Boyle, M. and Greenberg, S. (2005)
Rapidly Prototyping Multimedia Groupware. Proceedings of the 11th Int'l Conference on Distributed Multimedia Systems (DMS'05, Sep 5-7, Banff) Knowledge Systems Institute, IL, USA

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 designed to ease 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 simple ideas are simple 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. Systems such as Instant Messaging now add pictures, voice and video to what was once a simple text channel. Scores of experimental *multimedia groupware systems* supporting distributed colleagues treat multimedia as first-order data types [1,2] blending streaming multimedia with persistent shared application state.

Yet multimedia groupware design is challenging. It must cater to complex human and social factors to support both individual and team work practices [3]. 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.

 Funded 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).

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 [5].

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." Yet it is hard to develop a toolkit for building distributed multimedia systems because they require a wide gamut of hardware and software infrastructure.

On the multimedia side, there must be OS support for A/V hardware, and libraries for capturing, compressing, and rendering multimedia. On the network side, multimedia must be distributed to all machines participating in the groupware session. This 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]).

While some distributed multimedia toolkits provide robust and high-performance streaming multimedia services (e.g., [12,13]), they omit rich support for sharing other sorts of application data needed in groupware. Similarly, groupware toolkits that support application data sharing (e.g., [14]) do not robustly handle multimedia. Using these two classes of toolkits together simultaneously is awkward, as they use mutually incompatible programming environments and idioms.

Balancing abstraction and performance is tricky. A balance

that is appropriate for prototyping will like not be appropriate for production, and vice versa. For example self-contained applications like Microsoft NetMeeting [15] are easy to use and offer good performance but can be remotely controlled in only limited ways. Commercial toolkits like Microsoft DirectShow [16] and JMF Java Media Framework [17] are more flexible but are also incredibly complex to learn to use.

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 popular rapid application development platforms (e.g., Visual Basic, C#, Python) as well as lower-level languages like C++. In the sections that follow, we explain the requirements for this toolkit and illustrate how the Collabrary meets these requirements.

While the prototypes produced with the Collabrary are optimized and robust enough for limited deployments that suffice for understanding complex socio-technical factors, they are not as optimized or robust as systems made with production-oriented toolkits. The benefit of using the Collabrary is that prototypes made with it are highly malleable and quickly implemented, matching the needs of prototyping.

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 on implementing the novel aspects of the design, and make substantial, deep revisions to the 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 easy to learn, so that 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 need very little code to implement, using 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, the Collabrary provides direct access to multimedia data so they can be altered, uses programming idioms borrowed from other application domains already proven flexible, and 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.

- *Capturing* multimedia must be trivial.

```
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`.

- *Rendering* multimedia must be trivial and compatible with a rapid application development GUI toolkit.
- Simple multimedia *manipulation & analysis* must be trivial, while advanced manipulations must be possible.
- *Transmitting* multimedia and application state data must be trivial and match how end-programmers work with the data.

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 routine multimedia toolkit program task is audio/video capture. Yet, even this can be difficult as it involves discovering capture hardware, accessing it, 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). Some principles are analogues of patterns employed in other prototyping toolkits while others emerged during the process of employing the Collabrary in various prototypes [1,22].

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.

These features are not implemented in other toolkits.

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. 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 find familiar as it uses the same event dispatch 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. Stream-oriented pipeline architectures (e.g., [17]) automatically timestamp data in all streams using a common reference clock to ensure audio and video streams are tightly synchronized. While Collabrary end-programmers could implement timestamps themselves, it is not a feature of the event-oriented architecture.

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. 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#). Other multimedia toolkits typically render video into a widget it provides. Often, these private widget implementations are tightly coupled to the rest of the multimedia architecture for performance reasons. Yet, the standard GUI widgets compatible with the Collabrary offer adequate performance, and using them affords three critical advantages.

First, it keeps the toolkit accessible. End-programmers do not need to learn how to use a new widget. Private widget implementations often provide APIs that are 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 interaction with the video display via the mouse/keyboard can be implemented using conventional UI programming patterns and practices. Private widget implementations are often incomplete, and do not expose mouse or keyboard input event bindings that support new forms of user interaction.

Third, it keeps the toolkit *interoperable*. The widget used to render video with the Collabrary 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 using the visual interface designers already familiar to the end-programmer. Private rendering widgets are often implemented as a popup window that cannot be visually integrated with the rest of the application's GUI or 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 basic manipulations are 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 built-in analysis algorithms. For example, Bradski's CAMSHIFT face-tracking algorithm [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

```
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.

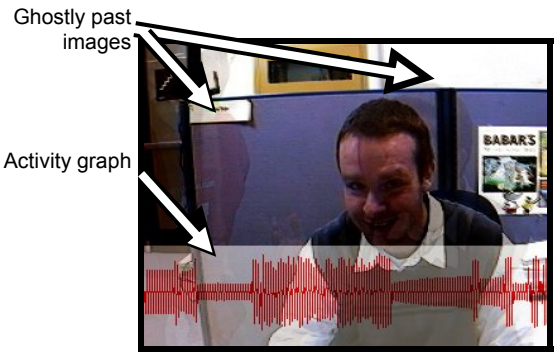

Fig. 3. Visual and graphical traces of activity, implemented by composing various pre-packaged Collabrary manipulations.

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 delta image 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 (not demonstrated due to space constraints). 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 passed to high-performance implementations of very sophisticated analysis and manipulation algorithms. This allows end-programmers the opportunity to quickly prototype or reuse a broad spectrum of effects and measures.

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. With the Collabrary, the end-programmer adds manipulation code to an existing capture event handler at a minimum cost of one additional line of code. Conversely, with a stream-oriented architecture as in the JMF the end-programmer must write a filter class and insert an instance of it into the pipeline at an appropriate place. The problem is filters are difficult to write. They implement extremely generalized interfaces which treat audio and video as generic byte arrays rather than rich image or audio types. Ultimately, the programmer is forced to implement mundane code—well in excess of just one line—that is irrelevant to the real work of the filter. This heavy "up-front" work does not match the needs of rapid prototyping.

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.

The Collabrary does not implement popular Internet protocols like SIP and RTP because some of the programming idioms used with them are not trivial enough for rapid prototyping. Simpler-to-program, but less robust and efficient protocols are provided, instead.

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.

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 is 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 display 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 is removed only when a client sets it'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 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 is illustrated 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* and makes "simple things simple" in a several ways. Third, we explained how the Collabrary is *flexible*, where it makes "complex 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" Report 2005-777-08, Dept. of Computer Science, University of Calgary, Canada, April, 2005.