GroupLab Collabrary: A Toolkit for Multimedia Groupware

Michael Boyle and Saul Greenberg
Dept. of Computer Science, University of Calgary
Calgary, Alberta, CANDA T2B 1X5
+1 403 210 9499
{boylem, saul}@cpsc.ucalgary.ca

ABSTRACT
GroupLab Collabrary is a toolkit for rapidly prototyping multimedia groupware. It provides a straightforward API to managing and distributing multimedia information between groupware program instances. The Collabrary also has a culture of use, where a small generic set of useful groupware programming patterns has evolved over time. In this workshop I will describe the Collabrary, what it offers groupware developers, and how these features in turn drive requirements for network services for rich groupware.

Keywords: Groupware, multimedia, networks services, notification servers, prototyping toolkits.

INTRODUCTION
The GroupLab team at the University of Calgary has made several major technical contributions to the CSCW community, particularly in our design and implementation of architectures and toolkits for rapidly prototyping groupware [e.g., 4]. Because we deal primarily with distributed groupware, these systems had to concern themselves with network issues, how data is distributed across the network, and how processes share data. Yet our true research interests lie not with technical issues about the design and implementation of network services for groupware. Rather, our main goal has always been to give programmers the power to rapidly prototype groupware applications, to use these prototypes to examine user experiences and to uncover the human and social factors associated with groupware tool use.

Yet we ended up devoting a great deal of our time experimenting, building and packaging various network services for groupware. This is because the services we then had at hand—mostly bare-bones TCP sockets—were too low level to help us rapidly prototyping the kinds groupware applications we were interested in building. This was the rationale for our GroupKit toolkit for building interactive graphical groupware applications, where we introduced the notion of *session managers* and *environments* as a way to manage and hide network details while providing a simple way for programmers to distributed string-based data [4].

Cite as:
Boyle, M. and Greenberg, S. (2002) GroupLab Collabrary: A Toolkit for Multimedia Groupware. In J. Patterson (Ed.) *ACM CSCW 2002 Workshop on Network Services for Groupware*, November.

While GroupKit proved extremely effective for prototyping 'conventional' groupware applications, we found it wanting when we moved into building media spaces. Specifically, the network services behind GroupKit were too limited to support the demands of rich multimedia groupware applications e.g., images and audio/video channels. Consequently, we set ourselves the new research goal of building an infrastructure for managing multimedia groupware communication. The idea was to leverage all the experiences and lessons learned from our previous toolkits and applications (as well as those reported by others) into a new toolkit that we felt could meet our desire to rapidly prototype multimedia groupware.

The result is the *GroupLab Collabrary*, a toolkit we developed for rapidly constructing novel multimedia groupware applications. Our Collabrary is a proven toolkit. We have used it to build serious applications under daily heavy use e.g., the Notification Collage media space that provides informal awareness and casual interactions for a distributed community [3]. We have also used it to implement many high-fidelity research prototypes, including a privacy-preserving video media space application [1] and the IMVis [4] instant messenger visualization. The Collabrary also has use outside our research group, where undergraduates built quite interesting groupware applications as part of their course project.

In this workshop, we will highlight those features of the Collabrary that make it a powerful tool for building rich, multimedia groupware. In particular, we look at its high level concepts – its architecture, its client-side API and the suite of groupware development patterns and practices that evolved over its use. We caution that we do not specifically address low level network services, for we believe that the strengths of the Collabrary are somewhat agnostic towards the network service upon which it is implemented. While we actually implement a simple TCP-based binary prototocol for client/server communications, this plumbing could conceivably be swapped out for, say, an XML web service layer that affords features not easily provided by the existing implementation. Consequently, what the Collabrary provides is valuable insight behind the requirements of network services if this service is to support its API and programming patterns.

COLLABRARY FEATURES

The Collabrary is implemented as a library of COM classes for the Microsoft Windows platform. The goal of the Collabrary is to ease many of the mundane yet tricky aspects of media space programming, namely capturing multimedia in a way that it can be processed algorithmically and then distributing it to other media space participants via the Internet. The Collabrary has two major parts. The first part included components that ease multimedia capture, display and manipulation e.g., video, audio, images. The second part, and the focus of this paper, is a *hierarchical shared dictionary* component that eases how multimedia data is captured in a data structure and shared between distributed groupware processes. It is this shared dictionary that encapsulates the network service layer.

Shared dictionaries

A dictionary is a collection of <*key*,*value*> pairs. A shared dictionary is a dictionary shared between distributed processes. This provides a simple mechanism for data to be shared among instances of a groupware application. Shared dictionaries are not new to groupware. Groupkit, the Collabrary and several toolkits built by others implement this basic idea. Typically, changes made to a key/value pair in the dictionary are propagated via the network to other processes (usually residing on different machines). This generates notifications or events, so that the programmer can catch selected changes and handle them asynchronously.

The strategy of using synchronous notification of changes to the dictionary has generated two programming patterns. First, the programmer just uses the shared dictionary to act on data events as they arrive i.e., as a pure notification service [2]. Alternatively, the programmer can use the dictionary as an event-producing model following the Model-View-Controller style. Our own experiences with GroupKit and other's experiences with similar groupware toolkits illustrate that providing for this clean separation of the shared model from the view/controller is an important pattern for successful groupware programming.

Rich values and marshalling

Unlike most other implementations of shared dictionaries, the Collabrary has the ability to marshal data as network-representable types, where the dictionary can persistently store this data into the key/value pairs. This includes primitive types such as numbers or strings, aggregate types including arrays and records, and complex types such as objects that may be marshaled by value. These complex types include multimedia such as audio or JPEG image data. For example, Figure 1 illustrates a fragment of a shared dictionary modeling a video and text-based multimedia chat program. Values are shown at the right of the := signs. We see string values representing a user's name ('Michael Boyle'), hexadecimal values representing a drawing color (#ffee00), and a JPEG object representing a single frame of video (<JPEG image data>).

Hierarchical keys

Collabrary shared dictionary keys are hierarchical, in that the string comprising a key's name looks much like file system path e.g., /a/b/c. The Collabrary also supplies the programmer with a simple pattern matching language that works over these keys, so that a sub-tree containing many <*key*,*value*> pairs can be treated as a single logical construct.

These hierarchical key names are a very powerful programming construct. With them, the groupware application developer can arrange and manipulate associated items in the dictionary as a loose hierarchy. This allows data in the dictionary to be compartmentalized, or to be aggregated into larger logical units.

To illustrate, the keys in Figure 1 are structured as followed. We see two root keys: /users stores information about users, and /chat stores information associated with each chat message. We also see that /users has two immediate children that identify two different people. Here, the 2nd term in each users path comprises a system-generated unique ID: /users/{a1b2c3..d4} and /users/{1d4234..77}. Similarly, there are two chat messages: /chat/#1 and /chat/#2. The third levels of the hierarchy define content: the name and video image of that particular person, and the author, text contents, and font/color description for the particular chat message.

Programmers can then specify patterns to act over these hierarchical keys. For example, one iterates a list of the names if all users by simply writing

```
ForEach n in /users/*/name
    print n
```

There are almost no restrictions on key name formats,

```
/users  ←stores per-user information
    /{a1b2c3..d4}  ←info about a particular chat user
        /.transient  := '{a1b2c3..d4}' ←metadata
        /name        := 'Michael Boyle' ←friendly name
        /video       := <JPEG image data> ←live video snapshot
    /{1d4234..77}  ←info about another user
        /.transient  := '{1d4234..77}'
        /name        := 'Saul Greenberg'
        /video       := <JPEG image data>

/chat  ←stores all chat messages
    /#1  ←info about one chat message
        /author      := 'Michael'
        /text        := 'Hey, Saul, how's the
                         weather in the mountains?'
        /font        := 'helvetica'
        /color       := #ffee00
    /#2
        /author      := 'Saul'
        /text        := 'It's sunny now, but the
                          forecast is showers.'
        /font        := 'times'
        /color       := #003bff
```

Figure 1 Shared dictionary hierarchy for a chat application.

making it possible for the developer to use meaningful nomenclature, and to arrange the items in a logical fashion that models the desired groupware application. That is, the groupware developer can readily apply familiar programming patterns borrowed from file system and object-oriented programming, e.g., using abstractions and factoring to reduce complexity and coupling.

Subscriptions

As previously stated, the Collabrary uses an asynchronous event-driven programming model for discovering changes made to the shared dictionary. Programmers manage these events by defining *subscriptions* to particular key patterns, and by attaching callbacks to these subscriptions. Pattern matching leverages hierarchical keys. For example, a programmer can write a single callback that manages changes to chat text by subscribing to /chat/*/text. When the callback is fired, the programmer can easily query the key that triggered the callback to find out the parent of this key (e.g., /chat/#2), and then use that information to access all other information related to that message (e.g., the child keys identifying the author, font and color).

It should be readily apparent that our approach to managing subscriptions via events and callbacks reflects the familiar design pattern of how programmers now manage GUI widget events. This makes it possible for the shared dictionary to be used conveniently within a GUI application framework.

Metadata

We exploit the hierarchical nature of our shared dictionary by associating and storing metadata side-by-side with regular data in a sub-tree of the dictionary. Using this metadata, the Collabrary architecture can then make decisions and take action on the programmer's behalf. For example, the programmer can attach a '.transient' key that associates a sub-tree with a particular conference participant (see Figure 1). When that participant disconnects from the server (either gracefully or by abortive closure), the server will automatically search for that .transient sub-key, and then remove the sub-tree associated with that key. For example, in the chat application in Figure 1, the sub-tree assigned to each user stores his/her friendly name, e-mail address, and availability status. When that user's client disconnects from the chat session, all of the information for that user is automatically removed from the dictionary because it is marked with .transient. This greatly simplifies housekeeping.

Because metadata in the Collabrary is just a standard key name that follows a particular convention (the Collabrary prefixes it with a '.'), clients can easily specify metadata through the standard Collabrary API i.e., no additional APIs are needed to manage metadata. It also means we can easily grow the metadata vocabulary in an incremental and fully backward-compatible fashion, e.g., future versions of the shared dictionary server may use metadata to implement access control lists (ACLs) or specify QoS parameters.

Signals

The shared dictionary normally stores all its data in a server (this data is also cached on clients for efficiency). While useful for implementing the model-view-controller pattern (especially for updating newcomers to the groupware session), there are times when the programmer does not need to store data once all clients receive a copy of it. Perhaps the best example of this is data that do not need guaranteed delivery or storage because another copy of it will be sent along shortly. Examples are motion video frames or mouse telepointer coordinates. A better pattern is to use the dictionary's capability to stream data to one or more participants by repeatedly 'signalling' the same key with a different value (e.g., compressed video frame) each time. The value is discarded after signaling is complete.

We can then apply simple QoS parameters to signals. such timeouts and priorities. Thus, these 'signals' are analogous to notifications found in pure notification servers [2]. Indeed, because the Collabrary shared dictionary architecturally uses a client-server network topology, it could easily be implemented atop an existing pure notification network service by augmenting it with a data-storing service. However, it is important to point out that the practical benefits of the Collabrary (e.g., automatic data persistence) are not provided by pure notification servers.

WHAT YOU CAN'T DO WITH THE COLLABRARY

The Collabrary shared dictionary is a research system, and we did not intend it for large scale commercially deployed groupware. Consequently, while fine for our research projects, its current implementation has several noticeable omissions. It does not scale well beyond a few dozen clients. It provides no support for authentication or authorization (access control). It does not implement wire privacy (encryption) and is, as constructed, not particularly platform independent. On a much higher level, the Collabrary does not offer QoS feedback, which is important for some kinds of interactive multimedia communications applications.

While addressing these omissions will likely require architectural and other changes to the implementation and underlying network services, the Collabrary features and programming patterns described above remain valid. Again, we reiterate: the value of the Collabrary is not in the design of its internal network service layer, but rather the demands upon such a layer it places.

CONCLUSION

The Collabrary contributes a hierarchical shared dictionary programming paradigm. It gives the programmer a great deal of power through its pattern-based subscriptions and asynchronous notifications. Its data model encourages the model-view-controller programming pattern now found in good groupware application design. The Collabrary offers

server-side persistent data, and also transient data with signaling for managing streaming multimedia. Both can be intermixed, and both are critical for prototyping rich, multimedia groupware.

In our experiences, programmers find the Collabrary an easy yet expressive infrastructure for building multimedia groupware. Programmers combine a few well-understood patterns for groupware development with its very simple yet generic API. This gives them the opportunity to rapidly build groupware prototypes, where they remain focused on the group interaction it affords, instead of on the mundane aspects of maintaining network connections and passing data through them.

Taken together, the Collabrary shared dictionary embodies several key concepts that could influence future network services for groupware programming.

REFERENCES

1. Boyle, M., Edwards, C. and Greenberg, S. (2000). The Effects of Filtered Video on Awareness and Privacy. *Proceedings of the CSCW'00 Conference on Computer Supported Cooperative Work* [CHI Letters 2(3)], p1-10, ACM Press
2. Fitzpatrick, G., Mansfield, T., Kaplan, S. Arnold, D., Phelps, T. and Segall, B. (1999) Augmenting the Workaday World with Elvin. *Proc 6th European Conf Computer-Supported Cooperative Work* (ECSCW'99), p431-435.
3. Greenberg, S. and Rounding, M. (2001) The Notification Collage: Posting Information to Public and Personal Displays. *Proceedings of the ACM Conference on Human Factors in Computing Systems* [CHI Letters 3(1)], 515-521, ACM Press.
4. Neustaedter, C., Greenberg, S. and Carpendale, S. (2002) IMVis: Instant Messenger Visualization**.** *Video Proceedings of the ACM Conference on Computer Supported Cooperative Work.*
5. Roseman, M. and Greenberg, S. (1996). Building Real Time Groupware with GroupKit, A Groupware Toolkit. March. *ACM Transactions on Computer Human Interaction*, 3(1), p66-106, ACM Press.