Kolo and Nebesko: A Distributed Media Control Framework for the Arts

Eitan Mendelowitz
Computer Science Department,
University of California, Los Angeles
eitan@cs.ucla.edu

Jeff Burke
HyperMedia Studio,
School of Theater, Film and Television
University of California, Los Angeles
jburke@hypermedia.ucla.edu

Abstract

Increasingly, artists employ distributed multimedia in live performance, themed entertainment, and media art. One barrier they encounter is the interconnection of physically distributed sensors and computational units for the control of media elements. To address this need, UCLA’s HyperMedia Studio created Kolo, a Java-based framework for the collection of sensor data and the control of distributed media devices, and Nebesko, a scripting language for managing a Kolo network’s state. Kolo presents developers and application authors with a simple and consistent control framework for their distributed applications while Nebesko provides distributed state management for manipulation of the network. Nebesko also enables non-programmer to create and use Kolo networks. This paper discusses motivations for a distributed multimedia control framework for the arts, the qualities necessary for its success, the arts as an application domain for distributed multimedia research, and Kolo and Nebesko’s implementation.

1. Artistic applications of distributed multimedia

Multimedia is not limited to the display of images, video, and animation on screens. Artists often couple digital projection, environmental lighting control, real-time music/sound generation, animatronics, and other media elements with sensors such as motion detectors, accelerometers, and cameras to create compelling, immersive, and reactive physical environments. New communication technologies and digital control of media provide new avenues for expression. Exemplified by the works of Rafael Lozano-Hemmer, David Rokeby, Ken Goldberg, Christa Sommerer, and Laurent Mignonneau, as well as those described below and in [1], these technologies allow artists to create systems of relationships among sensed events in the physical world, purely virtual components, and digitally controlled media elements to engender new aesthetic experiences.

Artists working with these technologies are adopting distributed computing for many of the same reasons others turn to networks, including flexibility and scalability. Distributed computing allows artists, often dealing with very limited resources, to take advantage of economies of scale. It also allows practically unlimited channels of input and output without requiring specialized hardware. For example, *The Bush Soul* [2] used three networked computers to drive a wide-screen, high-resolution panorama, multi-channel spatial sound, and a responsive haptic interface without the purchase of special multi-head video cards or expensive high-end audio cards.

Unfortunately, artists new to distributed technologies face significant technical hurdles which continue to prove time-consuming for the more experienced. Especially difficult for neophytes is the interconnection of distributed devices [Table 1] each having different APIs, physical connection methods, and timing requirements. This initial step of interconnecting devices can consume a significant amount of time. Developers are often forced to create (or re-create) connections between devices and a variety of authoring platforms.

2. Supporting distributed multimedia in the arts

Kolo seeks to address this challenge faced by both novice authors and experienced developers by simplifying the process of creating networks of physically distributed input/output devices allowing them to be accessed in a consistent namespace with a simple API, regardless of their location. Such networks can then be used as a platform for experimentation and for application development thereby supporting new performances, installations, and themed entertainment experiences.
The Kolo framework is built on a series of simple assumptions: (1) Network access to reasonably low latency and high sample rate sensor data is baseline functionality for many new artworks; (2) control cannot be hidden behind abstractions that do not provide access to aesthetically relevant parameters [3]; (3) a control interface to most devices can be provided as collection of named objects whose values (numbers, strings, lists, etc) can reflect device inputs or drive the state of device outputs; (4) practical distributed control can be supported in a common namespace referencing these data objects, whose values can be read and set in one-time requests or periodic updates by any other process that wants them; (5) by allowing persistent functional relationships among the values of these distributed objects interaction can be supported directly within the framework without additional application development.

Nebesko is a high-level scripting language built on Kolo that goes further to support interactive object creation and distributed state management, allowing functional relationships, distributed across processes and physical machines, to evolve in a coordinated fashion in response to time or distributed events.

The simple network interface and consistent namespace allows experienced developers to quickly integrate control of new devices with very low ‘coding overhead’, and allows authors and scriptors to use pre-built distributed components at an appropriate level of abstraction. We have found that Kolo’s simple approach to presenting control interfaces is (so far) sufficient for a wide variety of devices and projects.

Kolo provides low-latency reliable data transport and distribution over UDP/IP, manages a distributed namespace among networked devices, and creates functional relationships between sensors and new media devices. Nebesko uses an augmented finite state machine to provide structures for monitoring and administration of distributed state and synchronization between network devices. It also allows scriptors an opportunity to experiment with Kolo applications without having to learn a traditional programming language.

Kolo and Nebesko are intended to support highly designed environments: installation artworks, performances, media shoots, experiments in dimensional entertainment, etc. Though they must be flexible and resilient, our delivery requirements are less stringent than many broad ubicomp visions, like MIT’s Oxygen project. [4] Also, unlike the proposed ARIA Framework [5], Kolo is intended for the control of multimedia devices and not multimedia content delivery. Media in such environments tends to be stored at the decoder location or be connected using other existing network-based methods supported by commercial software. Integrated control over these remote playback devices is often the sole element requiring custom coding, and this is what Kolo is designed to simplify and support.

3. The arts as a driver application for distributed computing

Artworks are deployed in museums, galleries, public spaces, and/or theaters and are used by some combination of an untrained public and trained users (‘performers’). Because of this, they must meet real world demands that cannot be recreated in a laboratory. At the same time, artistic works usually allow a certain amount of control not possible in full ‘real world’ deployment. Spaces and pieces can be constructed to mitigate deployment issues including power consumption and connectivity, and docents, performers, and volunteers can supervise and guide ‘users’. Thus artwork can provide a transitional venue for testing of technologies, more demanding than the lab yet not totally unconstrained. Kolo’s development is driven by the deployment requirements of such environments. Just as new distributed technologies provide new expressive possibilities for artists, these new works provide an application domain somewhere between the ‘real world’ of industry and the idealized world of the laboratory, fulfilling a recognized need:

“[C]reating reliable and robust systems that support some activity on a continuous basis is difficult. Consequently, a good portion of reported distributed media applications work

<table>
<thead>
<tr>
<th>Device</th>
<th>Input</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>X</td>
<td></td>
<td>Computer keyboard (sometimes modified with non-standard switches)</td>
</tr>
<tr>
<td>Mouse</td>
<td>X</td>
<td></td>
<td>Computer mouse (sometimes modified with non-standard switches)</td>
</tr>
<tr>
<td>DMX</td>
<td>X</td>
<td>X</td>
<td>Theatrical lighting control equipment</td>
</tr>
<tr>
<td>Serial</td>
<td>X</td>
<td>X</td>
<td>Standard computer serial (RS-232 and RS-422)</td>
</tr>
<tr>
<td>Programmable Logic Controller</td>
<td>X</td>
<td>X</td>
<td>Modular devices that provide analog to digital converters, relays, analog outputs, etc.</td>
</tr>
<tr>
<td>MPEG Decoder</td>
<td>X</td>
<td>X</td>
<td>Multi-channel hardware video playback</td>
</tr>
<tr>
<td>Video input</td>
<td>X</td>
<td></td>
<td>Composite video for simple machine vision (motion detection, tracking, etc)</td>
</tr>
<tr>
<td>Database</td>
<td>X</td>
<td></td>
<td>Relational databases</td>
</tr>
<tr>
<td>Martin Lighting Director</td>
<td>X</td>
<td></td>
<td>Ultrasonic position tracking and localization</td>
</tr>
<tr>
<td>RFID Reader</td>
<td>X</td>
<td></td>
<td>Radio frequency identification for object detection, identification and coarse localization</td>
</tr>
<tr>
<td>Macromedia Director</td>
<td>X</td>
<td>X</td>
<td>Screen based motion graphics, media playback and sensors through plug-in architecture</td>
</tr>
<tr>
<td>Cycling '74 Max/MSP</td>
<td>X</td>
<td>X</td>
<td>DSP, sound generation, and media playback, midi, and sensors through plug-in architecture</td>
</tr>
</tbody>
</table>
remains at the level of demonstrational prototypes that are not designed to be robust... 

The requirement is for real use of a system, deployed in an authentic setting...By pushing on the deployment of more living laboratories for ubicomp research, the science and practice of HCI evaluation will mature.” [6]

Artists work with potential and often unintended uses of new technology that challenge traditional assumptions in engineering methodology. (For example, Opera scenic designer Louis Daguerre’s pioneering work in photography [7]). Both SIGGRAPH’s annual conference and the ACM Multimedia conference have associated art shows. Wilson [1] argues that the cross-fertilization SIGGRAPH’s art show encourages accelerates developments in computer graphics. Kolo is intended to support novel choices of devices and interconnections.

4. Example works

The UCLA HyperMedia Studio is developing Kolo to support its investigation of emerging communication and media technologies’ impact on traditional production in theater, film, and television and to explore the new forms of expression that these technologies enable. [8]

We have developed installation and performance projects with students and faculty from film and television, theater, electrical engineering, and computer science. Described below are three distributed media works created at the Studio. Macbett and Invocation and Interference informed our creation of Kolo and Nebesko while Ecce Homology marked our first large scale use of Kolo, which was developed during the NSF-supported Iliad Project. [9]

Invocation and Interference (by Fabian Wagmister) was first installed in 2002 at “Interferences: International Festival of Electronic Art” in Belfort. Upon entering Invocation and Interference, participants see a group of monitors of different. [Figure 1]

Moving towards a particular monitor causes the video to zoom and pan. The footage displayed on the monitors (and audio) is selected from a large database. Ultrasonic distance sensors hidden in the base of the pedestals determine the distance of the viewer from the monitors.

Invocation and Interference required the interconnection of distance sensors, MPEG2 decoders, real-time video processing hardware, and a database of video footage, with system control software. The use of sensor data to trigger and affect the playback of stored media is typical of the interconnections Kolo is designed to support. As is common in media artworks, connections were made in an ad-hoc manner that was successful but not modular enough to be reused in other artworks.

Green and Petre’s influential paper [10] describes a cognitive dimensions framework for the evaluation of programming environments. They argue that usable end-user languages should have a ‘closeness of mapping’ between task-specific entities (e.g., pedestal state management) and program entities (e.g., functions and classes). The ad-hoc nature of the custom control software for state management and device control used in Invocation and Interference is typical of many artworks and arises in part from the lack support for these type of features in common development environments like Visual Basic, C/C++, Java, Max, and Director.

In addition to installations, distributed computing offers new potentials for performance. The UCLA Theater Department’s 2001 production of Macbett, by Ionesco (directed by Adam Shive), featured sensor technology that adapted the stage environment based on the actors’ movement. For example, staffs carried by the satire’s witches were outfitted with ultrasonic position trackers [Figure 2]. By coupling real time position information gathered and processed in one set of computers with lighting and sound controlled by another set of machines, the witches were given ‘supernatural’ control other their environment. For example, one witch could use circles of her staff to generate proportional amounts of storm sound and lighting.
where the director and actors do not have to limit themselves from improvisatory in order to remain synchronized with design elements. As sensing systems become more robust, this possibility becomes more and more appealing. [11]

The systems for *Macbett* were constructed to provide flexible and interactive authoring at several levels: low level development required regular field testing to develop performance requirements, integration of components had to allow quick testing of new ideas in rehearsal, and the individual components themselves had to have many parameters that could be adjusted without restarting the system. The deployment experience of *Macbett* has impacted Kolo’s development significantly and many of these qualities have been carried over into its design, specifically with regard to runtime interactive controls. The need for iterative development and rapid change during rehearsal emphasize of two more of Green and Petre’s cognitive dimensions: *Viscous* environments make revisions difficult, while *imposed guess-ahead* forces premature commitment to a coding approach. Together they force developers to invest in a particular course of action before seeing results (*guess-ahead*) and then, if the results are not satisfactory, make change difficult (*viscosity*).

Kolo aims to be *low viscosity* and require *minimal guess-ahead* by (1) supporting online adjustment wherever possible; (2) avoiding unnecessary premature commitments with features like flexible data types; (3) offering interactive tools including the Nebesko scripting language and a GUI for runtime monitoring and manipulation of the Kolo network.

5. Curriculum integration

The HyperMedia Studio also supports courses using new technology in theater, film, and new genres. Like similar courses at other institutions, these courses are taught using Cycling 74’s Max/MSP and Macromedia Director, development environments also used in many new media artworks.

Max is a visual data-flow based programming language originally designed for algorithmic music composition. Over the years, artists creating interactive environments have adopted it. Like many data flow visual programming languages MAX is highly *viscous* and does not scale to large applications well. [10] We have also found that Max has very few constructs for doing complex state or time management. While it is very easy to create a “patch” for simple interactions, complex coordination of state evolution over time is quite difficult. Network applications are currently limited to using “Open Sound Control.”

Macromedia Director was originally designed for interactive CD-ROM creation. It uses an animation metaphor of a stage (the screen), a cast, and a score. Director has a powerful scripting language called Lingo and, like Max, benefits from a large number of third party plug-in “xtras” for support of external devices including sensors and actuators.

Unfortunately for those developing physically instrumented environments Director was created for screen based interaction. Director has a bias for non-overlapping states of interaction. Like Max, traditional Director networking is fairly limited: Director networking is typically done through the “Shockwave Multi-User Server”, which is targeted at creating collaborative multi-user applications. [12]

Our courses use both Director and Max. For example, we use an ultrasonic position tracking system. In many cases, students would like to control sound in Max/MSP based on position data from this system, while also controlling visuals on one or more Director processes from the same data. By creating Kolo drivers for the tracking system, Max, and Director, the latter two can easily share network access to data from the tracking system using Kolo.

These drivers include a Kolo Director ‘xtra’ and an Open Sound Control interface. (We are currently developing a native Max ‘object’ as well.) These Kolo drivers allow Max and Director processes not just to receive data but to be considered just like ‘devices’ themselves in a Kolo network. Max and Director are widely used and well suited to sound and screen based graphics respectively; it is not the goal of Kolo to duplicate their functionality, but rather to provide a platform for well-integrated use of these and other multimedia technologies in the arts.

6. New work using Kolo

Informed by the creation of works like *Macbett* and *Invocation and Interference*, as well as our courses, the initial version of Kolo was completed in spring of 2003 and was first deployed in *Ecce Homology* (2003-4), an immersive installation by West, Burke, Kerfeld, Mendelowitz, Holton, Drucker, and Yan. In *Ecce Homology* a computer-vision based user interface allows multiple participants to select genes from the human genome for visualizing BLAST, a primary algorithm in comparative genomics. [13]

The piece was successfully installed in the UCLA Fowler Museum of Cultural History, November, 2003 – January, 2004 and interacted with by thousands of museum patrons. Kolo successfully allowed our collaborators, bioinformatics application developers, to create software modules for the artwork without having to write network communications or distributed state management code. In fact, each of the developers implemented their modules separately knowing only that
integration would take place via a shared namespace of control parameters. After a quick introduction to the Kolo API, the developers exposed each module’s functionality over the Kolo network in a matter of hours.

The modules integrated via Kolo include, computer-vision, hand motion analysis, blast, graphics (3 graphics modules each running on a separate machine are used collectively to project a forty-five foot wide image over five projectors), and state control.

Because Nebesko was not yet complete, the state-management module was written in Java specifically for the project. Because of the large number of interdependencies that relied not only on sensor data and current state but also on past state and elapsed time coding was tedious and error prone. For example, changing a precondition for a given state could mean numerous code changes over multiple (dispersed) lines. These same problems arise in many types of interactive development including gaming:

“The major complication in C/C++ based AI and game logic programming lies in dealing with events that take a certain amount of game time to complete, and with events which are dependent on aspects of the object’s state. In C/C++, this results in spaghetti-code that is hard to write, comprehend, maintain, and debug. UnrealScript includes native support for time, state, and network replication which greatly simplify game programming.” [14]

As UnrealScript does for gaming, Nebesko seeks to address these problems by inherently supporting state transitions based on interconnections, distributed object state, and time.

As illustrated by these works, artists need closeness of mapping to device parameters (Invocation...). Interconnections between devices must be fluid, allowing rapid online modifications during development and testing (Macbett). Finally the complex nature of these works requires tools for managing time and distributed state (Ecce Homology).

7. Kolo and Nebesko

Kolo consists of a Java API that allows developers to easily incorporate new sensors and media elements in a consistent framework for control data transport. A higher level Java API allows application authors to interconnect and reuse these supported devices with the same interface regardless of whether they are running on a local or networked Kolo process. In addition to using Director and Max to create Kolo applications via our interface objects, scripts can use Nebesko.

The Kolo API and Nebesko are implemented in Java because of portability, freely available runtimes, and the large base of Java developers. Because Kolo is designed to incorporate platform- and hardware-specific device drivers, a C/C++ Java Native Interface (JNI) is provided. Using this JNI, a wrapper for the Kolo API is available in Macromedia’s Director via an “xtra” and will soon be available in Cycling ’74 Max/MSP though a set of native Max objects.

To promote quick adoption by developers, Kolo’s API is designed with relatively few core abstractions (especially when compared to such traditional distributed frameworks such as CORBA [15]). Kolo only requires the understanding of six concepts: knobs, values, subscriptions, groups, relationships, and arbitrators.

8. Knobs

The Kolo Network Object (Knob) is Kolo’s basic building block, and can be instantiated using either the Java API or Nebesko. Knobs are organized into trees: each knob may have at most one parent and an arbitrary number of children. Every knob has a name and a knob may not share its name with a sibling. Thus an ordered list of ancestors’ names, a path, uniquely identifies a knob. Knobs can be accessed from any process on the Kolo network as if they were local and can also be created on a remote Kolo process.

Knobs may have values, which can represent physical sensor readings, internal state, or any abstract quantity. Knobs can be written so that changes to these values can cause side effects, for example, the control of media devices or physical actuators. The following code snippet shows the instantiation of a knob for DMX control of theatrical lighting and one for ultrasonic localization.

Java:
new DmxEthergateOut(knobManager, new Path("root.dmxOut"));
new MldKnob(knobManager, new Path("root.mld"));

Nebesko:
create(DmxEthergateOut, root.dmxOut);
create(MldKnob, root.mld);

Note that while the two knobs perform different functions and create different hierarchies of children to expose functionality [Figure 3], they are both created in the same manner. Also, all of the following examples using these two knobs would work without revisions if these two knobs were started on separate processes or machines.

The Kolo API allows the easy creation of new knob classes that expose the functionality of a sensor or media device to the kolo network through the knob’s values and/or the values of its children. Our hope is that as the community of Kolo users grows so too will the number of supported devices.
8. Values

One difficulty in interconnecting heterogeneous data objects over a network is data type agreement. This led us to develop a single polymorphic value type for Kolo. Internally, a value is a long, double, string, Boolean, list (an ordered collection of values), or is undefined. When a value is accessed from the Java API the developer must request it in a primitive type, and the framework casts values to the requested type transparently. Nebesko does its best to handle casting automatically at run-time. The need for polymorphism in Nebesko is even more pronounced than the need in Kolo, as the target user audience (scripting-capable artists) is usually most familiar with languages that are not strongly typed.

To retrieve values in Java:

```java
// First, get a reference to the knobs (from their path)
Knob stageLight = knobManager.getKnob(new Path("root.dmxOut.1.1.dmxValue"));
Knob actor1DistToStageFront = knobManager.getKnob(new Path("root.trackers.1.distanceTo.1.downStage"));
// Then, get the value using
Value lightValue = stageLight.getValue();
Value downStageDist = actor1DistToStageFront.getValue();
// Next, do something with that value
System.out.println("The light value is " + lightValue.getStringValue());
```

In Nebesko, there is no need to get a reference to a knob since you can use a path in any expression and its value is cast automatically.

```java
print("the light value is " + root.dmxOut.1.1.dmxValue);
```

Note that both the Kolo and Nebesko code works identically for knobs created local to this process or remotely.

9. Subscriptions

In Kolo, values are set through subscriptions. Knobs receive their value from one or more subscriptions. A subscription can either be to another knob’s value (which may change over time) or to a constant value. A knob may subscribe to another knob regardless of its location on the network. Subscriptions may be time based (periodic), change based, or a hybrid of time and change based. A hybrid subscription is characterized by a minimum period $T_{min}$, a maximum period $T_{max}$, and a maximum change $d$. If $t$ is the elapsed time since the last data update and $d'$ is the change in value of the knob since the last updated then, a new value is pushed out when ($T_{min} < t' < T_{max}$ and $d' >= d$) or when ($t' = T_{max}$). These hybrid subscriptions are useful for sensor data that is expected to have episodes of rapid change followed by long periods of quiescence. It should be noted that periodic subscriptions are a special case when $T_{min} = T_{max}$ and change-based subscriptions are a special case when $T_{min} = 0$, $T_{max} = \infty$, and $d = 0$.

The default behavior of the Kolo API’s base knob, the SimpleKnob, is to determine its value from its last subscription received.

To control the intensity of the light created in the above example based on the distance of a tracked actor from the audience, the intensity of the light can be subscribed to the actor’s distance knob.

```java
stageLight.subscribeTo(actor1DistToStageFront);
in nebesko stageLight := actor1DistToStageFront;
```

To turn off the light we could subscribe it the constant value 0. In Java, `StageLight.subscribeTo(0);` and with Nebesko, `stageLight := 0;`

10. Groups

Developers of distributed media applications often wish to address collections of individual devices in unison (e.g. a bank of lights in different locations but focused on the same area). A Kolo group is a knob that maintains a list of members (as a child knob). A group’s value is a list that contains the values of all its members. Any operation that can be performed on a knob can be performed on a group and the group applies the operation to all of its members. Any operation applied to a child of a group is applied to the children of its members. Java:

```java
ValueList memberList = new ValueList();
memberList.addValue("root.dmxOut.1.1");
memberList.addValue("root.dmxOut.1.2");
memberList.addValue("root.dmxOut.1.3");
Knob lightGroup = new Group(knobManager, new Path("root.lights"));
lightGroup.getChild("members").subscribeTo(memberList);
Knob intensities = knobManager.getKnob(new Path("root.lights.dmxValue"));
i...
```
and Nebesko:
create(Group, root.lyights);
Root.lights.members:= ["root.dmxOut.1.1", "root.dmxOut.1.2", "root.dmxOut.1.3"]; Root.lights.dmxValue := 255;

Kolo’s unique provision for groups arises from our experience with live performance, which frequently requires simultaneous control of heterogeneous elements (e.g. light and sound intensity at a particular moment).

11. Relationships

A relationship is a knob whose value is defined as a function of its subscriptions. For example, a light’s intensity attribute can be made equal to the minimum of two actors’ distances from their audience by creating a relationship that returns the minimum of its subscriptions and subscribing the light to that relationship. In Java the relationship would look like:

```java
public class Min extends Aggregator {
    public Min(KnobManager knobManager, Path path) throws KoloException {
        super(knobManager, path);
        double min = Double.POSITIVE_INFINITY;
        Enumeration sources = subscriptions.elements();
        while(sources.hasMoreElements()) {
            Subscription source = (Subscription) sources.nextElement();
            double value = source.getValue().getDoubleValue();
            if(min > value){ min = value; }
        }
        set(min);
    }
}
```

Note that only one method, `evaluate`, was overwritten in the creation of this relationship.

To use this new class to set the value of the DMX light requires the following:

```java
Knob min = new Min(KnobManager, new Path("root.min"));
min.subscribeTo(actor1distToStageFront);
min.subscribeTo(actor2distToStageFront);
StageLight.subscribeTo(min);
```

In Nebesko, relationship creation is considerably less verbose. To create and use a new relationship, a scriptor simply needs to write an expression, for example:

```java
Root.lights.min := min(actor.1.distance, actor.2.distance);
```

Unlike in Java, Nebesko do not create a new class. Rather Nebesko allows the creation of relationships knobs that interpret and execute any Nebesko expression.

Relationships provide a powerful way for artists to specify dynamic control over media based on real-time sensory input. [9]

12. Arbitrators

In complex environments, multiple relationships may vie for control of a single attribute,[16] Arbitrators are privileged relationships that examine competing subscriptions and arrive at a single value for a knob. For example, Min can be used as an arbitrator.

In the relationship example we subscribed the stage light to the min knob. If we then subscribe stage light to the position of a third actor the light would no longer be subscribe to Min. If we wanted to assure that the light’s intensity is always determined by the minimum of all of its subscriptions we could assign min as an arbitrator.

```java
StageLight.setArbitrator(new Min(knobManager, new Path("root.min")));
StageLight.subscribeTo(actor1distToStageFront);
StageLight.subscribeTo(actor2distToStageFront);
StageLight.subscribeTo(actor3distToStageFront);
```

Or in Nebesko

```java
SetArbitrator(StageLight, new Min(root.min));
// we are using the Java defined Min
stageLight := actor1distToStageFront;
stageLight := actor2distToStageFront;
stageLight := actor3distToStageFront;
```

Arbitrators allows for a consistent and well-defined manner of dealing with conflicting relationships.

13. Implementation

A peer-to-peer network of KnobManagers handles Kolo’s network functionality. A KnobManager acts as a daemon for a Java process and supports the (1) creation, referencing, and destruction of knobs, relationships, and groups, (2) the creation and termination of subscriptions, and (3) the assignment (and un-assignment) of arbitrators. Every Kolo process must have at least one KnobManager and, within a Kolo network, each KnobManager must be uniquely named. When a knob is created, its presence is broadcast to all KnobManagers, which are collectively responsible for maintaining a consistent namespace.

KnobManagers allow developers to work with remotely managed knobs as if they were local objects. The KnobManager even permits working with ‘virtual’ knobs that do not exists as objects registered in the namespace. When a process attempts to perform an API call on a ‘virtual’ knob, its closest registered ancestor handles the call. For example, a large array of theatrical lights may have hundreds of lights. Rather than clutter the knob manager with the creation of hundreds of knobs, a single light-array knob could handle all the requests while still allowing the developer to access individual lights through the familiar Kolo API, for example getKnob on the ‘virtual’ knob representing a single light quality’s path (e.g. `getKnob("root.lightArray.400.intensity")`).

When possible, KnobManagers attempt to minimize network traffic by aggregating subscription. For example if knob A and knob B both are managed by the same KnobManager and subscribed to knob C, the KnobManager will aggregate the subscriptions, allowing C’s values to be sent only once over the network.

KnobManagers communicate with one another via a bus supporting both unicast and broadcast messages, including remote function calls. These managers may
organized into groups to segregate broadcast traffic. All control messages between managers, e.g. for knob creation notification, are multicast to the entire group. If a manager is a member of more than one group, any broadcast message received from one group is forwarded to all its other groups. This allows managers to be organized to optimize network traffic.

The bus uses UDP for all network communication because of its low latency and multicast properties. The Kolo bus currently uses Spread [17] as its transport for unicast, multicast, and reliable UDP transport.

14. Conclusion

The arts are a rich area for experimentation with distributed media systems and such systems to offer enormous creative potential. To aid our exploration and to help address the steep development curve that many artists face in working with such systems, we created Kolo, the framework for distributed media control described above. Conceived with Kolo but only recently implemented, Nebesko is a scripting language specifically designed for managing distributed state in Kolo networks. In addition to its use in classes taught at UCLA’s HyperMedia Studio, Kolo has successfully been deployed in Ecce Homology and our Advanced Technology for Cinematography project [18]. We continue to refine Kolo and Nebesko even as they are in their current form to enable new works and plan to release them as open source software in 2005.

15. Acknowledgements

Significant contributions to the design and implementation of Kolo were made by HyperMedia Studio researchers Joseph Kim and Adam Pingel. The authors would also like to thank Fabian Wagmister, director of the HyperMedia Studio, for his input and support throughout the development process. Initial work on Kolo was supported by NSF under grant IIS-0207869.

16. References


