Boyle, M. and Greenberg, S. Rapidly Prototyping Multimedia Groupware. Report 2005-786-17, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. April.

Rapidly Prototyping Multimedia Groupware

Michael Boyle and Saul Greenberg

***Abstract*—Multimedia groupware systems provide rich support for distributed team work. Yet effective design of these systems is difficult because they must cater to complex human and social factors. Rapid prototyping partially mitigates this, for it allows designers to build, deploy, test and quickly evolve design ideas. The problem is that multimedia groupware is hard to prototype because distributed multimedia systems are complex to implement. To solve this problem, we offer the Collabrary, a toolkit specifically designed for easy prototyping of multimedia groupware. The Collabrary blends real-time streaming multimedia, asynchronous shared application state, and novel multimedia analysis and manipulation algorithms to provide rich functionality for distributed teamwork. Implementing core functionality – multimedia capture, analysis, manipulation, transmission and rendering – is trivial. The Collabrary also affords lessons that inform the design of universally accepted toolkits for building distributed multimedia systems: we illustrate why toolkits should be accessible for learnability, lightweight so easy ideas are easy to build, and flexible so that novel unanticipated ideas are possible to implement.**

***Index Terms*—distributed multimedia groupware, prototyping.**

I. Introduction and Motivation

INCREASINGLY, groupware systems are incorporating multimedia functionality. Traditional systems such as Instant Messaging now add pictures, voice and video to what was once a simple text channel. Scores of experimental groupware systems supporting distributed colleagues now rely on multimedia as first-order data types [1,2]. In general, these *multimedia groupware systems* blend ephemeral streaming of multimedia data with persistent shared application state.

Yet multimedia groupware design is challenging, for it must cater to complex human and social factors if it is to support both individual and team work practices [3]. This leads to design uncertainty. One well-known method of handling this design challenge is prototyping, i.e., "artifacts that simulate or animate some but not all of the features of the system" [4]. Prototypes vary in fidelity and purpose, but all lead to iterative design. A *low-fidelity* prototype might consist of ideas sketched on paper to quickly get a sense of the major design concept. A *medium-fidelity* one can be a first-cut subsystem implementation that helps one determine factors such as feature usability and/or system performance. A *high-fidelity* prototype can be an extensive interactive user interface that can be deployed to users and marketers for feedback.

However, satisfying the socio-technical design problem of multimedia groupware requires *working system* prototypes: initial implementations of the system deployable to resilient users who do not mind occasional glitches and restarts. These prototypes may: be constrained to idealized hardware, software, and network platforms; be deployed only over a secured network or within benign social situations to alleviate security concerns; or contain only a subset of expected functionality. This limited deployment is extremely valuable. It helps the designer uncover socio-technical issues that are otherwise hard to detect except under extended, real use. As Buxton [5] notes, working groupware prototypes permit "living with the technology" that is critical to identifying and solving the most pervasive and troublesome problems.

The problem is that multimedia groupware is hard to prototype because distributed multimedia systems are complex and difficult to implement. As a solution, this paper offers the Collabrary, a toolkit specifically designed to allow developers to easily prototype distributed multimedia groupware.

A. Toolkits for Multimedia Groupware

Greenberg [6] argues the need for easy-to-program toolkits for novel interface areas: "By removing low-level implementation burdens and supplying appropriate building blocks, toolkits give people a 'language' to think about these new interfaces, which in turn allows them to concentrate on creative designs." In Allan Kay's words, "easy things should be easy; hard things should be possible".

Yet it has been hard to develop a toolkit for building distributed multimedia systems, because these systems require a wide gamut of hardware and software infrastructure. On the multimedia side, there needs to be operating system support for accessing multimedia hardware, and algorithms and APIs for capturing, manipulating, compressing, and rendering multimedia data. On the network side, multimedia data must be distributed to all machines participating in the groupware session. This distributed groupware aspect is complex, for it may require basic communication services (e.g., TCP, UDP and multicast IP), time-synchronization of multiple concurrent data streams (e.g., RTP [7]), and session management (e.g., SIP [8]). It may also require protocols for coordinating application behavior and sharing state (e.g., RPC, XML Web Services [9]), notification services (e.g., Elvin [10]), relational databases and/or distributed shared memory (e.g., JSDT [11]).

A number of distributed multimedia toolkits excel at providing robust and high-performance streaming multimedia services to applications (e.g., [12,13]), yet they often omit rich

Manuscript received April 30, 2005. Work supported by NSERC & MSR.

M. Boyle and S. Greenberg are with the Dept. of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, AB T2N 1N4 Canada (phone: 403-220-6087; e-mail: {saul *or* boylem}@ cpsc.ucalgary.ca).

support for sharing other sorts of application data necessary for groupware. Similarly, groupware toolkits that support application data sharing (e.g., [14]), do not robustly handle multimedia data. Using these two classes of toolkits together in the same application is often awkward, for they usually have incompatible programming environments and idioms. Commercial counterparts of these research systems are often no better. Some, such as Microsoft NetMeeting [15], are self-contained applications that can be remotely controlled in only limited ways. Those that are more flexible are often incredibly complex to learn to use, e.g., Microsoft DirectShow [16] and JMF Java Media Framework [17].

In this paper, we present the Collabrary, a toolkit we developed to aid the rapid implementation of working system prototypes of multimedia groupware applications. It is implemented as a Microsoft COM object library and can be used with a variety of popular rapid application development platforms (e.g., Visual Basic, C#), scripting languages (e.g., Python), and lower-level languages like C++. In the sections that follow, we explain the requirements behind a toolkit supporting prototyping of multimedia groupware, and illustrate how the Collabrary meets these requirements.

II. Toolkit Requirements.

The need to implement working system prototypes *rapidly* makes special demands of the toolkit used. In particular, it must *trivialize common programming tasks* so that prototypes can be built and rebuilt from scratch quickly. This allows end-programmers to focus the bulk of their attention on implementing novel aspects of the design, and the freedom to make substantial, deep revisions to their prototypes without lamenting time lost on prior unsatisfactory versions. In the Collabrary, we have sought to make common programming tasks trivial in three important ways that will be discussed extensively in the remainder of this paper.

- *Accessible*: the toolkit should be easily to learn, where novice toolkit users can develop applications after only modest training. To meet this goal, we emphasized simple programming idioms already familiar to end-programmers.
- *Lightweight*: common tasks should require very few lines of code to implement, and the code needed should use simple programming statements. To meet this goal, we designed the Collabrary to provide rich functionality that is difficult or tedious to implement from scratch. It performs many important tasks automatically or as default behavior.
- *Flexible*: the toolkit should be supple enough to design a wide range of unanticipated applications. To meet this goal, we have design the Collabrary for flexibility. It provides direct access to internal multimedia data structures (so they can be altered). It uses programming idioms borrowed from other application domains already proven successful and flexible. It allows optional customization of default behaviors.

We also identify four common multimedia groupware programming tasks that we strive to make accessible, lightweight, and flexible in the Collabrary.

```
class MainForm : Form {
  PictureBox pictureBox;
  Camera camera=new CameraClass();
  Microphone mic=new MicrophoneClass();
  Speaker spkr=new SpeakerClass();
  MainForm() {
    camera.Captured+=…camera_Captured…;
    camera.Size=…320x240;
    camera.FrameRate=15;
    mic.Captured+=…mic_Captured…;
    mic.Recording=true;
  }
  void camera_Captured(IPhoto frame) {
    pictureBox.Image=…frame…;
  }
  void mic_Captured(IWaveform samples) {
    spkr.Play(samples);
  }
  [STAThread] static void Main() {
    Application.Run(new MainForm());
}}
```

Fig. 1. Capturing and rendering multimedia: `Collabrary.Camera`, `Collabrary.Microphone`, and `Collabrary.Speaker`.

- *Capturing* multimedia must be trivial.
- *Rendering* multimedia must be trivial and compatible with a rapid application development GUI toolkit.
- Simple multimedia *manipulation & analysis* must be trivial, while implement advanced manipulations must be possible.
- *Transmitting* multimedia and other shared groupware state/data must be trivial and done in a way that is natural for end-programmers to think about.

In the following sections, we describe how a Collabrary end-programmer achieves these four common programming tasks. We use snippets of C# code to illustrate the toolkit in action, and discuss important aspects of the API relevant to the task.

III. Capturing Multimedia

Perhaps the most common task asked of a multimedia toolkit is audio/video capture. Even this can be difficult, as it involves enumerating capture devices available on the computer, accessing a source, configuring capture properties (e.g., video frame size and rate, audio sampling rate) and then controlling capture. The Collabrary trivializes multimedia capture by offering simple hardware abstractions and by notifying the programmer of multimedia acquisition through a familiar event-based paradigm. Fig. 1 shows Collabrary program code that illustrates trivial audio/video capture.

A. Hardware Abstractions

The Collabrary provides end-programmers with succinctly-named classes that encapsulate high-level abstractions of multimedia hardware. In Fig. 1, video and audio are captured by a `Camera` and a `Microphone` object, respectively. The key is that these abstractions remove unnecessary programming complexity while adding robustness.

The `Camera` class will be used to illustrate six ways the Collabrary makes multimedia capture simple yet robust. The principles apply equally to audio and file-based multimedia input (not shown in the figure).

1) It works with any ‘plug and play’ camera; the programmer does not need to specify device-specific properties.
2) The program runs without exception even if no camera is

attached to the computer: the `Camera` object automatically inserts a 'test pattern' image in place of live video.
3) The program continues to function even if the camera is detached, and automatically connects as soon as a new camera is attached.
4) Multiple copies of the `Camera` object can simultaneously share access to the same camera device.
5) Objects require little initialization before they may be used because their properties are embedded with useful defaults. These can be overridden: the `Camera.FrameRate` default of 0 fps, which indicates manual capture, is reset to 15 fps to start automatic capture. The frame size is also specified.
6) No 'shutdown' or 'cleanup' code is required: the objects gracefully release resources when garbage collected.

Comparatively, other toolkits are heavyweight. For example, JMF [17] does not automatically provide these six features.

B. Event-Oriented Architecture

To promote accessibility, the Collabrary manages multimedia capture using the event-driven callback paradigm familiar to GUI programming. When multimedia is captured by a Collabrary object, the object "raises an event." The end-programmer can attach a callback method to handle the event.

As seen in Fig. 1, these event handlers for the video camera and microphone are attached in the `MainForm` constructor. The `camera_Captured` method handles the `Camera.Captured` event and is invoked each time a video frame is captured, where the captured frame is passed as a parameter to the event handler. Audio is treated similarly, where the `mic_Captured` method handles the `Microphone.Captured` event. Periodically, after collecting a small block of audio data (by default, every 50 ms of audio) the `Microphone` object will raise its `Captured` event.

There are two main advantages of this event-based idiom. First, it uses an asynchronous programming paradigm that end-programmers will already be familiar: it uses the same event dispatch and handling mechanisms, syntax, and programming patterns as the GUI toolkit. Second, read/write access to multimedia is provided directly as a natural consequence of handling the events. However, there is a trade-off: multimedia pipeline architectures (e.g., [17]) timestamp data in all streams using a common reference clock. The pipeline manager uses these timestamps to ensure audio and video streams are tightly synchronized. Although it is possible for Collabrary end-programmers to implement timestamps themselves, our event-driven architecture does not yet offer this synchronization (but see Section V.d)

IV. Rendering Multimedia

The second most important task that a multimedia groupware toolkit must support is trivial rendering of multimedia that has been captured and transmitted.

Fig. 1 shows how a programmer trivially renders the captured audio/video to the local machine's GUI display and sound hardware. Other multimedia toolkits only render video into a widget that it provides. Yet video rendering with the Collabrary makes use of the image rendering classes and widgets provided by the GUI toolkit (e.g., the PictureBox class in C#). Using standard widgets for rendering affords three critical advantages.

First, it keeps the toolkit accessible. End-programmers do not need to learn how to use a new widget. When a multimedia toolkit provides its own video rendering widget, this widget often provides an API that is inconsistent with that of the GUI toolkit. This makes it difficult for end-programmers to get started using the multimedia toolkit.

Second, it keeps the toolkit flexible**.** New forms of user interactions with the video display via the mouse/keyboard can be implemented using the UI programming patterns and practices already familiar to them. For example, when a toolkit provides its own video rendering widget, it often does not expose mouse or keyboard input event bindings that support rapid prototyping of new forms of user interaction.

Third, relying on the GUI toolkit for video rendering keeps the multimedia toolkit *interoperable*. The widget is assured to work in perfect harmony with the rest of the GUI toolkit and can be easily composed with the rest of the application's GUI. Furthermore, when the GUI toolkit uses a visual interface designer, the designer can be used to configure video renderer properties. When the multimedia toolkit provides its own video rendering facilities, it is often implemented as a popup window that cannot be visually integrated into the rest of the application's GUI and cannot be configured with the visual interface designer. In extreme cases, the multimedia toolkit may be entirely incompatible with the GUI toolkit and impossible to use.

V. Multimedia Manipulations

The Collabrary is intended to support the rapid prototyping of *novel* multimedia groupware applications. In these kinds of applications, the designer may want to manipulate audio and video in a variety of ways. To support the rapid prototyping of novel multimedia interactions, the Collabrary must make analyzing and manipulating audio and video trivial.

A. Pre-Packaged Manipulations

Some manipulations can be anticipated, and consequently the Collabrary offers a number of pre-packaged audio and video manipulations. One example is background subtraction and replacement. Fig. 2 shows a modification to the code in Fig. 1 to implement background subtraction/replacement with the `frame.Subtract` method. Other examples include: video filters such as pixelization, blurring, and posterizing; image composition such as alpha blending; and, raster graphics primitives.

The Collabrary also has a few analysis algorithms built-in. For example, Bradski's CAMSHIFT face-tracking algorithm

```
void camera_Captured(IPhoto frame) {
  Photo newbkg=new CameraClass();
  newbkg.Load("newbkg.jpg");
  frame.Subtract(frame, newbkg, …);
  pictureBox.Image=…frame…;
}
```

Fig. 2—Background subtraction and replacement is trivial in the Collabrary.

[18] is implemented by a `Collabrary.FaceTracker` object. With this object, the position and size (in pixels) of a face in the video can be obtained with the addition of a few simple method calls. As another example, a motion-detection algorithm can be prototyped in just a few lines of code that use image subtraction without background replacement and compare the `Photo.PSNR` (peak signal-to-noise ratio) value of the difference frame against a threshold.

B. *Composing Effects from Pre-Packaged Manipulations*

Custom effects can be easily achieved by composing several manipulations and analyses together. For example, Fig. 3 illustrates a sophisticated custom video manipulation, inspired by [19], built by composing the pre-packaged analysis and manipulation algorithms provided by the `Collabrary.Photo` object. Just 30 lines of code completely implement what is seen. Low-frame rate video snapshots are visually blended together to show a history of activity i.e., a frame is alpha-blended to the history of recent video frames only when it differs markedly from the previous snapshot in the history. Thus we see 'ghostly' versions of the person in Fig. 3 as he has moved about. Also, a scrolling EKG-like diagram appears at the bottom of the video. This diagram represents the activity level in the video over time. The motion detection scheme mentioned previously is used to detect changes, and the `Photo.DrawLine` method is used to draw the chart lines.

C. *Custom Direct Read/Write Manipulations*

The Collabrary multimedia data types (`Photo` for video frames and `Waveform` for audio sample blocks) provide end-programmers with direct read/write access to the buffered data. Two types of access are provided. One type provides 'safe' high-level (but only modestly efficient) methods to read and write pixels and audio samples as though they were in a 2D array. The other type is 'unsafe' but highly efficient access, where the end-programmer acquires a pointer to the underlying data buffer in memory. This pointer can be used for high-performance implementations of very sophisticated analysis and manipulation algorithms. This allows end-programmers the opportunity to quickly prototype a broad spectrum of in-place or out-of-place transforms.

D. *Event-Oriented Architecture Eases Manipulation Tasks*

The event-oriented capture pipeline architecture used in the Collabrary makes implementing multimedia manipulations much more accessible and lightweight compared to toolkits based on stream-oriented architectures. For example, in order to gain direct access to multimedia data in the pipeline using the JMF the end-programmer must write a filter class and insert one of the filter objects into the pipeline at an appropriate place. Writing a filter is conceptually difficult. The programmer must create a class that implements interfaces required by the pipeline manager. These interfaces are extremely generalized, however: both audio and video media types are presented as byte arrays instead of a rich image or audio type. This makes it awkward to draw raster graphics or compose images within a filter. Ultimately, the programmer is forced to implement mundane code that is irrelevant to the real work of the filter. This makes the toolkit less fit for rapidly prototyping multimedia groupware.

Fig. 3. Visual and graphical traces of activity, implemented by composing various pre-packaged Collabrary manipulations.

VI. Transmitting Multimedia

Lastly, the Collabrary makes distributing multimedia data across networks to other computers trivial. Implementing this task makes use of: session management protocols; audio/video codecs (e.g., [20]); transport protocols that account for late, lost or out-of-order messages; and, protocols for negotiating, monitoring and regulating quality-of-service (QoS).

This task is the most difficult to implement robustly, but is essential for deployable groupware. First, the algorithms and protocols themselves are conceptually complex. Second, implementations must be carefully coded to meet performance requirements and robustly handle a myriad of possible exceptions. While there are many toolkits to insulate the end-programmer from the gory details of implementing standards robustly, they often require:

- set up/administration of network services that are separate software downloads e.g., SIP requires proxy servers;
- network features unavailable to intended prototype users, e.g., OpenMash [12] requires multicast IP; or,
- multiple toolkits to be used concurrently e.g., RTP does not provide a guaranteed lossless in-order delivery stream for arbitrary-length messages, making it inappropriate for sharing certain kinds of application state information.

Consequently, the Collabrary does not implement popular Internet engineering standards like SIP and RTP because some of the programming idioms used in these standards are not trivial enough to support rapid prototyping.

Fig. 4 shows code that implements a simple *n*-way videoconferencing application. For brevity, audio support has been omitted, but if included it would follow similar programming patterns as video. As shown in Fig. 4, the Collabrary uses a markedly different architecture for transmitting multimedia. The centerpiece of this architecture is the *shared dictionary*. This distributed data structure blends programming idioms from notification servers [10], groupware programming [13], distributed shared memory systems [21], Model-View-Controller architectures [4], and filesystems. In the remainder of this section, we illustrate how this shared dictionary is used to rapidly prototype multimedia groupware.

```
/* Initialisation */
Hashtable windList=new …;
Camera camera=new …;
camera.Size=…320x240;
camera.FrameRate = 10;
VideoCodec codec=new …;
codec.Open("MJPEG",320,240,…);
SharedDictionary sd=new …;
sd.Open("tcp://www.host.com:video");

void sd_Opened(…) {
  /* Tie data to connection status */
  sd[sd.Me+"/.transient"]=sd.Me;
  /* Store a user display name */
  sd[sd.Me+"/name"]="Mike";
}
```

```
void sd_Closed(…) {
  /* Prompt user to reconnect */
  if(sd.Troubleshoot(…)) {
    retries=1;
}}

void sd_Entered(string id) {
  /* Create separate GUI window */
  VideoWin win=new VideoWin(id);
  win.Show();
  winList[id]=win;
}

void sd_Exited(string id) {
  /* Dispose of GUI window */
  VideoWin win=windList[id];
  win.Close();
  windList[id]=null;}
```

```
void camera_Captured(IPhoto curFrame) {
  /* Store compressed video frame */
  sd["/user/"+sd.Me+"/video"]=
    codec.Compress(curFrame);
}

public VideoWin(string id) {
  /* Set window caption */
  this.Text=sd[id+"/name"]…;
  /* Subscribe to video */
  Subscription video=sd.Subscribe(id+"/video");
  video.Notified+=…video_Notified…;
}

void video_Notified(…object val…) {
  Photo p=videoCodec.Decompress(val…)…;
  pictureBox.Image=…p…;
}
```

Fig. 4. Implementing an *n*-way video conferencing application with the Collabrary shared dictionary.

A. *Centralized Server Network Architecture*

End-programmers do not need to think about the setup of the shared dictionary network, as the `SharedDictionary` object they use to access it takes care of all the details. This keeps end-programmers focused on the structure of the data they wish to share, not the mechanics of sharing it.

Internally, this object uses a client/server architecture for a centrally-coordinated data store. Clients send updates to the server, which orders them and forwards them to other clients. Data is cached at each client for rapid access.

To the end-programmer however this object looks like a hash table that maps hierarchically structured keys—text strings resembling paths in a conventional disk file system—to values. The object manages the connection to the server transparently, automatically marshalling data sent to the server.

The shared dictionary automatically deals with *late-comers* by providing a client with a completely up-to-date version of the data store at the time it connects to the server, similar to [11]. The `Opened` event is raised on the client after it has connected and fully updated its local cache. In the figure, the handler for this event stores a "display name" for the current client which used as a window caption on other clients.

When the connection is closed or broken due to a network connectivity problem, the end-programmer can handle the `Closed` event and set a flag to have the connection automatically re-established. The code in the figure uses the `Troubleshoot` method to notify the end-user of connection troubles and ask for permission to reconnect.

When a client connects to the shared dictionary server, the server informs the other clients already connected to it, and they in turn each raise the `Entered` event. In this simple example, a separate window is created to render the video from each client. This window will be deleted in the `Exited` event handler when the corresponding client disconnects.

B. *Organizing & Storing Data in a Hierarchical Dictionary*

Values that may be stored in the shared dictionary may be of practically any type. The Collabrary automatically marshals the data i.e., convert it into byte array that can be transmitted over a network. This makes the shared dictionary:

- *accessible*, because novice programmers need not concern themselves with marshalling;
- *lightweight*, because expert programmers need not write any code to take care of marshalling; and,
- *flexible*, because data are shared in their normal types.

A value is stored using a simple assignment syntax e.g., `sd["/user/name"]="Mike"`. The value is removed by overwriting it with `null`. This is:

- *accessible*, because it is the same syntax as that which is used with the system-supplied hash table class;
- *lightweight*, because assignment is one of the simplest programming statements; and,
- *flexible*, because the end-programmer decides the names of keys and the values stored at each.

The shared dictionary supports hierarchical organization of data because keys look like paths in a disk filesystem. In the figure, the `SharedDictionary.Me` property retrieves the current connection's id and prefixes it to the `"/video"` substring to generate the complete key used to transmit compressed video frames.

C. *Subscription Notifications & the MVC Architecture*

The Collabrary shared dictionary has a mechanism whereby the end-programmer can request notification of changes made to the dictionary. The end-programmer obtains a `Subscription` object, specifying a key or pattern of keys to watch, and handles the `Notified` event on it. The simple pattern matching language available resembles the "filename globbing" pattern matching language used in UNIX and related disk file systems. (The code in the figure does not need to make use of pattern-based subscriptions.)

Video is streamed by repeatedly storing individual video frames at the same key in the shared dictionary. The server broadcasts the updates to all connected clients. As each update is received, the key is inspected and the `Notified` event handler for any matching subscription is invoked with parameters that describe the change. In the figure, a separate subscription is used to decompress the compressed video from each client and render it into its own GUI window.

The ability to organize data hierarchically and receive asynchronous notification of data changes allows the end-programmer to employ the shared dictionary as the "model" within a Model-View-Controller or Presentation-Abstract-Control architecture pattern [4]. These models are important because they allow the end-programmer to separate the abstract data model from how it is gathered (i.e., the input gathered by the controller) and how that data is displayed (via the view or presentation). This separation is critical in a distributed environment where different clients may have different views or different means of managing user input.

D. Controlling Presence Distribution of Keys & Values

The Collabrary shared dictionary includes features to control how long keys or values stay in the shared dictionary. Normally, when a client puts a value in the shared dictionary, it is sent to all clients and it is stored in the dictionary indefinitely. It can be overwritten (by any client, not just the one that first put it there) by assigning a new value to the same key. The entry will be removed when a client sets the key's value to `null`. A client receives a copy of all data on the server and does not need to obtain a subscription for it or otherwise express interest in it. However, the shared dictionary server may silently drop an unsent and unneeded update when the link to a particular client is slow or congested.

The default persistence and distribution behavior is good for most purposes, but may be changed to make the prototype more robust in lower-bandwidth network conditions. Several options are available to:

- control data caching;
- receive only updates for keys it has subscribed to;
- ensure every update (even redundant ones) are received;
- send high priority data preemptively;
- specify which other clients receive the data;
- indicate how long data stays in the cache; and,
- tie the presence of keys in the cache to the connection status of a particular client.

For example, Fig. 4's `Opened` event handler stores a flag in the dictionary that binds persistence of the subtree used to store a client's data to the connected status of the client. The server automatically removes the subtree when the client disconnects.

VII. Discussion And Conclusions

We believe that the Collabrary is a significant contribution to rapidly prototyping multimedia groupware because it *trivializes* four common programming tasks for multimedia groupware: capturing, manipulating, transmitting and rendering multimedia. This was done in three ways. First, we illustrated how the Collabrary is *accessible* because it allows end-programmers to use programming idioms that are already familiar to them. Second, we have shown how the Collabrary is *lightweight* where it makes "easy things easy" in a number of ways. Third, we explained how the Collabrary is *flexible*, where it makes "hard things possible".

While space does not allow us to elaborate, the above design features have been validated in practice. The Collabrary has seen active use for several years by a variety of researchers. It is the architecture underneath several long-running and heavily used media space prototypes, e.g., the Notification Collage [1], Community Bar [22], and Home Media Space [23]. It is the basis of several quite novel systems, such as *mixed presence groupware* [24] and user interfaces for generating custom notifications [25]. It was used to teach undergraduates groupware programming, where students designed and quickly implemented many intriguing systems [6] in a very short amount of time.

However, we recognize that some will see the Collabrary as just another toolkit. Perhaps the more long-lasting contribution is our design requirements: we believe any universally accepted prototyping toolkit for distributed multimedia groupware research must trivialize four common programming tasks – capturing, manipulating, transmitting and rendering – by being accessible, lightweight, and flexible. The Collabrary merely shows one way that this can be accomplished.

Try it yourself. The Collabrary may be downloaded from http://grouplab.cpsc.ucalgary.ca/collabrary.

References

[1] S. Greenberg & M. Rounding, "The Notification Collage: Posting information to public and personal displays", in *Proc. ACM CHI*, 515-521, 2001.

[2] A. Fass, J. Forlizzi, & R. Pausch, "MessyDesk and MessyBoard: Two designs inspired by the goal of improving human memory" in *Proc. DIS*, 303-311, 2002.

[3] G.Fitzpatrick, *The Locales Framework: Understanding and Designing for Wicked Problems,* Kluwer Academic Publishers, 2003.

[4] *Human Computer Interaction, Second Edition,* Dix, Alan, Finlay, Abowd, & Beale eds., Prentice Hall International, 1998.

[5] W.A.S. Buxton "Living in augmented reality: Ubiquitous media and reactive environments", in *Video Mediated Communication*, Finn, Sellen, & Wilbur eds., Lawrence Erlbaum Associates, 363-384, 1997.

[6] S. Greenberg, "Toolkits and interface creativity", *Multimedia Tools and Applications*, Kluwer Academic Publishers, in press.

[7] *RTP: A Transport Protocol for Real-Time Applications*, IETF RFC 3550, 2003.

[8] *SIP: Session Initiation Protocol*, IETF RFC 3261, 2002.

[9] *Web Services Architecture*, W3C Working Group Note 11, Feb. 2004.

[10] G. Fitzpatrick, S. Kaplan, T. Mansfield, A. David, & B. Segall "Supporting public availability and accessibility with Elvin: Experiences and reflections", in *J CSCW*, 11:3, 447-474, 2002.

[11] R. Burridge, *Shared Data Toolkit for Java Technology User Guide*, Sun Microsystems JavaSoft, 2004.

[12] S. McCanne, E. Brewer, et. al, "Toward a common infrastructure for multimedia-Networking Middleware, in *Proc. NOSSDAV'97*, 1997.

[13] N. Roussel, "Exploring new uses of video with videoSpace", in *Proc. EHCI*, LNCS 2254, 73-90, Springer, 2001.

[14] M. Roseman & S. Greenberg, "Building real time groupware with GroupKit", in *ACM TOCHI*, 3:1, 66-106, 1996.

[15] *Microsoft NetMeeting, 3.01 SDK*, Microsoft Corporation, 2003.

[16] *Microsoft DirectShow 9.0 SDK*, Microsoft Corporation, 2005.

[17] *Java™ Media Framework API Guide*, Sun Microsystems, 1999.

[18] G.R. Bradski, "Computer video face tracking for use in a perceptual user interface", in *Intel Technology Journal Q2'98*, 1998.

[19] C. Gutwin & R. Penner, "Improving interpretation of remote gestures with telepointer traces", in *Proc. ACM CSCW 2002*, 49-57, 2002.

[20] *Video coding for low bit rate communication*, ITU-T H.263, 2005.

[21] N. Carriero & D. Gelernter, "Linda in context", in *Comm. ACM*, 32:4, 444-458, 1989.

[22] G. McEwan & S. Greenberg, "Community Bar: Designing for awareness and interaction", in *ACM CHI 2005 Workshop on Awareness systems: Known Results, Theory, Concepts and Future Challenges*. 2005.

[23] C. Neustaedter & S. Greenberg, "The design of a context-aware home media space", in *Proc. UBICOMP*, Springer-Verlag, 297-314, 2003.

[24] A. Tang, M. Boyle, & S. Greenberg, "Display and presence disparity in mixed presence groupware", in *JRPIT*, 37:2, 71-88, May 2005.

[25] S. Greenberg, & M. Boyle, "Tracking visual differences for generation and playback of user-customized notifications" (in submission).