Introducing Collaboration to Single User Applications

Brian Cornell
UNC Chapel Hill
brian@cs.unc.edu

Abstract

Most research in collaborative systems has been based on designing an ideal system from ground up. Most applications are designed without collaboration in mind however, so we need to consider how we can apply these systems to existing single user applications. I present a summary of some recent work on introducing collaboration to single user applications, and an analysis and comparison of their usefulness.

1. Introduction

Introducing collaboration to single user applications is an important concern in the field of computer supported cooperative work (CSCW). Much work has been done in the past 30 or so years in researching how computers can help us work collaboratively. Despite all of this work, most off-the-shelf applications are still single user. Our image editors, word processors, CAD tools, etc are all designed for use by one person at a time.

The biggest issue with the single user nature of current applications is the entry barrier for making them collaborative. These applications have grown so large that it is impractical to change their base code to make them collaborative. Add to this that not all developers are even convinced that computer based collaboration is useful. So while some vendors may avoid collaboration due to difficulty, other vendors may not even consider it.

Yet we have a large community of users who would like to use collaborative applications if available. We also have a community of researchers and developers who feel that collaboration is important. What all of this means is that as developers who want to see collaboration in these applications, it is up to us to provide the collaborative features that the vendors cannot.

In the next section I will give an overview of the retrofitting process, and then provide a motivating example in section 3. Sections 4 and 5 will describe techniques to apply collaboration to single user application. Section 6 will then provide comparison and then section 7 provides a short conclusion.

2. Retrofitting

If we want to provide collaborative capabilities in existing single user applications, we must retrofit them rather than designing from ground up. By retrofitting I mean that we will not change the basic behavior or architecture of the application. The application will remain essentially the same with the exception of the collaborative capabilities, and will not lose or

change any other functionality. In contrast, an application may be designed for collaboration with a different set of features and different architecture, but that it not our goal here.

There are three main layers at which we can retrofit an application. The application layer is where the actual code and implementation of the application is. At this layer we can create a branch of the application code itself and have a completely different collaborative version. The operating system layer is where the windowing system and IO systems reside. At this level we can replicate graphical data and input streams as most screen sharing systems do. The last layer, the programming environment layer, resides between the application and operating system layers. At this level we can use an API or plugin system of the application, or exploit the capabilities of its runtime environment to insert collaboration.

3. Motivating Example

Before we talk in more detail about all of these options, let us motivate this with an example application. We will assume that the application was designed with no intent to ever be used collaboratively. We will also assume however that it meets any prerequisites for the methods I will describe in this paper.

Imagine a text editor designed for programming. It has many advanced editing features such as syntax highlighting, smart indenting, folding, etc. All of the documents it supports are plain text though. We would like to use this editor collaboratively for concurrent development of projects by small teams (less than 10 people). We assume any necessary infrastructure such as versioning systems, shared file systems, email, and instant messaging systems.

Ideally there is a large list of features we would like to see in this editor. First of all, we want multiple people to simultaneously view and edit the text. We would also like to be able to chat with the other people, and see their status. If they are currently editing, we would like to be able to see what they are doing. We would like there to be minimal lag between when we do something and when we see the result. If there are editing conflicts, we would like some sort of merging scheme to resolve them gracefully. Finally, people are likely familiar with the interface of the editor, so we don't want to change it much.

4. OS and Application Layers

The operating system layer has been used by many people for many years to use applications collaboratively. The most common technique at this level is screen sharing[3, 5]. In screen sharing, an application retrieves information about the display of other applications and sends that over the network. Input data is sometimes collected from the remote clients and sent through to the application.

Screen sharing has a few problems despite its wide use. While it allows multiple people to view the application, only one can edit at a time and so there is an input bottleneck. There can also be considerable latency since there is a round-trip over the network involved between input and related display. Screen sharing is an attempt at allowing other people to be there across distance, but provides nothing beyond that.

There are some great benefits of screen sharing though, because the protocols often run on almost any platform, and they can support any application with no modification. For our motivating example we get simultaneous viewing with the same UI, and we can have asynchronous editing by multiple people, but we don't get concurrent editing, chat, status,

merging, or rich awareness data, and there is usually considerable lag.

We also cannot forget the application layer, which would involve reprogramming our application. We could implement the features we want directly in the editor if we have the source code, but we might not have that source. Also the editor might be incredibly complex, something like Vim[4] or Emacs[6], that might make it too difficult to modify. We also have to implement every detail ourselves, and it will likely not share a collaboration interface or infrastructure with any other application.

5. Programming Environment Layer

Given the major pitfalls with the other two layers, we are driven to concentrate on the programming environment level. We further divide this layer into three slices. Aspect oriented programming will try to insert code into an application using aspects[1]. Plugins will allow us to add code through provided interfaces[2]. Transparent adaptation will allow us to share a document or model between copies of an application[7].

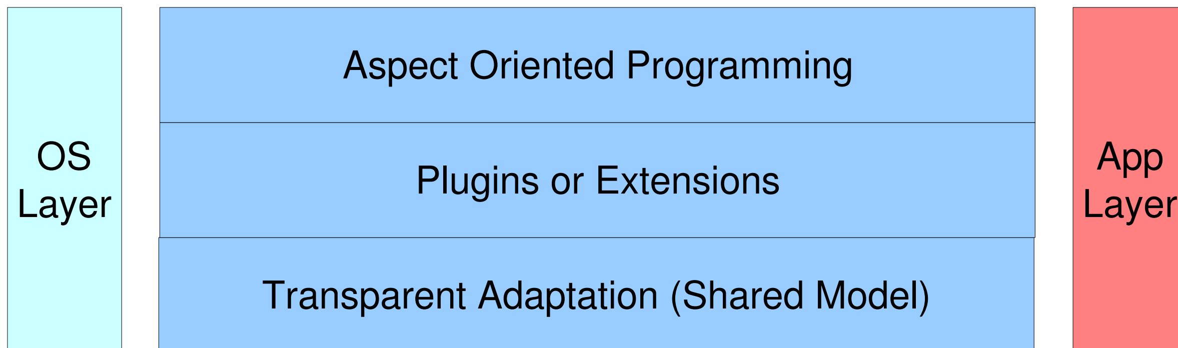

Figure 1: Retrofitting Layers

5.1. Aspect Oriented

For the aspect oriented approach, we will use the relatively new concept of aspects to hook into related function calls in the original application. From the code of the aspect we will influence the UI of the application to be collaborative. When applied using a dynamic runtime environment such as the Java virtual machine, aspects can modify running applications without access to the source code.

Aspects consist of a rule and advice. The rule describes where the aspect compiler should insert this code, and the advice is the code that will be executed. The rule consists of a location keyword, a comma separated list of arguments in parentheses, and a matching condition after a colon. The location keyword may be something such as after or before, telling the aspect compiler whether the advice should be executed before or after a block that matches this rule. The arguments are like function arguments and will be available in the local scope of the aspect advice. The condition then must identify what code to look for, and must provide the contextual source for all arguments. There are many matching operations based on function names, arguments, return values, and objects they operate on. For complete details on aspects, refer to documentation for a specific aspect compiler.

So as an example in our editor, the aspect to the right will execute after each time the setText method is called on a TextLine object. The target keyword in the matching condition specifies that the argument *line* should be filled with the object that this method is called on. We will assume that the TextLine object represents a line of the document in our editor. When we intercept this call, we will add an icon to the front of the line indicating who last changed that line using data from some versioning software. In this way we can now see who has been doing what in the code, and perhaps even lookup status information from an IM system.

```
after(TextLine line) :
 (target(line) && call(*
 TextLine.setText(..))) {
      // Add an icon to the
      // line showing who
      // last changed it
      // based on info
      // from external
      // versioning software
}
```

Example Editor Aspect

There are a few benefits to this approach. First, we don't need to recompile the application. Second, the rules allowed by aspects are very flexible. Fore example, we could change all lines as above, or we could change just the first line if we have a way to uniquely identify it. We also have some level of access to the original application's data and objects, so we can put in more contextual information. For our motivating example we could add a variety of awareness information, status, and perhaps even IM capabilities without changing the interface significantly.

There are also some problems with this approach though. If our application doesn't use a runtime environment that supports introspection or some way to apply aspects without the original source, we can't use this approach. Also the internal objects of the application that we want to modify or get data from could be quite complex making it difficult to get what we want. Furthermore, because of the nature of the aspects we can only provide new features as a result of function calls in the application, so we can't for example change the document based on actions taken at a remote site. For our programming editor this means that we don't get document synchronization of any sort, and it may even be difficult to implement any sort of chat interface.

5.2. Plugins

The second approach at the programming environment level is to use the plugin or extension architecture provided by the application to add collaborative components. For example, perhaps our editor has a plugin system that allows us to create additional panes in the editor window. The content of these panes is controlled by a function loaded dynamically from a shared object library for the plugin. Our editor may not however provide any way for these plugins to access the document or other parts of the application.

What plugins provide us is a way to add collaboration without access to the source code for applications written in any language and using any runtime environment. It is also likely that even if the application changes or a new version is released, our plugin may still work or require only a small update. Because the application is designed to be extended using these plugins there should already be proper support and it is less likely that our new components will cause problems with the rest of the application. In our editor we could easily add panes that provide awareness and status information about collaborators and allow us to chat with them in a UI that fits the editor.

The biggest problem with this plugin approach is that the application we are modifying must have support for plugins. Many applications don't include such support, or have limits in the

support that would make it difficult to do what we want. It is very common for plugins to only be able to add components or behavior that is triggered in a predefined way by the application. Plugin systems are often designed such that plugins can't change anything that the application already does. What this means for our editor is that we can't access the document, so we can't get any sort of synchronization or contextual awareness.

5.3. Transparent Adaptation

The third and final approach we will discuss for retrofitting at the programming environment level is what we call transparent adaptation. Basically this means that we will adapt things that happen in the application using some application API into a sequence of events. We will correlate these events to operations that perform on some linear document or model. We will then apply any consistency technique we'd like to synchronize these operations. Actions received from remote sites will be translated back into the application using the API.

We could use any consistency technique that we'd like to merge the operation streams, but one example is operational transform. In this technique we transform future operations based on the side effects of operations we apply. For example, let us assume that we have a string “abc” and two operations in our queue, one that inserts “z” as position 1, and one that deletes “c” at position 3. Inserting the character z at the beginning of the string will transform the operation to delete the third character, now b, into one that deletes the fourth character, now c.

For this transparent adaptation technique, we need to translate between the API and streams of linear operations. For this to work, the application must have some concept of a linear document or model, and the API must have access to it. We translate actions in the application into what are called Adapted Operations (AO)[7]. These operations are then sent to remote sites and used by the consistency technique. Operations we get from remote sites are transformed using the consistency technique and then applied to the local document using the API.

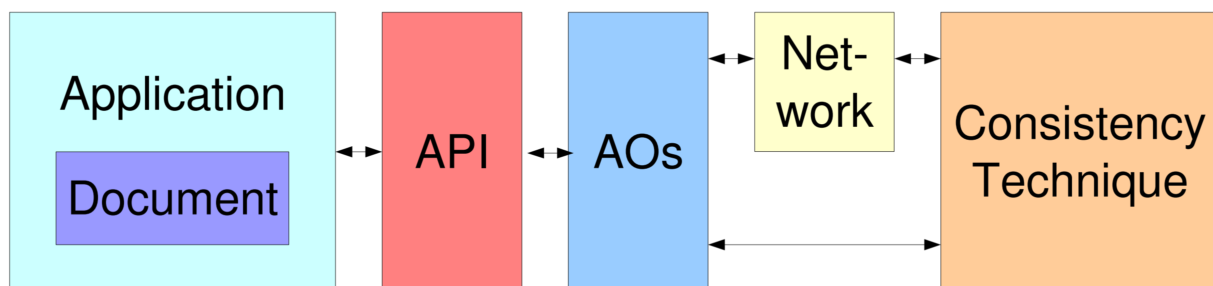

Figure 2: Transparent Adaptation Architecture

One of the main benefits of transparent adaptation is that the single user application is completely unaware of collaboration. Thus the interface remains exactly the same. We can do simple collaboration using this technique no matter how complex the editor is, as long as it can be linearized. We don't require the application to use any specific language or infrastructure as long as we can get access to an API. For our programming editor this means that we can get synchronous viewing and editing with no local latency (I see my own changes

immediately). We can apply any merging system we'd like, and the UI does not change at all.

There are a few problems with transparent adaptation however. First of all, it is only applicable to editor style applications where the document or model must somehow linearize. It may be argued that any application fits this requirement, but it is more natural for some than others. For example, a text editor has a naturally linear document, but a 3d modeler may not. The application must also have some sort of an API that is rich enough for us to get what we need. The per application adaptation to and from these adapted objects could get quite complex. There is also no way of using this method to change the application UI to add collaborative features. For our editor this means that we could not get any sort of awareness, chat, status, or the like, because these features require modifying the UI.

6. Comparison

When we look now at these three approaches, we can see that all of the features of our ideal collaborative editor are covered, just not all from the same technique. Perhaps we could combine the techniques in the same applications. These techniques are not mutually exclusive, and provide nearly disjoint sets of features. Aspects provide modification of existing behavior, especially in the UI, plugins provide additional but often separated behavior, and transparent adaptation provides document or model synchronization.

In our motivating example of a collaborative editor, we could combine these three techniques and get exactly what we want. We could use transparent adaptation to synchronize the document, plugins to add chat and collaborator panes, and aspects to insert contextual information gathered from the other two into the editor UI. All of these techniques have the advantage that they don't change the UI significantly, and don't disable or change existing features, but all also require a significant per-application implementation.

We can also look at the scale of these three approaches. Plugins and transparent adaptation are more applicable to large applications because larger applications are more likely to provide a plugin system or rich API. Aspects seem better suited for smaller applications because they require introspection and modification based on the internal structure of the application, which could get quite complex.

7. Conclusion

Unfortunately none of these approaches is completely general or easy to adapt to new applications. There isn't a perfect solution yet, so perhaps we need to further explore ways to combine or use the different retrofitting layers to find better solutions. However if developers are willing to put significant effort into retrofitting an application, these techniques will apply to many existing applications and provide some needed collaborative functionality.

As a final piece of advice for application developers, take a look at the requirements for these techniques. Even if you don't provide collaboration in your application, it would help other a lot if you would provide the hooks and support so that somebody can use a technique like these to add collaboration for you.

References

1. Cheng, Li-Te, S.L. Rohall, J. Patterson, S. Ross, and S. Hupfer. “Retrofitting collaboration into UIs with aspects.” *Proceedings of the 2004 ACM conference on Computer*

supported cooperative work 2004: 06-10
2. Hupfer, S., S. Ross, and J. Patterson. "Introducing collaboration into an application development environment." *Proceedings of the 2004 ACM conference on Computer supported cooperative work* 2004: 21-24
3. Microsoft Corp. "Remote Desktop Protocol" http://msdn.microsoft.com/library/en-us/termserv/termserv/remote_desktop_protocol.asp
4. Moolenaar, Bram. "VIM The Editor" http://www.vim.org
5. Richardson, T., Q. Stafford-Fraser, K.R. Wood, and A. Hopper. "Virtual Network Computing." *IEEE Internet Computing* 1998, Volume 2: 33-38
6. Stallman, R.M. "EMACS the extensible, customizable self-documenting display editor." *ACM SIGPLAN Notices* 1981, Volume 16: 147-156
7. Xia, S., D. Sun, C. Sun, D. Chen, and H. Shen. "Leveraging single-user applications for multi-user collaboration: the coword approach." *Proceedings of the 2004 ACM conference on Computer supported cooperative work* 2004: 162-171