].

While the multimedia-networking research community still faces significant challenges, we believe that research on multimedia toolkits has matured to the degree that the particular arrangement of multimedia-system components into a "middleware architecture" is perhaps less of a research problem and more of an "engineering art." We strongly believe that the most interesting research problems have to do with large-scale network protocols and systems --- how one combines together the components of the toolkit into complex and scalable systems, not how one defines the individual arrangement of API's and object semantics of the building blocks within the toolkit. Yet, no such no widely accepted approach for structuring multimedia middleware exists within the research community.

This paper describes our approach to multimedia middleware not as a research result in its own right *per se*, but rather as an opportunistic vehicle for enabling and accelerating research results within and across the community. Much as Berkeley Unix and CMU Mach served as crucial experimental platforms for extensive operating systems research in the eighties, a common digital-media research middleware could potentially enable rapid advances in multimedia-systems and multimedia-networking research. We foresee a common framework where experimentalists can pick and choose building blocks from a comprehensive "library" of media and protocol objects. This library would be assembled incrementally as individual research efforts each contribute to and extend the framework. In particular, one could imagine the following wide array of system building blocks at one's fingertips:

- video compression modules like the layered video work from Berkeley [18, 31, 30, 1];
- audio compression modules like the robust-audio codec from UCL [6];
- reliable-multicast protocol modules, e.g., based on SRM [4] can be customized for the application at hand e.g., shared whiteboards [15], webcast tools [11], or floor-control applications [14];
- primitives for archive and playback of collaborative sessions, e.g., modules from the *MBone VCR* [7];
- multicast address-allocation modules for dynamically creating "lightweight sessions" [8];
- building blocks for session advertisement and explicit session invitation protocols [5];
- building blocks for special-effects processing of video, audio signal processing algorithms like echo cancelation; and so forth.

Unfortunately, all of the above systems are currently implemented in custom and often *ad hoc* environments with little opportunity to make them interoperate. For example, porting UCL's robust audio codec [6] to the LBL *vat* application [9] would require substantial code modifications.

Our programming framework builds on our six or so years of collective experiences developing the Continuous Media Toolkit (CMT) and the

LBNL/UCB MBone tools software architecture (as it appears in *vic* [17] and *vat* [9]). In addition, we have closely examined and borrowed novel architectural concepts and code from the VuSystem developed at MIT. By combining the best characteristics of these three projects --- CMT, VuSystem, and the MBone tools --- we have leveraged a large amount of existing code into a flexible and extensible multimedia toolkit called *mash*. Although this work does not directly advance the state of the art in multimedia networking, the system represents an important enabling technology that forms the cornerstone of our and hopefully other researchers' agendas. This system provides:

- an architecture for digital multimedia application development that brings together the best features of several existing and refined systems;
- a digital media programming kernel that is small and simple and a hierarchical arrangement of multimedia objects to provide *multiple levels of abstraction* to the middleware user;
- a publicly available digital media toolkit that can be leveraged across multiple research projects in the multimedia networking research community thereby unifying complementary but disparate efforts into a cohesive system; and,
- a modular programming environment that supports a "plug and play" abstraction for rapid prototyping and dynamic loading

The remainder of this paper describes the history of *mash*, its architecture, and how it builds on the MBone tools, CMT, and VuSystem. We detail the mechanisms used to tie together an object-oriented scripting language (OTcl) with an object-oriented compiled language (C++) and how this architecture provides a highly flexible and easily extensible programming substrate. Finally, we outline how the *mash kernel* is arranged into a set of extensible objects, all linked into the Tool Command Language Tcl [21], and how these objects are used to build arbitrarily rich applications and environments.

2.0 Background

CMT, VuSystem, and the MBone tools, though each developed independently, all converged on the same basic architecture, which is split into low-overhead control functionality implemented in a scripting language like Tcl and performance-critical data handling implemented in a compiled language like C or C++. Compiled objects provide core, composable mechanisms that are arranged and configured through the scripting language to effect arbitrary application policies. This separation has proven quite useful because it cleanly divides the burden of design, maintenance, extension, and debugging from the ultimate goal --- the actual research experiments --- by providing the application programmer with an easy to use, reconfigurable, and programmable multimedia networking environment. Moreover, this model forces the programmer to factor out control and data abstractions into separate modules and thereby encourages a programming style where objects are free of built-in control policies and semantics and can thus be easily reused.

Figure 1: The "VuSystem Architecture".

Figure 1 illustrates the basic architecture. High-volume multimedia data is typically generated by a *source* object and piped through one or more *filter* objects. Eventually, the media reaches a *sink* object and is consumed. The source, for example, might be a video capture device, filters might be color space converters, compressors, packetizers, and the like, while the sink might be a network transmission protocol. Although CMT, the MBone tools, and several other toolkits employ a similar programming model, this architecture was first cleanly articulated in the VuSystem literature and we thus call this approach the *VuSystem Architecture*.

Given that each of these three systems implements the VuSystem Architecture, it would seem counterproductive to develop yet another toolkit based on the same model. However, each toolkit has certain individual weaknesses; to overcome these weakness, we set out to extract the best features of each existing system and, in the end, obtain a cleaner and more powerful framework. To justify this approach and establish context for our design, we now briefly discuss the advantages and disadvantages of these existing systems.

2.1 MBone tools

The architecture that evolved in the MBone tools originated in the first version of *vat* that one of us developed in 1990. [1] Back then, Tcl/Tk did not exist and we instead used the Interviews C++ structured graphics library [13] to build the user-interface. In time, we developed the companion video tool *vic,* and mirrored *vat's* software architecture in *vic*, but we implemented the user-interface with Tcl/Tk rather than Interviews. Although we initially used Tcl exclusively for the user-interface, the convenience and power of the split Tcl/C++ programming model gradually became apparent. Over time, *vic's* design looked more and more like the VuSystem Architecture and eventually these architectural improvements were "back ported" to *vat*. As our understanding of the audio/video protocols matured and we better understood how to factor our common pieces of the architecture, the *vic/vat* implementations began to share code and slowly migrated from independent, monolithic applications to scripted tools built from a toolkit.

A principal shortcoming of this evolutionary design is that many pieces of the system do not cleanly conform to the architecture and are instead glued together with *ad hoc* scaffolding. Hence, the Tcl scripts are often hard to manage and difficult to re-use. Another oversight was our initial perspective that the best level of design granularity was that of "composable tools" coordinated across a multipoint interprocess communication abstraction called a "Conference Bus" [17]. While a good building block, the Conference Bus is only part of the story. We now believe what was obvious to many before us --- that each composable tool should additionally be implemented from a flexible, reusable toolkit in the form of the VuSystem Architecture.

At the same time, a clear-cut advantage of the MBone tools is that they are robust, widely deployed, readily available, and conform to widely-accepted Internet standards. Consequently, our ultimate architecture should allow us to integrate this existing code base.

2.2 Continuous Media Toolkit

The Berkeley Continuous Media Toolkit (CMT) [24, 29] project started on the same time frame as the Mbone tools. CMT began as an architecture for playing pre-recorded multimedia data over local area networks [26]. In addition to the development of the core architecture, several important technologies have been "spun-off" from the CMT project. These include Tcl-DP [28, 22], an extension to Tcl/Tk that facilitates distributed programming, and the Berkeley MPEG player [25].

CMT adopted a VuSystem Architecture based on Tcl/Tk early in its development, allowing objects to be mixed and matched into arbitrary applications. CMT objects are implemented in C, and as described above, composed with Tcl. CMT has a well-developed architecture, thoroughly documented API's, and a rich set of objects and applications for multimedia authoring, video indexing, video-on-demand, and so on. The CMT project is active and its infrastructure continues to grow.

In addition to the basic set of objects, CMT provides a framework for synchronizing and controlling several multimedia streams. It also contains a mechanism for scripting actions when a specific event occurs during a multimedia presentation. A CMT application can evaluate an arbitrary fragment of Tcl code when an object event occurs using the `cmbind` command much as window event bindings are established using Tk's `bind` command. Example events include frame drops and receiving a network packet. Events can be aggregated so that the application is not overwhelmed with fine-grain events. Such callbacks, when combined with Tcl-DP, facilitate the development of novel feedback mechanisms to control skew, network flow, and resource allocation.

But, like the MBone tools, CMT has some shortcomings. The internal architecture is flat and does not exploit object inheritance or a consistent object-oriented programming model. Moreover, the RTP networking code is underdeveloped compared to the MBone tools. Because CMT did not use a traditional object-oriented language, complexity is overloaded into monolithic objects rather than decomposed into fine-grained objects. For example, the decoder objects invoke dithering routines and manipulate X windows rather than imply decoding video to uncompressed form and passing the result to a generic downstream object.

2.3 VuSystem

The VuSystem is a similar multimedia toolkit built on top of an object-oriented Tcl extension called OTcl [32]. C++ classes implement multimedia objects that produce, consume, or filter real-time media streams and a Tcl shadow object mirrors each C++ object. Methods invoked on the Tcl shadow object are dispatched to the C++ object. Like CMT, objects can be created, composed, and configured from Tcl and the interconnections between objects rearranged while running. In addition to its elegant, object-oriented architecture, the VuSystem is built around a consistent and uniform fine-grained object model with inheritance.

Unfortunately, the VuSystem also has limitations. Not only is the project inactive, but its large, existing code base is built on an early, underdeveloped version of OTcl. Moreover, its custom user-interface is limited and does not reap the benefits of the ongoing, large-scale development effort that underlies the Tk toolkit. As well, the system has no support for media compression, thus preventing its application to heterogeneous environments where network bandwidth is scarce. Finally, the VuSystem project goals were more oriented toward system design issues compared to network communication models, and consequently, its network support is primitive; it does not, for example, support RTP or multicast.

2.4 A Combined Architecture

Based on the introspection of our work and the analysis of the VuSystem work, we arrived at a new architecture, embodied in the *mash* toolkit, that exploits the best attributes of each of the previous systems. Our design process began and is evolving as follows:

1. We built a preliminary toolkit prototype by arranging the bulk of the C++ objects from the existing MBone tools into a collection of fine-grained objects each with an OTcl API. This prototype allows us to immediately exploit MBone tools' code base and robust implementation.
2. We retrofitted the VuSystem's elegant object model by adopting Wetherall's "OTcl" package [32] and built a number of high level abstractions called "macro-objects"out of the fine-grained objects from the prototype toolkit.
3. Lastly, we plan to merge the *mash* and CMT toolkits into a single infrastructure by developing an OTcl "scaffolding layer." To this end, we developed an incremental transition strategy to merge CMT's most powerful abstractions (e.g., the buffering model and event prioritization scheme) into the *mash* kernel by linking the *mash* kernel and CMT library into a single interpreter that can execute either CMT or *mash* scripts. Once we develop high-level OTcl API's, application developers will write to this API and the distinction between *mash* and CMT will evaporate.

The key abstraction in this "combined architecture" is how we implement and split objects across C++ and OTcl. Consequently, we now discuss in some detail the motivation and design of our "split object" software model.

3.0 The Software Model: Split Objects

The implementation of a multimedia-networking toolkit is naturally decomposed using object-oriented representations. Protocol modules and media codecs, for instance, are all conveniently represented using class hierarchies and inheritance. We might derive an H.261 video encoder from a "video encoder" parent class, which is in turn derived from a generic codec module. Likewise, network protocols and media codecs are naturally represented using message passing among object instances since both these functions typically involve a number of processing stages. Because multimedia applications often generate highly dynamic flows where users come and go and the parameters of media streams evolve in real-time, the programming model must furthermore support easy and flexible reconfiguration of object pipelines. Finally, object-oriented design is "good programming methodology" --- it provides a flexible framework and, assuming appropriately designed objects and object API's, supports efficient code re-use.

For these reasons, we adopted an object-oriented software architecture for our multimedia middleware. Our architecture builds on the MIT Object Tcl system, or OTcl. OTcl provides an object-oriented extension to Tcl without any modifications to the Tcl core, thereby making the code independent of the ongoing Tcl development effort. It supports dynamic binding of the class hierarchy, multiple inheritance, and topological **method** combination along the lines of the *Common Lisp Object System*.

Our approach elaborates the VuSystem Architecture described above by bridging the gap, somewhat seamlessly, between C++ and OTcl. We view an object as an abstract entity whose methods can be implemented on either side of the OTcl/C++ boundary. C++ code can invoke methods defined in OTcl, while OTcl code can invoke methods defined in C++. The common idiom whereby Tcl callbacks are installed in C data structures is replaced by a simple, clean object **method** API. A callback is simply a self-referential **method** invocation.

In our OTcl/C++ framework, fine-grained objects are implemented in C++ and complex abstractions are built by coalescing these fine-grained objects into "macro-objects" using scripts to glue together components. To support a clean and manageable programming model, the details of a macro-object can be encapsulated in an OTcl object with a uniform **method** API. In turn, macro-objects can be arranged into larger and richer macro-objects thereby giving the toolkit programmer *multiple levels of abstraction* and allowing her to choose the appropriate level of detail for a given implementation.

A common objection to this approach is its complexity. Why not implement an object-oriented application entirely in C++? One can still use clean practices and separate policy and mechanism within the confines of a single programming language. We feel, however, that this approach requires unrealistic discipline from most programmers. Moreover, because policy and mechanism are so semantically different, we believe they are most appropriately implemented in separate languages: "mechanism" is particularly suited for a low-level language like C++ where run-time performance is often critical, while "policy" often requires *ad hoc* and flexible rule systems that are more easily implemented in a high-level scripting language such as

OTcl. Finally, the OTcl/C++ split provides a number of other tangible advantages:

- The model offers *late binding*. Objects can be created on the fly and object type dependencies resolved at runtime.
- Memory management is, for the most part, non-existent at the scripting level. Data structures do not become corrupt because of pointer bugs, while programming errors can be gracefully caught, handled, and possibly debugged with run-time exception mechanisms.
- The compilation and link stages are eliminated from the development cycle. Further, debugging is facilitated by the ability to program simple diagnostic constructs on the fly. Using the OTcl object hierarchy, we can conveniently insert debugging hooks into the base class implementations and thereby trap and trace actions uniformly across all or selected objects.
- Finally, the OTcl/C++ architecture becomes a design metaphor for the policy/mechanism split that, once mastered by the *mash* programmer, becomes a boon to code reuse and extensibility.

3.1 OTcl: A Brief Tour

In this section, we briefly outline the OTcl programming model to provide a flavor for how our OTcl extensions are ultimately applied to our multimedia toolkit.

OTcl objects appear simply as new procedures in the Tcl interpreter --- a **method** is invoked on an object by calling the object as a procedure and passing the **method** name and parameters together as arguments to the Tcl procedure. For example, an object of class `type` is created and associated with an object named "`object`" as follows:

```
type create object
```

Each object has its own set of instance variables that can be manipulated using the following syntax:

```
object set var $value
set v [object set var]
```

A **method** `foo` is dispatched on an object as follows:

```
object foo $arg1 $arg2 ...
```

(The `set` command above is simply another **method**.) Finally, the object may be deleted with:

```
object destroy
```

Suppose we want to define an OTcl class called "VideoAgent" as the principal API to the underlying video sub-system. A video stream can be created and manipulated by invoking methods on the VideoAgent class and most of these methods can be written in OTcl. For example, we might create the class as follows and define a **method** to build nodes using the OTcl instproc primitive[2] :

```
Class VideoAgent
VideoAgent instproc set-fps { f } {
      $self instvar fps_ encoder_
      set fps_ $f
      if [info exists encoder_] {
            $encoder_ set-fps $f
      }
}
```

The `instproc` class **method** --- the OTcl analog of the Tcl `proc` command --- defines a new instance procedure or object **method**, while the `instvar` class **method** brings an object instance variable into local scope. This scope is defined by the implicit `$self` member variable that is set to the object instance analagous to the C++ `this` pointer. Thus, in the `set-fps` **method** above, we declare `encoder_` as an instance variable and if it exists, the frame rate parameter is passed on to this object. This ability to attach instance variables to objects overcomes a key shortcoming of Tcl --- its lack of structured name spaces. Without such extensions, Tcl provides only global and local variable scope and thus the development and maintenance of large-scale software projects all within the scripting language is cumbersome.

All OTcl methods are virtual; this leads to a key advantage of our approach --- the ability to easily create new variants of core mechanisms. Suppose we wanted to add support for special-effects processing to the VideoAgent module. One approach is to extend the existing modules that make up the VideoAgent class as well as the VideoAgent with knowledge of effects processors and filtering modules, and to add configuration hooks to enable/disable different special-effects options. Instead, a better approach is to exploit object class inheritance to override basic function with new function only where absolutely necessary. For example, we might create a variant of a VideoAgent that constructs the encoding path in a slightly different fashion:

```
Class SpecialAgent -superclass VideoAgent
SpecialAgent instproc create-encoder {} {
      ...
}
```

Instead of overloading the generic VideoAgent with new functionality, we can compose special-effects algorithms from existing and perhaps new objects. We need not unnecessarily complicate the generic case with effects primitives that might otherwise introduce incompatibilities into the already-working subsystem. The ease of such extensions derive directly from the fact that a scripting language is such a good match to policy and structure specification, freeing the programmer from concerns about low-level issues such as memory management.

3.2 OTcl and C++

Although OTcl exports an API for interfacing to C programs, it has no direct support for C++ and no explicit support for constructing objects split across OTcl and some other language. To fill this void, we extended OTcl across C++ using an abstraction called *shadow objects*. To support shadow objects, we added two new Tcl procedures: `new` and `delete`. Whenever the programmer creates a new OTcl object with `new`, the shadow-object layer creates a corresponding C++ shadow object and binds it to the OTcl object. As a programming convenience, `new` also assigns each object a unique name so that the programmer need not explicitly name every OTcl object.

The mechanism we use to intercept and create shadow objects is relatively straightforward and exploits the existing OTcl class machinery through a new class called "TclObject." Every split object is derived from this base class; thus, the TclObject constructor is called for every new object and we can conveniently create the C++ shadow object from this vantage point. Likewise, when the OTcl object is destroyed, the TclObject destructor is invoked, and we in turn delete the C++ shadow object.

As a side effect of creating the C++ shadow, we arrange for undefined OTcl **method** invocations to be dispatched to C++. Fortunately, OTcl provides a convenient hook to do this, through its `unknown` **method**. If the user invokes a **method** on an object that is undefined, the OTcl dispatcher instead calls the **method** named `unknown`, which by default, prints an error message and terminates the program. Our TclObject class overrides the `unknown` **method**; instead of printing an error, the TclObject version of `unknown` redirects the **method** to C++. Figure 2 illustrates how **method** invocations are passed up the OTcl inheritance chain (e.g., C1 to TclObject). If no **method** with the given name can be found in the classes, the `unknown` **method** is called. When this happens, the C++ `command` **method** of the underlying C++ object gains control. Like all Tcl command extensions, this function takes a C-style string argument vector (a count of strings and their values) and in turn interprets the **method**, performs the appropriate action, and optionally returns a result.

Figure 2: OTcl/C++ method invocation.

As with OTcl, "**method** combination" is implemented in C++, but unlike OTcl, it must be done using explicitly-scoped names. If the leaf class of an object instance does not recognize the **method**, it invokes the `command` **method** of its parent C++ class. This new invocation then has the opportunity to interpret the command, but again, if it too does not recognize the command, control is transferred to its parent. Eventually, some class along the path to the TclObject base class recognizes the **method**, carries it out, returns a value, and the call-stack unwinds eventually causing a value to be returned to and control to resume in the Tcl interpreter. If, on the other hand, the command is not recognized by any of the derived classes, then the `command` **method** in the TclObject base class eventually gains control and raises an error condition. By default, the base class prints an error message and exits.

In addition to defining methods on either side of the OTcl/C++ boundary, the programmer is free to define and/or access instance variables from either C++ or OTcl. A binding between an OTcl variable and its C++ counterpart is established using the `bind` **method** of the TclObject class. `Bind` takes a pointer to the C++ variable and the OTcl variable name and causes writes to the OTcl variable to be reflected to the C++ shadow copy and likewise reads from the OTcl variable to be computed from the current C++ variable's value. We implemented this abstraction with little effort since Tcl already includes an extensive and flexible variable-tracing facility.

3.3 Class Extensions

While the TclObject class allows one to define object instances that are split across C++ and OTcl, we still require a mechanism for declaring the OTcl Class that creates the given object instances. That is, there must be some mechanism that allows the programmer to create support for a new object type in C++ and export that new class from C++ into OTcl, effectively extending the set of class types that `new` understands. To do so, we introduced a new C++ base class called **TclClass**. The **TclClass** constructor takes its class name as an argument and has the side effect of registering that new name as a newly supported class type. Furthermore, the class name is hierarchical, so that we can explicitly define class hierarchies using a pathname syntax, e.g., the class A/B/C has parent A/B, which in turn has parent A, which finally has the implicit parent TclObject. The **TclClass** class contains a virtual **method** called `create`, which is invoked whenever `new` is called to make a instance of the corresponding class. The programmer implements `create` by writing code to create a new object instance (i.e., a derived class of TclObject) and return a pointer to the new object.

Using this class extension mechanism, we can statically extend the OTcl interpreter with new class types simply by linking in new object modules. We merely create a template subclass of **TclClass** that specifies the new class name and creates new objects via `create`. We define a static instance of this object in a module and link that new module into the OTcl interpreter. The side effect of creating the statically declared object at load time will cause the new class to be created in the OTcl interpreter. In short, no files in the core *mash* source repository need be modified to add support for a new OTcl object type.

3.4 An Example

Figure 3 illustrates how the **TclClass** and TclObject classes are specialized to add a new object, say an H.261 video encoder, to the repertoire of multimedia objects. We derive a subclass "H261Encoder" from an intermediary subclass called Encoder, which is in turn, derived from a TclObject. Presumably, there are many types of encoding objects and all such objects share a number of common functions implemented in the Encoder subclass. As part of specializing the H261Encoder class, we define its `command` **method** to interpret the OTcl "set-quality," "resize," etc. object methods. Likewise, we define a template class, called H261EncoderClass, for creating new H261Encoder objects and we define its `create` **method** to simply return a new

H261Encoder. The entire C++ class definition is given in Figure 4. The new class is automatically created as a side-effect of defining the static instance `class\_h261\_enc`.

Figure 3: Defining a Tcl object.

```
static class H261EncoderClass : public TclClass {
public:
    H261EncoderClass() : TclClass("Encoder/H261") {}
    TclObject* H261EncoderClass::create(int argc, const char*const* argv){
        return (new H261Encoder());
    }
} class\_h261\_enc;
```

Figure 4: C++ template for a new object.

Because the programmer might want to interrogate and/or manipulate C++ variables from OTcl, any number of "interesting" C++ variables can be exported into OTcl from the C++ constructor. For example, the H261Encoder constructor exports an instance variables as follows:

```
H261Encoder::H261Encoder()
{
     bind("quantizer_", &quantizer_);
     quantizer_ = 10;
     ...
}
```

Recall that `bind` establishes a correspondence between a C++ instance variable and an OTcl instance variable. Thus, if we create a new object from OTcl as follows:

```
set encoder [new Encoder/H261]
```

we can set its associated quantizer variable (stored in the C++ object) using an OTcl **method** invoked on the new object:

```
$encoder set quantizer_ 2
```

As a programming convenience, the initial value for an instance variable that has an associated OTcl binding is copied from the class variable with the same name. For example, we might set the default audio encoding buffer size to 160 samples across all object instances simply by setting the corresponding OTcl class variable:

```
Encoder/Audio set bufferSize_ 160
```

Then whenever we create a new instance of an audio encoder object, or a subclass thereof, its `bufferSize_` instance variable is initialized to 160.

Finally, we might want to extend the H261Encoder class with new methods that need not be implemented in C++ but instead can be easily prototyped in OTcl. For example, we might
implement the **method** that maps a generic video quality parameter (say that ranges from 1 to 100) into an H.261 quantizer (that varies from 1 to 31) as follows:

```
Encoder/H261 instproc set-quality q {
     $self instvar quantizer_
     set quantizer_
          [expr ($q-1)*31/100+1]
}
```

Even though the original encoder object is implemented in C++, the object can be extended with methods defined completely in OTcl.

4.0 MASH Kernel

tal of pr se ne se go to

Now that we have described the software model for extending Tcl with object-oriented extensions using OTcl and for reflecting the OTcl programming model into C++, we can describe the *multimedia kernel* that implements the core middleware primitives built on top of the OTcl/C++ object system. This discussion reflects our initial attempt at an evolving design. In particular, as we make additional progress on the CMT integration, the objects and APIs are likely to be refined and improved.

The core set of *mash* objects consists of compiled objects that produce or consume data buffers, which may be packet data, audio data, video data and so forth. In the current version of *mash*, objects are not typed and there are no run-time checks to verify that objects are linked together in a legal or consistent manner. For example, if we attach an object that produces packet buffers directly to an object that accepts video buffers, the system will likely abort because of an illegal memory reference. One solution to this problem is to modify the toolkit API by adding type checks to each buffer transaction. But a better approach is to implement type checking all at the OTcl level. By maintaining a table that maps object names to types, we can simply add an "attachment time" check to verify the integrity of the interconnect, i.e., that the types of each named object argument is type safe.

The *mash* programming model is event driven. Events are dispatched to objects either from the Tk event dispatcher or from via **method** invocation from OTcl. The objects are arranged into a data flow graph and data generated by events (e.g., an I/O read event) is typically processed and passed on to the downstream object.

Many events in the system are driven off timers and whether the corresponding handlers are implemented in C++ or OTcl is typically determined by their expected frequency. Coarse grained timers, for instance, are almost always carried out by OTcl classes. For example, network transmission statistics might be displayed in the user interface every second or so. To do this, an OTcl class would schedule an appropriate timer, interrogate the underlying objects, and update the user interface. Real-time video frames, on the other hand, require finer grained timing. Here, a 25 or 30Hz timer runs in the C++ side of the implementation to trigger video frame captures (or the capture process might be driven off I/O completion events also scheduled in C++).

The *mash* project is dedicated to supporting a wide range of building blocks within this framework. Our toolkit contains a large and growing number of fine-grained building blocks including: RTP packet recorders, RTP packet players, audio/video device interface modules, audio/video software-based codecs, simple video effects processors, support for the LBL Conference Bus [17], a set of objects that orchestrate the Scalable Reliable Multicast protocol framework [4], many and varied user interface elements, network and encryption objects, RTP session objects a packet buffer model, a class interface to the "ghostscript" postscript interpreter, video and image rendering/dithering objects, Netscape plugin modules, and so forth.

Although these objects all taken together provide a rich infrastructure, they each export low-level API and building full-blown applications out of such fine-grained objects would require a large implementation effort. To raise the level of abstraction, we arrange sub-collections of these fine-grained objects into the "macro-objects" described earlier. Once a macro-object is implemented and its interface defined, it can be re-used in an arbitrary configurations. If the underlying implementation changes, then only those objects affected by the changes, which are typically internal to the class, need to be updated. Finally, the macro-objects are arranged into additional hierarchical structure offering the *mash* programmer multiple levels of programming abstraction.

4.1 Multiple levels of abstraction

Figure 5 depicts our model for multiple levels of abstraction with an example of a video application. At the lowest level, one must create bare network objects to send and receive packets over the network. In common multimedia protocols, the communication is divided into a network channel and a control channel and it is convenient to encapsulate this decomposition behind a suitable abstraction (e.g., the session object shown in the diagram). Sessions can come in different flavors (e.g., RTP or SRM) and RTP sessions are either audio or video (in this case video). This functionality can be nested in a single macro-object called a VideoSession. Now, within a video session media sources come and go and their streams must be consequently managed by creating objects pipelines to decode and possibly display the media in its proper format. Once we implement code to orchestrate streams in this fashion, we can further encapsulate this function in the VideoAgent class.

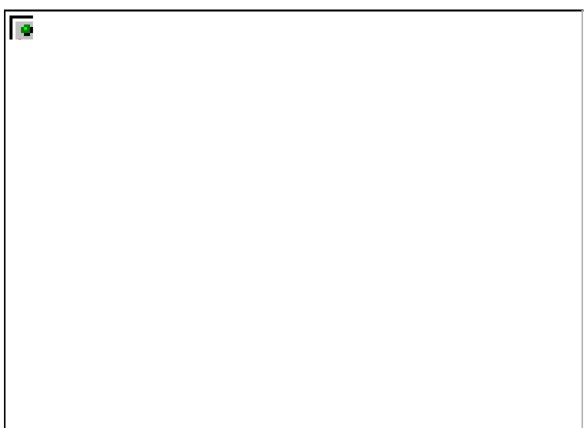

Figure 5: Abstraction hierarchy.

These video streams are not useful unless we do something with their decoded output. In a "video gateway" [2], we might re-encode the output in another format at a different rate for bandwidth adaptation, whereas in a conferencing application [17], we must display the decoded streams to the user. Another example is a bank of objects connected to timers selecting channels from a broadband cable. The gateway and cable objects may not have a GUI because they are operated and controlled by a broadcast management system whereas the conference application include a GUI for the local user. Hence, we should allow the VideoAgent to be embedded in multiple environments. Figure 5 shows a *vic*-like configuration where we create a "VicUI" object and attach it to the VideoAgent via a well-defined API. Finally, all of this function can be embedded in a VideoApp class to allow the programmer to easily create an entire video application from a script in a few lines of code.

5.0 Applications

tal of pr se ne se go to

Even though the *mash* toolkit is of very recent vintage, we have already developed a number of applications that exploit the system. As depicted in Figure 6, one of our first tasks was to factor each of the MBone tools into two separate pieces: a policy component that implements a variant of the original user interface and a mechanism component that implements bare, media processing.

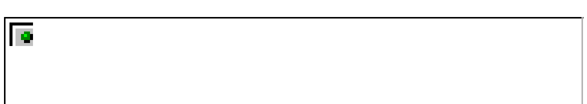

Figure 6: MBone Tools split.

To do this, we imposed our split-objects programming model onto the original code and reorganized the MBone tools' Tcl scripts as a hierarchical collection of OTcl classes (with small changes to the C++ API to support the new programming model). Additionally, we moved all object create and deletion into OTcl to simplify and unify the management of object instances. The end result is quite powerful, as it is now quite easy to "throw together" new applications by composing toolkit building blocks. For example, a *vic*-like application can be more or less built with the following OTcl script:

```
set agent [new VideoAgent ...]
set ui [new VideoUI ...]
$agent attach $ui
```

Even more convincing, certain new functionality that would have previously been quite difficult to implement is now more or less trivial. For example, once *vic* and *vat* had individually become stable and mature, we planned to develop a new tool that combined their function into a single user-interface. However, this plan never materialized because the old code base complicated the task.With *mash*, on the other hand, a combined A/V tool is trivially implemented as follows:

```
set audio [new AudioAgent ...]
set video [new VideoAgent ...]
set ui [new AudioVideoUI ...]
$audio attach $ui
$video attach $ui
```

The only non-trivial challenge here was to implement the AudioVideoUI class. By exploiting the existing OTcl class hierarchy, i.e., by re-using pieces of the audio and video UI objects, the AudioVideoUI was implemented quite easily in an afternoon.

We close our discussion of *mash* applications with a brief outline of some of the tools we are developing as part of our research in scalable and heterogeneous multimedia networking. First off, we are building a next-generation whiteboard called *mediaboard* [23] to further explore reliable multicast protocol issues and to develop an "active objects" architecture for large-scale multipoint networked animations. We continue to refine audio and video compression algorithms and protocols and are implementing layered codecs in combination with the Receiver-driven Layered Multicast (RLM) protocol [19] to deliver real-time video streams to heterogenous receivers. Because RLM operates on coarse time scales, we implemented it entirely in OTcl. And further, because our network simulator *ns* [16] and *mash* share the same OTcl/C++ framework, the RLM code can move seamlessly from its simulation environment into production use. We have also developed video gateways or "proxies" [2], which carry out rate-adaptation as an alternative approach for dealing with receiver bandwidth heterogeneity. Another area that we are tackling is the design of scalable multicast control protocols.We drive this design process with real applications and application-level protocols for collaborative floor-control and adaptive, intra-session distributed bandwidth allocation. We are designing information dissemination models and building webcast-like applications [11]. Finally, our archive system ties all the pieces of the architecture together in a comprehensive and large-scale storaging and indexing system that will facilitate multimedia content authoring, cross-referencing, and playback both at high-quality in the local environment as well as at adaptive-quality across the wide-area Internet.

6.0 Summary

tal of pr se ne se go to

Multimedia applications are often large and complex pieces of software. We described a systematic approach --- through object hierarchies and a split OTcl/C++ programming model --- for taming this complexity. By simplifying each primitive component, we built a system that is easy to reason about, and by deferring the composition of simple objects to a scripted language like OTcl, we moved the burden of debugging and verifying the complex interaction of consituent components out of the compiled language, which now only implements mechanism, into an easy-to-use, high-level, scripting language that is more suited for this task. In summary, our object-oriented digital media middleware, the *mash* toolkit:

- is compatible with MBone tools because we largely exploited this code base in our framework,
- is currently operational for several apps (*vic*, *vat*, A/V combo),
- promotes easy and flexible code reuse, and \item provides multiple levels of abstraction for developer.

Ultimately, we hope that the availability and acceptance of a digital-media middleware like ours will facilitate and accelerate research that leads to breakthroughs in multimedia networking.

7.0 Acknowledgments

tal of pr se ne se go to

The *mash* architecture benefited substantially from our experience developing the LBNL MBone tools in collaboration with Van Jacobson. Our work is supported by DARPA contract N66001-96-C-8508 and grants from Fuji Xerox, IBM, Intel, Microsoft, and Philips.

Footnotes

tal of pr se ne se go to

[1] McCanne in collaboration with Van Jacobson at the Lawrence Berkeley National Laboratory.

[2] In our object-system, we append underscores to instance variables to distinguish them from local or global variables.

[3] ftp://ftp.ee.lbl.gov/conferencing/vat

4 http://www-nrg.ee.lbl.gov/ns/

5 http://www.bmrc.berkeley.edu/cmt/

References

[1] Amir, E., McCanne, S., and Vetterli, M. A layerd DCT coder for Internet video. In *Proceedings of the IEEE International Conference on Image Processing* (Lausanne, Switzerland, Sept, 1996), pp. 13-16.

[2] Amir, E., McCanne, S., and Zhang, H. An application-level video gateway. In *Proceedings of ACM Multimedia '95* (San Francisco, CA, Nov. 1995), ACM, pp. 255-265.

[3] Craighill, E., Fong, M., Skinner, K., Lang, R., and Gruenfeldt, K. SCOOT: An object-oriented toolkit for multimedia collaboration. In *Proceedings of ACM Multimedia '94* (Oct. 1994), ACM, pp. 41-49.

[4] Floyd, S., Jacobson, V., McCanne, S., Liu, C.-G., and Zhang, L. A reliable multicast framework for lightweight sessions and application level framing. In *Proceedings of SIGCOMM '95* (Boston, MA, Sept. 1995). ACM, pp. 342-356.

[5] Handley, M., and Jacobson, V. SDP: Session description protocol, Nov. 1995. Internet Draft (work in progress).

[6] Hardman, V., Sasse, M. A., Handley, M., and Watson, A. Reliable audio for use over the Internet. In *Proceedings of INET '95* (Honolulu, Hawaii, June 1995).

[7] Holfelder, W. MBone VCR--video conference recording on the MBone. In *Proceedings of ACM Multimedia '95* (San Francisco, CA, Nov. 1995), ACM, pp. 237-238, 545-546.

[8] Jacobson, V. SIGCOMM '94 Tutorial: Multimedia conferencing on the Internet, Aug, 1994.

[9] Jacobson, V., and McCanne, S. *Visual Audio Tool.* Lawrence Berkeley Laboratory. Software on-line[3].

[10] Kouvelas, I., Hardman, V., and Watson, A. Lip synchronisation for use over the Internet: Analysis and implementation. In *Proceedings of GLOBECOM '96* (London, UK, Nov. 1996).

[11] Liao, T. WebCanal: a multicast Web application. In *Proceedings of the 6th International WWW Conference* (Santa Clara, CA, Apr. 1997).

[12] Lindblad, C. J., and Tennenhouse, D. L. The VuSystem: A programming system for compute-intensive multimedia. *IEEE Journal on Selected Areas in Communications 14*, 7 (Sept. 1996), 1298-1313.

[13] Linton, M., Calder, P. R., and Vlissides, J. M. InterViews: A C++ graphical interface toolkit. Tech. Rep. CSL-TR-88-358, Stanford University, Palo Alto, CA, July 1988.

[14] Malpani, R. Floor control for large-scale Mbone seminars. Computer science department, University of California, Berkeley, May 1997.

[15] McCanne, S. A distributed whiteboard for network conferencing, May 1992. U.C. Berkeley CS268 Computer Networks term project and paper.

[16] McCanne, S., and Floyd, S. *The LBNL Network Simulator*. Lawrence Berkeley Laboratory. Software on-line[4].

[17] McCanne, S., and Jacobson, V. *vic:* a flexible framework for packet video. In *Proceedings of ACM Multimedia '95* (San Francisco, CA, Nov. 1995), ACM, pp. 511-522.

[18] McCanne, S., Vetterli, M., and Jacobson, V. Low-complexity video coding for receiver-driven layered mulitcast. *Accepted for publication in IEEE Journal on Selected Areas in Communications* (1997).

[19] McCanne, S. R. *Scalable Compression and Transmission of Internet Multicast Video.* PhD thesis, University of California, Berkeley, Dec. 1996.

[20] Mines, R. F., Frissen, J. A., and Yang, C. L. DAVE: A plug and play model for distributed multimedia application development. In *Proceedings of ACM Multimedia '94* (Oct. 1994), ACM, pp. 59-66.

[21] Ousterhout, J. K. *Tcl adn the Tk Toolkit.* Addison-Wesley, 1994.

[22] Perham, M., Smith, B. C., Janosi, T., and Lam, I. Redesigning Tcl-DP. In *Proceedings of the Tcl/Tk Workshop* (Boston, MA, July 1997).

[23] Raman, S., and Tung, T. -L. Mediaboard using the Scalable, Reliable Multicast toolkit, Dec. 1996. U.C. Berkeley term project and paper.

[24] Rowe, L., et al. *Continuous Media Toolkit (CMT).* University of California, Berkeley. Software on-line[5].

[25] Rowe, L. A., Patel, K. D., Smith, B. C., and Liu, K. MPEG video in software: Representation, transmission, and playback. In *High Speed Network and Multimedia Computing, Symp. on Elec. Imagin Sci. & Tech.* (San Jose, CA, Feb. 1994).

[26] Rowe, L. A., and Smith, B. C. A continuous media player. In *Proceedings of the Third International Workshop on Network and OS Support for Digital Audio and Video* (San Diego, CA, Nov. 1992), ACM.

[27] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications*. Internet Engineering Task Force, Audio-Video Transport Working Group, Jan. 1996. RFC-1889.

[28] Smith, B., Rowe, L. A., and Yen S. Tcl distributed programming. In *Proceedings of the Tcl/Tk Workshop* (Berkeley, CA, June 1993).

[29] Smith, B. C. *Implementation Techniques for Continuous MediaSystems and Applications*. PhD thesis, University of California, Berkeley, Dec.

1994.

[30] Tan, W., Chang, E., and Zakhor, A. Real time software implementation of scalable video codec. In *Proceedings of the IEE International Conference on Image Processing* (Lausanne, Switzerland, Sept. 1996).

[31] Taubman, E., and Zakhor, A. Multi-rate 3-d subband coding of video. *IEEE Transactions on Image Processing 3,* 5 (Sept. 1994), 572-588.

[32] Wetherall, D., and Lindblad, C. J. Extending Tcl for dynamic object-orientated programming. In *Proceedings of teh Tcl/Tk Workshop* (Ontario, Canada, July 1995).