A Mosaic of Software

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

***Abstract*—In this paper we describe a Software Server, a background process that in conjunction with a central repository of lightweight wrapper scripts allows functionality within heterogeneous software to be called in a simple and consistent manner. The key role of the Software Server is to provide a common interface to software functionality in a manner that can be programmed against, in essence re-introducing an API to compiled code. Using the Java restlet framework, we provide a RESTful interface consisting of URL endpoints allowing any programming/scripting language capable of accessing URLs to utilize software functionality as a black box. In addition to being widely accessible the RESTful interface allows for a secondary role from Software Servers by giving them the ability to turn any traditional desktop software into a cloud based web service. In this paper we describe these Software Servers, the scripts we use to wrap primarily GUI based software, and show how these servers allow software to be called and interconnected into workflows across distributed machines. Finally, quantitative experiments showing the feasibility of the described Software Servers on a number of applications are presented.**

I. Introduction

There is a great deal of software on desktop machines. Even with mobile apps making a surge, graphical user interface (GUI) based desktop software is still the primary outlet for high end day to day productivity software. The world however is clearly changing. Distributed, scalable, persistent, robust, mobile, the "cloud" are all things that are becoming more and more prominent today. Classical desktop software does not exactly fit into this new realm of light weight clients, web services, and remote/flexible resources (both CPU and storage).

In the sections below we describe a Software Server. Where conventional web servers allow data to be accessible from anywhere within the web, Software Servers allow arbitrary software functionality to be accessible over the web. The shared software functionality is accessible through a uniform RESTful interface allowing all software to be used in a similar manner. While the notion of integrating distributed software has been addressed before in a variety of workflow publications [1, 2, 3] we emphasize the ability to utilize arbitrary software, a simple/fixed/concise/widely accessible interface, and overall ease of use. Through the uniform API imposed by the Software Servers, functionality within compiled software can be used as part of any workflow system [4, 5] or reused directly within novel code. Because this API is web based Software Servers also serve as a bridge between contemporary

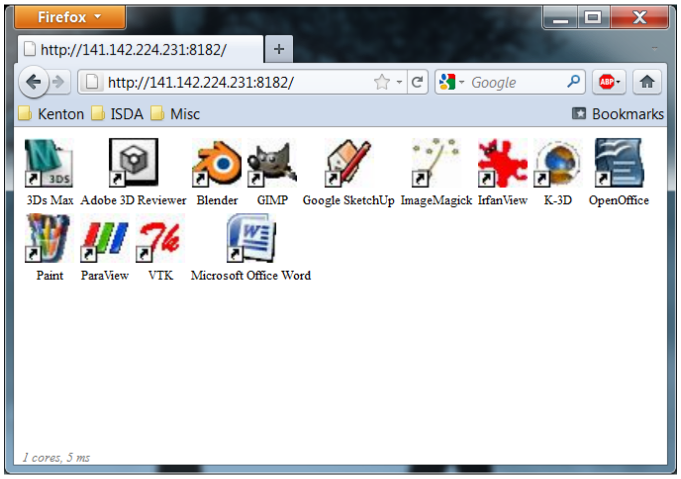


Figure 2. *Software servers allow desktop software to be used within new code via a re-attached RESTful API. This RESTful API also allows users to access software functionality through a browser. In the image shown a user has pointed their browser to the IP address of a machine hosting a Software Server which is sharing functionality from a number of installed applications. The page shown is made to look like a traditional file manager, presenting icons for each of the available applications. If the user were to click on an icon they would be taken to a web form which would allow them to choose an input file, an output format, a task to perform using the selected software, and a "submit" button allowing them to carry out the task on the remotely installed software.*

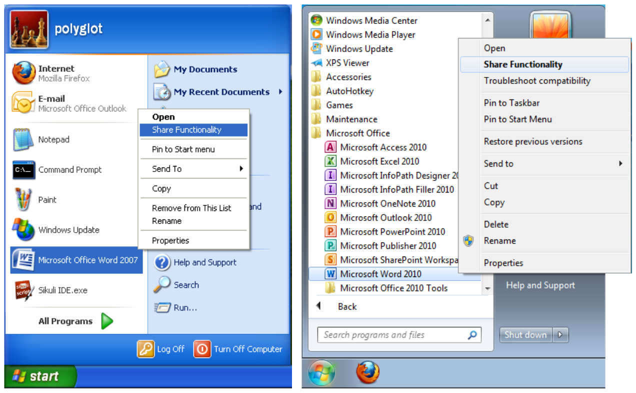


Figure 1. *Functionality within traditional desktop software is made available in the "cloud" through a RESTful interface by right clicking on it and selecting "Share Functionality" from the menu. Behind the scene a remote software registry is queried for associated wrapper scripts, found scripts are downloaded, configured, and a Software Server is started. Remote users can access a select set of operations within the software through any programming/scripting language capable of accessing URL's or through a web browser.*

GUI based desktop software and the "cloud" paradigm. In Section II we describe the Software Servers and the wrapper scripts required to make them work. In Section III we describe potential uses of such functionality. In Section IV we address the robustness of these servers and give our conclusions in Section V.

II. Software Servers

Sharing data (i.e. files) remotely is fairly easy via a number of services such as ftpd, files servers such as nfsd and smbd, and of course web servers such as httpd and tomcat. Sharing programs (i.e. applications) is a bit different in that while programs are nothing more than files, these files are distinct in that they contain instructions that execute on a given hardware platform and operating system. To use a program you must have the correct hardware (real or virtual), operating system (real or emulated), and a means of interacting with the program (e.g. keyboard, mouse). Programs can have a variety of interfaces from command line interfaces to graphical user interfaces. Sharing access to programs with command line interfaces can be done via a number of services such as telnetd and sshd. These programs are utilized remotely exactly as they would be on the local machine, via the keyboard. Sharing access to programs programs with graphical interfaces can also be done via a number of services such as VNC and rdesktop. These programs too are utilized remotely exactly as they would be on the local machine, via a keyboard and mouse.

The proposed Software Server like telnet, ssh, VNC and remote desktop provides access to remote software. The key difference is how it makes that software available. While the previously mentioned services provided the same interface one would have on the local machine, a Software Server replaces this interface. The motivation for replacing the interface is twofold. The first reason is a need for consistency in that we wish to interact with different programs in the exact same way. The second reason is that we wish to have an interface that can be programmed against. It is fairly trivial to call command line software within a script or program by simply making a call to the local system. The same cannot be said of software with graphical interfaces however. We wish to be able to call any program from within newly created code and to do so in exactly the same way. Our Software Servers utilize wrapper scripts to interact with arbitrary 3^{rd} party software. In the subsections that follow we will describe these wrapper scripts, the means by which these scripts are obtained, and the interface that the Software Server then presents.

A. Wrapper Scripts

The Software Server interfaces with 3^{rd} party software by calling small wrapper scripts that encapsulate equally small pieces of functionality within the software. We place no requirements on the interfaces a particular piece of software supports. In this paper however we will focus on what we consider to be relatively the most difficult interface to interact with in an automated manner, specifically graphical user interfaces designed for human interaction. The wrapper scripts can be written in any text based scripting language. The text based requirement comes from how we store information about the scripted operation within the script. Scripts must follow specific naming conventions and comment conventions in order to be used by the Software Server. The functionality of a particular script is determined by its name which takes the form:

*alias_operation.**

The alias above is simply a short name used to identify a particular application. While this alias can be anything it should be consistent among scripts that operate on the same application. The operation identifies the particular functionality of the 3^{rd} party software that the script controls. Operations supported by the current Software Server implementation are shown in Table 1. The "open", "save", and "convert" operations deal with opening and saving files in the software. Currently Software Servers use files as the primary means of passing inputs and obtaining outputs from software. The "exit", "kill", and "monitor" scripts deal with maintaining the state of the Software Server. Software behaves unpredictably at times. If not dealt with any deviation from the normal operation of the software could potentially bring down the server (e.g. a dialogue box asking to overwrite an existing file or to install an update). If an application fails to respond with output after a designated amount of time and a present "monitor" script is unable to recover the application, the "kill" script forcibly kills the process allowing the server to either try again or move on.

With only the "open", "save", and "convert" scripts one can perform file format conversions with the 3^{rd} party software. There is additional functionality within most 3^{rd} party software that can be scripted however. Consider for example the filters within photo editing software to change contrast or perform edge detection. Perhaps even functionality as simple as changing the font. A "modify" script allows for these types of operations that modify data. Unlike the other types of scripts the "modify" script operation is implied by any non-recognized script name (i.e. scripts should never be named *alias_modify.**). Instead the operation specified in the name should indicate what the modification is (e.g. "blur", "edges", "fontToArial", "tableToChart").

The remaining information required by the Software Server in order to use the script is stored as commented lines at the top of each script. The first of these commented lines is required by all scripts and contains the full name of the software along with the version of the software in parenthesis. The next lines are only applicable to operations that take arguments (i.e. files). The second line will indicate the types of data operated on (e.g. images, documents, 3D models, etc...). The last two lines will contain comma separated lists of file extensions that are accepted as inputs and/or outputs (depending on if this is an "open", "save", or "convert" script). An example comment "header" is shown below for a script "GSkUp_open.ahk" which will open a file using Google SketchUp:

```
;Google SketchUp (v7.0)
;model
;3ds, ddf, dem, dwg, dxf, skp
```

Operation	Arguments	Description
open	1	Open the file given as an argument.
save	1	Save the data currently open within the software to the file name given as an argument. The format of the output file is determined from the output file's extension.
convert	2	Perform both the "open" and "save" operation within the same script. Open the file specified by the first argument and save it to the file given as the second argument.
exit	0	Exit the application by closing the software.
kill	0	Forcibly exit an application by killing its process.
monitor	0	Monitor a given application for infrequent events that could hinder the usage of the software.
"modify"	0 or 2	Modify data loaded from a file. This operation can take no arguments and be called between an "open" and "save" opertion, or, it can take two arguments opening the file specified by the first argument, modifying the loaded data, and saving the result to the file specified by the second argument. The "modify" operation is implicit in that it is assumed as the operation in the case where the operation name doesn't match the above list.

Table 1. *The operation names supported by Software Servers. The "open", "save", and "convert" deal with loading and saving files within the software. The "exit", "kill", and "monitor" scripts deal with maintaining the state of the machine running the Software Server, specifically watching for rarely observed occurrences such as warning dialogue boxes and forcefully killing applications that are too far gone. The modify scripts deal with operations that don't involve loading and saving files such as smoothing an image, changing a font, or building a chart.*

The information contained within the script name and header is all the Software Server needs in order to use the script. The rest of the script is responsible for actually performing the operation it claims to perform. As our focus is mainly on graphical user interface (GUI) based software we will briefly describe two of the languages we have used to script functionality within such software: AutoHotKey [1] and Sikuli [2].

1) AutoHotKey: AutoHotKey is a GUI scripting language that makes use of the messages that widgets within the Windows OS use to communicate and respond to events. Through tools such as AutoIT Spy and Winspector Spy one is able to identify the the ID's of particular widgets as well as intercept and store the message passed during a desired event (e.g. a mouse click on a menu item). These messages can be posted (or replayed) from within a script in order to reproduce a given GUI interaction. AutoHotKey also allows scripting solely based on the text found within windows and widgets (allowing one to avoid the extra step of identifying widget ID's and the messages used for a particular event).

We utilize AutoHotKey primarily for Windows applications [3] though it appears that there are now versions for Linux and OS X [4]. AutoHotKey scripts we have created for software operations have tended to be fairly small. The longest script the authors have encountered for a particular operation was roughly one hundred lines (the bulk of this being an if-statement to choose the correct action for a given format). Most scripts are far shorter.

2) Sikuli: Sikuli [6, 7] is a vision based GUI scripting language written in Java. Unlike AutoHotKey, Sikuli focuses exclusively on what is seen on the desktop by taking screen shots of the desktop and comparing them during script execution to know what is happening and where. Sikuli is written as an extension of Python, using JPython, and provides a very convenient IDE for writing scripts. The provided IDE allows a script writer to walk through the automation of a GUI application, pause periodically, and capture images from the screen to use within the script. Clicking a button from the script is as simple as calling the "click()" function and passing into it an image of the button to be clicked (with the image being easily obtained from within the IDE).

Because Sikuli is written in Java and makes no use of specific system functionality to control the GUI it can be used on a variety of platforms. Because it acts as an extension to a well known scripting language and uses a convenient IDE, Sikuli scripts are fairly easy to write. Our main motivation for looking at Sikuli was to avoid timing issues that can sometimes occur within our AutoHotKey scripts. Specifically, it is important to not click on a button until it is actually present on the screen. While this is usually manageable within AutoHotKey it can sometimes be a challenge. A vision based language gets around this by operating much like a human being would, in that a human using a GUI based application will not click on a button until the button appears and they see it.

A drawback of Sikuli is that the created scripts are not necessarily portable. A consequence of being vision based the script is dependent on the appearance of the desktop (such as fonts used for text and other window styles/themes). In addition we have noticed that Sikuli scripts run much slower than AutoHotKey scripts that perform the same task.

B. Obtaining Scripts

The work of [8, 9] used a less developed notion of Software Servers to create an extensible and distributed conversion service for the purpose of measuring information loss across file format conversions. As an extension to that work a web based registry was created to document software [5] emphasizing the inputs and outputs supported. This registry, called the Conversion Software Registory (CSR) [10], allows one to easily identify 3^{rd} party software capable of performing conversions between a given input and output format based on file extensions. In order to make it easier to install new instances of the Polyglot conversion service described in [8] we began associating Software Server wrapper scripts with the

[1] http://www.autohotkey.com
[2] http://sikuli.org
[3] http://isda.ncsa.illinois.edu/drupal/content/software-servers
[4] http://www.autohotkey.com/forum/topic54494.html
[5] http://isda.ncsa.illinois.edu/NARA/CSR

Task	Operation Sequence
convert	convert
convert	open, save
X	modify
X	open, modify, save

Table 2. *The tasks a Software Server broadcasts (generated from the wrapper scripts available). Conversion tasks are made up from a single "convert" script or a pair of "open" and "save" scripts. Tasks involving modify scripts are named according to the name of the modify operation (indicated by X's above). Modify tasks can be made of a single "modify" script if it takes two arguments (an input and an output) or an "open", "modify", and "save" script if the modify script takes no arguments (indicating that it deals with whatever is currently open in the application at the time).*

software entries of the CSR database. Doing this has made the process of sharing software functionality extremely simple for an end user.

A script installer tool provided as part of the Software Server searches the local system for installed software (on a Windows system this is done by searching for uninstallable applications within the system registry). For each locally installed application the tool will query the CSR for matching software that also contains associated wrapper scripts. If scripts are found they are download and configured to run on the local system (e.g. correcting for any software path variations that occur). Once configured the Software Server can be started and will share the functionality scripted by the union of obtained scripts.

C. Tasks

The Software Server does non allow for the direct calling of individual scripted operations as some don't make sense when called alone. For example calling an "open" operation to open a file within an application has no purpose unless a "modify" and/or "save" follows. In addition if a sequence of operations is requested of the server such as an "open", "modify", and "save", the server must guarantee that these operations are executed all at once in order to prevent other requests on the same application from corrupting the desired result. To do this the Software Server hides individual operations and instead presents "tasks". These tasks are nothing more than a sequence of operations to obtain a particular result which are guaranteed to be run without interruption. Tasks are automatically generated from the given scripts (see Table 2). Using the inputs/output operations alone allows for format conversions tasks. If modify operations are available they are included between open/save operations in order to pass in an input, carry out the modification, and return some output. We point out that one usually creates a "convert" operation for an application or an "open"/"save" pair, not both. If both are present the server will choose the direct conversion. The same goes for two argument and no argument "modify" scripts.

D. RESTful Interface

The primary motivation for the creation of these Software Servers is to provide an interface that is consistent among all software, simple, widely accessible, and capable of being programmed against. To meet these requirements we have decided to go with a RESTful interface based on the Java Restlet [6] framework. Built on top of the HTTP protocol restlets are a popular interface for web services in general. Software Servers are accessed via URL endpoints of the form:

```
http://host:8182/software/[application]/[task]/[output]/[input]
```

The host is simply the name or IP of the host running the Software Server. The "[application]" in the URL specifies the alias of the application to use. In turn the "[task]" specifies the task to carry out, "[output]" specifies the desired output format, and "[input]" the input file to perform the task on. The parameter options available on the server can be obtained by accessing the various endpoints. For example to determine what software is available on the host one would visit:

```
http://host:8182/software/
```

to obtain a list of available software. To then see what tasks are available for one of the available applications one can then visit:

```
http://host:8182/software/[application]/
```

where "[application]" is one of the aliases displayed at the previous URL. To see what output formats are supported by a given task one can then go to:

```
http://host:8182/software/[application]/[task]/
```

where "[task]" is one of the tasks displayed at the previous URL. Finally to see what input formats are supported for a given application task for a given output format one can go to:

```
http://host:8182/software/[application]/[task]/[output]/
```

where "[output]" is one of the outputs displayed at the previous URL. To carry out a task one can either POST a file to the URL above or URL encode the URL to a file on the web and append it to the end. When this is done the Software Server will immediately return with a URL to where the result file can be downloaded. The execution of the task will depend on the tasks position in the queue and on the specs of the machine running the software. Attempting to access the download URL before the task is complete will result in a "404 - File not found". A user in a browser, program, or shell script can use this error to wait until the file is available. We point out that the benefit of going with the chosen RESTful interface is that the shared software functionality can be accessed across many programming/scripting languages in addition to browsers. Specifically any programming or scripting language that supports accessing URLs can treat this shared software functionality as just another function within a library and literally hide the fact that the "black box" is actually software running on a remote machine.

Software servers provide additional information through various other endpoints such as:

```
http://host:8182/alive
```

which will simply return true if the the server is running,

```
http://host:8182/busy
```

which will notify the accessing user/program as to whether or not the server is currently busy running an application per another request,

```
http://host:8182/processors
```

which will return the number of processors available on the host machine, and

[6]http://www.restlet.org

```
http://host:8182/memory
```

which will return the amount of memory available to the Java virtual machine on the host machine.

E. Security

Software servers can be configured so as to require a password in order to access the software functionality provided. In addition a special admin user can be specified for operations which go beyond the access of software functionality such as:

```
http://host:8182/screen
```

which will return a screen shot of the host machines desktop and

```
http://host:8182/reboot
```

which will remotely reboot the host machine. Both these capabilities are useful for checking on and resetting a bogged down server but at the same time should not be exposed to everyone that is allowed to use the Software Server. If passwords are enabled and a user wishes to utilize the Software Server functionality within a program or script they can include the username and password within the URL:

```
http://username:password@host:8182/software/...
```

III. Applications

One should think of Software Servers in the same way that they would think of a dynamic or static library. Both provide functionality and make it accessible through a specified API. How that functionality is carried out is of little importance. We refer to this process of re-attaching an API to compiled software as *imposed code reuse*. In this light anything that you would expect to do with a dynamic or static library can also be done with a Software Server. Below we describe several applications that utilize Software Servers.

A. Distributed Software Servers

A Distributed Software Server provides the same RESTful interface as a regular Software Server but runs no software of its own. Instead it listens for available Software Servers, catalogues the capabilities of those it finds, and presents the union of the found Software Server capabilities as its own. Cataloguing the available functionality within a Software Server is a simple matter of crawling the URL's shown in the previous section. When a request is made of the Distributed Software Server the catalogue of found Software Servers is searched and the request is passed on to the first non-busy server found.

The current Software Server implementation can, depending on its configuration, either broadcast its existence directly to a specific Distributed Software Server via TCP or multicast its existence to any number of Distributed Software Servers within a specified ttl (i.e. time to live) via UDP. When a Software Server configured in this way comes online it will spawn a thread to continuously notify the one or more distributed servers which will in turn add its functionality to its catalogue. The distributed servers will check on the individual Software Servers periodically to determine if they are still alive and if not will drop them from their catalogue. In this way Software Servers can come and go, expanding or degrading the capabilities of the Distributed Software Server without totally bringing it down.

B. Software as Libraries

Software servers re-attach an API to compiled code so that it can once again be called within code. As an example of that we show here a small bash script to convert the files within a directory to another format:

```
#!/bin/bash

host="http://141.142.224.231:8182"
application="A3DReviewer"
task="convert"
output="igs"
input="stp"
url=$host/software/$application/$task/$output

for input_file in `ls *.$input` ; do
  output_url=`curl -s -H "Accept:text/plain" -F "file=@$input_file" $url`
  output_file=${input_file%.*}.$output
  echo "Converting: $input_file to $output_file"

  while : ; do
    wget -q -O $output_file $output_url

    if [ ${?} -eq 0 ] ; then
      break
    fi

    sleep 1
  done
done
```

As a bash script this is only a very simple example of what is possible. The available software functionality can be chained together to produce more complex workflows. Any language capable of accessing URL's (e.g. Java, Python, etc...) can be used. The RESTful API itself can be wrapped within the code to make the call look like a more native function call (as opposed to accessing a URL).

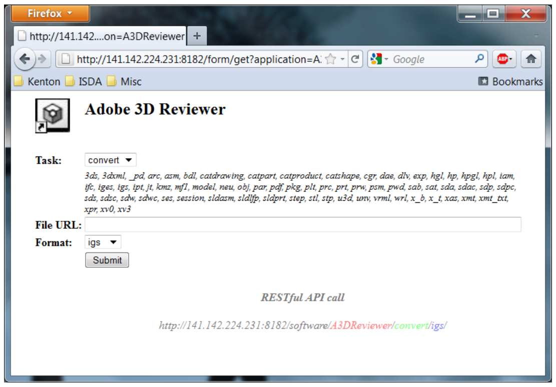
C. File Format Conversion

The driver for the creation of the described Software Servers was to construct a practical nearly "universal" file format converter. The task of building such a conversion engine by implementing the needed loaders/transcoders for the thousands of different file formats (many proprietary, closed, and/or with lengthy specifications) was unrealistic. Given that the main reason so many formats exist is that software vendors tend to create new formats specific to their own applications and the fact that their software would load their own formats plus often a handful of others it seemed like a good idea to use available software to build such an engine. Clearly the software had to be automated for this to be a true service that could convert potentially millions of files. With the help of AutoHotKey this was possible.

The conversion service referred to as NCSA Polyglot [8, 9], named for a person who speaks many languages, constructs a directed graph from the software available across a number of Software Servers. This graph referred to as an input/output graph (or I/O-graph) contains vertices that represents the union of the file formats supported by the available software and directed edges between these vertices/formats for each application that is directly capable of converting from the source to the target. New conversions can be created between formats not supported by any single application by looking for a shortest path between a given source and target vertex within the graph. Conversions are carried out by submitting individual tasks to their respective Software Servers.

D. Software Functionality Sharing

The Software Servers in combination with the script installer tool and the Conversion Software Registry allow for


Firefox
http://141.142....on=A3DReviewer
http://141.142.224.231:8182/form/get?application=A
Google
Kenton
ISDA
Misc
Bookmarks

Adobe 3D Reviewer

Task: convert

3ds, 3dxml, _pd, arc, asm, bdl, catdrawing, catpart, catproduct, catshape, cgr, dae, dlv, exp, hgl, hp, hpgl, hpl, iam, ifc, iges, igs, ipt, jt, kmz, mf1, model, neu, obj, par, pdf, pkg, plt, prc, prt, prw, psm, pwd, sab, sat, sda, sdac, sdp, sdpc, sds, sdsc, sdw, sdwc, ses, session, sldasm, sldlfp, sldprt, step, stl, stp, u3d, unv, vrml, wrl, x_b, x_t, xas, xmt, xmt_txt, xpr, xv0, xv3

File URL:

Format: igs

Submit

RESTful API call

http://141.142.224.231:8182/software/A3DReviewer/convert/igs/

Figure 3. *The form presented to a user after clicking on a software icon within the browser view of a software server. From this form the user can select a task to perform, select an input file, select an output format (which may be the same as the input format), and submit the task to be executed. To emphasize the available RESTful API access to the software functionality we display the API call below and update it in real time as the user makes selections. The user can achieve the same effect of pressing the "submit" button by simply accessing the shown RESTful API URL.*

a very convenient means of sharing specific software functionality. On the Windows OS one can typically share data simply by right clicking on a folder and selecting "Share". By doing this data can be accessed remotely in manner that looks identical to the way one would access the data if it were on the local machine. Software servers allows something very similar to be done in terms of software functionality. We modify the Windows 7 registry as follows to add a "Share Functionality" item to the context menu for shortcut (*.lnk) files:

```
[HKEY_CLASSES_ROOT\lnkfile\Shell]
[HKEY_CLASSES_ROOT\lnkfile\Shell\Share]
@="Share Functionality"
[HKEY_CLASSES_ROOT\lnkfile\Shell\Share\command]
@="C:\\Users\\polyglot\\Desktop\\Polyglot2\\Share.bat \"%1\""
```

When the "Share Functionality" option is selected within the right-click menu presented on top of an application's shortcut icon, Figure 1, Windows will call "Share.bat" with the name of the shortcut as an argument. The shortcut name usually identifies the software thus we use this to identify the scripts to download from the CSR. Once the associated software scripts are downloaded they are configured for the local system and an instance of the Software Server is started. At this point the scripted software functionality is available via the RESTful interface discussed previously. While the main driver behind the development of Software Servers has been the imposed API on software to utilize its functionality within new code, due to the RESTful interface one can also access the functionality via a web browser. In Figure 2 we show a browser pointed to the IP address of a Software Server sharing functionality from a number of applications. By default an icon for each of the available applications is shown in a layout made to be similar to that of the file managers found on most modern operating systems. By clicking on one of the icons a user can access the software's functionality via a web form. From this form, shown in Figure 3, the user can select the task to perform, the input file, and the format of the output file. When a task is submitted by a user it will be carried out on the Software Server and the output file returned to the user via a link shown to them immediately after pressing the "submit" button. When the output file is ready the user can click on the link to download it.

IV. Robustness

The key concern in terms of implementation when automating software that was not designed to be automated is robustness. True libraries should be fairly robust. Cloud based services are expected to rarely go down. The fact of the matter however is that desktop software sometimes crashes. As mentioned in the previous sections we not only try to build the wrapper scripts as robustly as possible but include "exit", "monitor", and "kill" operations which allow the Software Server to regain control when something unexpected occurs.

We empirically evaluate the robustness of a Software Server by measuring the number of tasks per hour it is capable of performing. We perform this evaluation per application by performing "convert" tasks on randomly selected input files from a small data set to randomly selected output formats. For each application we consider two data sets consisting of 20 files each. The first set, referred to as the "valid" set, contains files with formats that are claimed to be supported by the called application. The second set, referred to as the "mixed" set, has half its files made up of formats claimed to be supported by the software and half of files not supported by the software but renamed to have extensions that match supported formats. The purpose of this "mixed" set is to deliberately try to break the Software Server by causing the application being called to choke. This set mimics an adversarial user who might be trying to bring down the system. In reality a service should be able to withstand incorrect inputs as this situation is likely to happen once in a while regardless of the intent of the users. We point out that there is no way for the Software Server to absolutely determine the validity of the input file as it relies solely on file extensions. We also argue that digging further into the file runs contrary to the idea of a Software Server which was built so one did not have to know the details of the many formats available.

Experiments were run on a Software Server running within a virtual machine running on VMWare's ESXi 4.1.0. The underlying hardware consisted of 2 4-core 2.5 GHz Xeon processors, 8 GB of memory, and 4 TB of hard disk space in a RAID 5 configuration. The virtual machine used utilized 1 CPU, 1 GB of memory, 100 GB of disk space and ran Windows XP. Random conversions requests were issued for a little over an hour to each of the applications evaluated, with each application being evaluated twice (one for each data set). The results of our experiment are shown in Table 3 and Table 4.

When given the valid data set the Software Server was able to perform on average 1394.71 tasks per hour with an average wait per execution of 4.42 s. Of the files submitted we estimate that on average 88.46% were converted successfully. A conversion is deemed successful if a non-empty file is created as output. Not surprisingly the applications with the highest throughput are those with command line interfaces such as ImageMagick and IrfanView (a GUI application that can also be called from the command line in a headless manner). When scripting applications for the Software Server we always try to choose the most robust means of using the software, meaning command line interfaces are chosen over GUI interfaces when possible. VTK and Blender also have high throughputs, both these applications having built in scripting capabilities that were again chosen over the GUI interface for the sake of robustness. The software that was scripted through the GUI interface tended to be slower with 3DS Max having the least throughput at 355.08 tasks per hour. We point out that 3DS Max is a very large high end application. In addition it generated a relatively large variety of dialogue boxes when opening/saving files making it a bit more difficult to script for all possible scenarios (a possible reason for the lower success rate of 67.13%). None the less 355 tasks per hour is not bad considering that this is GUI based software designed for human interaction, obtaining roughly one tenth the throughput of the fastest application. For a little context one need only ask how many people would it take to manually use this software to convert 355 files an hour. How much would you have to pay them? How long would they be willing to keep doing this incredible mundane job?

In the case of the mixed set it is convenient to see how the throughput changes relative to the ideal situation with the completely valid data set. Table 5 indicates the percentage of the ideal case achieved when the Software Server is run with data from the mixed set. We see that on average we achieve 55.41% of the throughput achieved in the ideal case at 64.36% the success rate. Both these values are close to 50% which is what we would expect given that half of the mixed set is made up of bad files. The important thing to take from this result however is that the Software Server and machine it was running on did not fail and new tasks were able to be carried out after unexpected situations were encountered. As we can see from last columns of Table 3 and Table 4 the applications kill script is called significantly more often in the case of the mixed set. The reason for this is that when the application attempts to open a file it believes is of a type it can open and it is in fact some completely foreign type that was renamed to fool it, the software will behave in some unexpected way. When this occurs the script controlling the software will often be lost. The Software Servers are configured to wait a maximum of 30 seconds for any one operation to complete. Thus once an application deviates from a script, as is often the case when opening these sabotaged files, the Software Server will wait 30 seconds and then call an associated kill script to forcefully kill the software and move on to another task. Based on the number of times called it is clear that these kill operations are critical in order to ensure that the Software Server does not fail. If the kill operation did not exist we can simply look to the last column of Table 3 and Table 4 to see how many times the Software Server would have failed per hour for each application.

V. Conclusion

In this paper we have described a web service that shares software functionality through an interface that is consistent among software, simple, widely accessible, and capable of being programmed against. These Software Servers allow arbitrary 3^{rd} party software to be used as a black box within new code while at the same time allowing conventional desktop software to be used within the “cloud” as web services. We have shown how new software can potentially be added to a given Software Server by a simple two click process similar to that of sharing folders in the Windows operating system and then accessed through a web browser. The only thing missing that would allow any software to be added to a Software Server via this simple process is a well stocked script repository. We hope that by gaining a community of users and providing tools such as IDE’s or IDE plugins specifically designed for GUI scripting we will be able to build such a repository.

Acknowledgments

This research has been funded through the National Science Foundation Cooperative Agreement NSF OCI 05-25308 and Cooperative Support Agreement NSF OCI 05-04064 by the National Archives and Records Administration (NARA).

References

[1] C. Pancerella, “The use of agents and objects to integrate virtual enterprises,” *SANDIA Report 8226*, 1998.

[2] R. Whiteside, E. Friedman-Hill, and R. Detry, “Pre: A framework for enterprise integeration,” *SANDIA Report 8505C*, 1998.

[3] P. Bajcsy, R. Kooper, L. Marini, B. Minsker, and J. Myers, “A meta-workflow cyber-infrastructure system designed for environmental observations,” *Technical Report ISDA01-2005*, 2005.

[4] B. Ludascher, I. Altintas, C. Berkeley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the kepler system,” *Concurrence and computation: Practice and Experience, Special Issue on Scientific Workflows*, 2006.

[5] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, “Taverna: A tool for building and running workflows of services,” *Nucleic Acids Research*, 2006.

[6] T. Yeh, T. Chang, and R. Miller, “Sikuli: Using gui screenshots for search and automation,” *UIST*, 2009.

[7] T. Chang, T. Yeh, and R. Miller, “Gui testing using computer vision,” *CHI*, 2010.

[8] K. McHenry, R. Kooper, and P. Bajcsy, “Taking matters into your own hands: Imposing code reusability for universal file format conversion,” *The Microsoft e-Science Workshop*, 2009.

[9] ——, “Towards a universal, quantifiable, and scalable file format converter,” *The IEEE Conference on e-Science*, 2009.

[10] M. Ondrejcek, K. McHenry, and P. Bajcsy, “The conversion software registry,” *The Microsoft e-Science Workshop*, 2010.

Application	tasks/hour	success rate	average wait	kills
3ds Max	355.08	67.13%	10.00 s (8.85 s)	37
Adobe 3D Reviewer	628.11	99.53%	5.54 s (3.99 s)	0
Blender	2024.71	100.00%	1.65 s (0.82 s)	0
Google SketchUp	362.00	95.36%	9.81 s (7.48 s)	1
ImageMagick	1871.34	89.59%	1.68 s (3.70 s)	12
IrfanView	3163.12	99.91%	0.92 s (0.85 s)	0
Microsoft Paint	795.74	93.66%	4.36 s (6.88 s)	48
Microsoft Word 2007	756.04	80.37%	4.60 s (3.39 s)	11
ParaView	750.93	93.41%	4.67 s (6.62 s)	47
VTK	3240.04	65.60%	0.94 s (0.43 s)	0
Average	1394.71	88.46%	4.42 s (4.30 s)	15.60

Table 3. *Software server robustness test results on the "valid" data set containing files with valid input formats. These results represent an ideal throughput for the given system.*

Application	tasks/hour	success rate	average wait	kills
3ds Max	180.04	36.61%	19.86 s (11.02 s)	67
Adobe 3D Reviewer	271.01	66.79%	13.15 s (11.37 s)	81
Blender	542.44	85.04%	6.5s (10 s)	82
Google SketchUp	221.97	64.16%	16.09 s (11.08 s)	80
ImageMagick	2291.69	45.70%	1.3 s (2.56 s)	7
IrfanView	2700.51	55.35%	1.06 s (1.1 s)	0
Microsoft Paint	198.54	46.77%	17.97 s (13.21 s)	106
Microsoft Word 2007	162.33	28.05%	22.05 s (11.84 s)	110
ParaView	282.96	68.40%	12.55 s (12.08 s)	87
VTK	2597.36	69.02%	1.21 s (1.1 s)	0
Average	944.89	56.59%	11.17 s (8.54 s)	62.00

Table 4. *Software server robustness test results on the "mixed" data set containing files with half valid input formats and half invalid input formats renamed to appear as valid input formats. These results represent the throughput achieved given a hostile user attempting to deliberately hinder the system. The important thing to note here is that the server did not fail and the overall effect was a diminished throughput. In an actual system such a user can be easily detected and banned from the service.*

Application	tasks/hour	success rate	average wait
3ds Max	50.70%	54.54%	198.60% (124.52%)
Adobe 3D Reviewer	43.15%	67.11%	237.36% (284.96%)
Blender	26.79%	85.04%	393.94% (1219.51%)
Google SketchUp	61.32%	67.28%	164.02% (148.13%)
ImageMagick	122.46%	51.01%	77.38% (69.19%)
IrfanView	85.37%	55.40%	115.22% (129.41%)
Microsoft Paint	24.95%	49.94%	412.16% (192.01%)
Microsoft Word 2007	21.47%	34.90%	479.35% (349.26%)
ParaView	37.68%	73.23%	268.74% (182.48%)
VTK	80.16%	105.21%	128.72% (255.81%)
Average	55.41%	64.36%	247.55% (295.53%)

Table 5. *Software server robustness results showing difference between "valid" and "mixed" data sets (indicated as a percentage of the throughput achieved in the ideal case using the "valid" data set).*