Designing a Scalable Cross Platform Imposed Code Reuse Framework

Kenton McHenry, Rob Kooper, Luigi Marini, Peter Bajcsy
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign, IL
{kmchenry,kooper,lmarini,pbajcsy}@ncsa.illinois.edu

In order to construct a file format conversion service supporting as many formats as possible we have introduced the notion of imposed code reuse in our past work (McHenry et al. [2-4]). Traditional code reuse, when an option, can save a significant amount of energy and time with regards to new software development while at the same time adding robustness through the use of code that has been proven over time. Imposed code reuse comes into play when original source code is not available. Consider proprietary software where only a compiled binary version is available. Such software will often provide only a graphical user interface (GUI) allowing humans to utilize the software with a mouse, keyboard, and monitor. While a GUI is useful for human interaction it is not exactly useful for accessing functionality programmatically by software developers. Imposed code reuse attempts to bridge this access to functionality locked away in compiled software by wrapping it through various scripting languages in such a manner that it can be used in other software through an API like interface.

In McHenry et al. [2-4] a system called NCSA Polyglot was developed by utilizing Java and AutoHotKey[1] scripting to wrap GUI based applications on the Windows operating system. AutoHotKey, a GUI scripting language, takes advantage of the message passing system within Windows. Using tools such as Winspector Spy[2] and AutoIt3 Windows Spy[1], the messages passed to various windows can be monitored and recorded from sample interactions with an application. The created scripts can then run the needed application and replay the recorded messages as needed in order to interact it. NCSA Polyglot used a simple workflow referred to as an Input/Output-graph to store file format conversion information with formats as the vertices and applications capable of converting between a source/target format as the edges. Conversions were carried out by searching the graph for a shortest path between a desired source and target format and executing the required scripts for each edge along the path. Weights were later attached to the edges based on a comparison of the before and after files [4]. These weights attempted to capture the information loss that occurred during the conversion (as a function of both software and

[1] http://www.autohotkey.com
[2] http://www.windows-spy.com/

formats involved). With the weights filled in future conversions could then use a shortest weighted path to choose conversions with the least amount of information loss.

There are various technical issues that need to be addressed when imposing code reuse on closed 3rd party software. The first issue is that of *robustness*. Though we can record the messages sent to an application, the timing of those message has to be controlled. For example, consider detecting a mouse click and replaying it at a later time. In order to obtain the same outcome of the click we must make sure the appropriate window is open and that it is in the same state as when the mouse click was recorded. Simply pausing for some fixed amount of time is not desirable as timings may change depending on the load of the system. The second issue is that of *scalability*. Because of what imposed code reuse usually involves, taking over a desktop in order to control a GUI based application, only one task can be performed at a time. In order to improve performance of the service utilizing imposed code reuse it is thus beneficial to allow for parallelism via clouds of such machines (real and/or virtual). This second issue also brings up the question of *platform support*. In order to reuse software functionality as broadly as possible we should not limit ourselves to one operating system. In terms of NCSA Polyglot this means not restricting ourselves to wrapping software with only the AutoHotKey scripting language which is primarily designed for Windows.

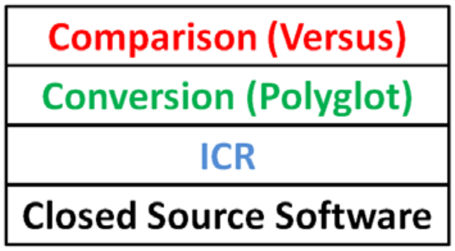

Figure 1. *The layered design of the second generation NCSA Polyglot conversion service. At the bottom we have the closed source 3rd party software we wish to utilize functionality from. On top of that we have the Imposed Code Reuse layer whose sole purpose is to provide a robust, consistent, and error tolerant Java API interface to closed source software functionality. On top of this we have the conversion layer which utilizes the "open" and "save" functionality of underlying software. This layer on top of the ICR layer pretty much makes up what is NCSA Polyglot in terms of a conversion service. At the very top we have NCSA Versus [1], an API/library of domain specific comparisons, which utilizes the underlying conversion layer to convert sample files, compare content before and after, and estimate information loss. By layering the Polyglot architecture we have simplified the implementation of each layer, allowed for more focused error checking (robustness), and made the overall system suitable for distributed executions supporting multiple OS platforms.*

In order to address the aforementioned issues, we begin by breaking the Polyglot service into multiple layers. Specifically we separate the multiple tasks within the original Polyglot service: imposed code reuse of closed source software, file format conversion, and the measuring of information loss (see Figure 1). Of particular importance is the imposed code reuse layer (ICR) which is focused specifically on providing a consistent Java API to underlying close source software functionality. Within its own layer it is responsible for executing wrapper scripts, detecting execution failures, and gracefully dealing with errors. We add robustness while at the same time addressing scalability and extensibility by distributing this layer among multiple machines (see Figure 2). On each machine an ICR server responds to requests from ICR clients, which in turn provide a consistent means of accessing the used software (see example below):

```
TaskList tasks = new TaskList(new ICRClient(server, port));
tasks.add(“Blender”, “convert”, “./heart.wrl”, “heart.stl”);
tasks.add(“A3DReviewer”, “open”, “heart.stl”, “”);
tasks.add(“A3DReviewer”, “export”, “”, “heart.stp”);
tasks.execute(“.”);
```

In this example an ICRClient is instantiated and a TaskList is used to organize tasks of the form “program”, “operation”, “input”, and “output”. All tasks take on this form whether or not inputs and outputs are needed. Currently we define errors as instances when a script does not respond after a given length of time. When this occurs the ICR server will attempt to deal with it though associated monitor and kill scripts. After retrying an operation some fixed number of times, rather than crashing the conversion service, the operation is aborted and the error can be passed up to the next layer above as an exception.

In order to add cross platform support we are removing our reliance on AutoHotKey scripts as software wrappers. The ICR server will utilize any text based script as long as it follows certain naming, comment, and argument guidelines. While we plan on still utilizing AutoHotKey scripts for Windows based systems we can use other languages such as Apple script for Mac systems, and one of a number of languages for Linux based systems. We have also begun constructing an alternative purely Java based scripting language that could be used across multiple platforms based on Java’s Robot class. This language which we refer to as ICR Monkey script after the saying “monkey see, monkey do” is being designed around the need for robustness, in particular the timing of reproduced events.

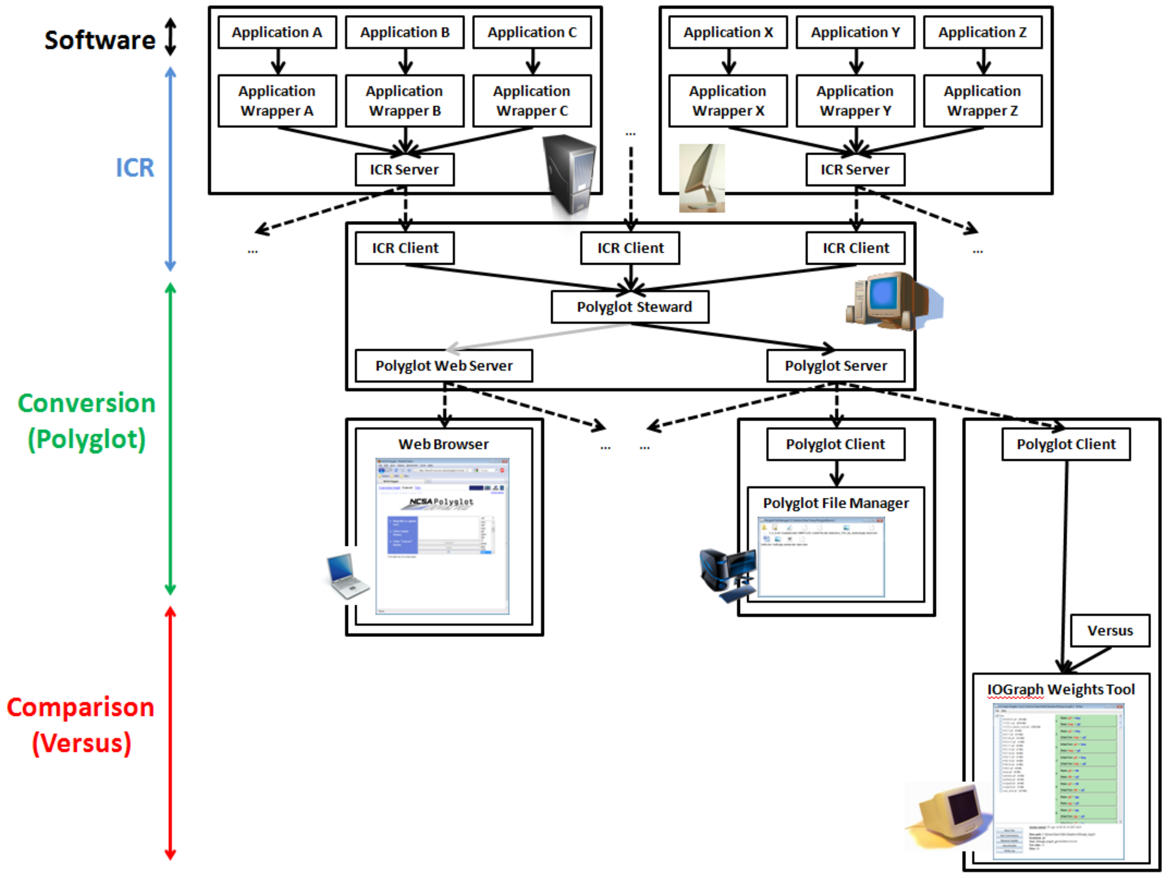

Figure 2. *The overall design of NCSA Polyglot with each layer color coded as in Figure 1. Again at the very bottom we have various closed source applications which we wish to utilize functionality from. These applications themselves run on various distributed machines and different OS platforms. Each of these machines runs an ICR server whose job it is to execute wrapper scripts as needed in order to perform operations from connected ICR clients. It is these ICR clients that provide a consistent API for other software to use. A Polyglot steward on yet another machine can be instantiated to open multiple ICR client sessions and organize their capabilities into a single I/O-graph. The Polyglot steward then in turns provides an API designed specifically for conversions, hiding away the fact that these conversions will be carried out on multiple distributed machines. This conversion service can then be run as a web service (via a Polyglot web server process) or through a desktop application (through a Polyglot server/client/file manager). The I/O-Graph weights tool can be run as well, using its own Polyglot client connection and Versus in order to assign weights to the various conversions.*

Acknowledgements

This research was supported by a National Archive and Records Administration (NARA) supplement to NSF PACI cooperative agreement CA #SCI-9619019.

References

[1] L. Marini, R. Kooper, K. Mchenry, M. Ondrejcek, and P. Bajcsy, "Content-Based File-to-File Comparison Services," Microsoft eScience Workshop, 2010 (submitted).

[2] K. McHenry and P. Bajcsy, "Framework Converts Files of Any Format," Society of Photo-Optical Instrumentation Engineers (SPIE) Newsroom, 2009.

[3] K. McHenry, R. Kooper, and P. Bajcsy, "Taking Matters into Your Own Hands: Imposing Code Reusability for Universal File Format Conversion," in Microsoft eScience Workshop, 2009.

[4] K. McHenry, R. Kooper, and P. Bajcsy, "Towards a Universal, Quantifiable, and Scalable File Format Converter," in The IEEE International Conference on eScience, 2009.