Project MXC-403

Augmented Unix Userland

Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Computer Science
By

Sam Abradi
stabradi@wpi.edu

Ian Naval
ianaval@wpi.edu

Fredric Silberberg
fbsilberberg@wpi.edu

Submitted On: April 30, 2015

Project Advisor:
Professor Michael Ciaraldi
ciaraldi@wpi.edu

Abstract

The Unix philosophy has resulted in stringing together simple general purpose utilities to accomplish complex tasks. However, using text as a communication medium between these programs requires several formatting utilities to transform output from one program into valid input to another. These transformations can be very complicated and confusing. This project looked at two alternative shell designs that use different communication methods between programs, simplifying interprogram communication and leveraging existing technologies to make developing new utilities and shell scripts easier.

Contents

List of Figures

List of Tables

1 Introduction

The Augmented Unix Userland (AUU) project explored non-traditional alternatives to the design of communications between traditional core-utility programs in Unix-based systems, such as Linux, Mac OSX, and BSD variants. In this project, we explored and evaluated two different solutions that partly or entirely change the manner in which console based programs communicate with each other and the user, in the hopes of improving, or Augmenting, user experience. In this section, we will provide a brief overview of terminology necessary to understand this project, discuss our motivations for this project, and finally outline the rest of the paper.

1.1 AUU Terminology

The traditional definition of userland is "Anywhere outside the kernel... This term has been in common use among Unix kernel hackers since at least 1985."[10] In this project, we looked at improving a more limited subset of programs. We restricted our definition of userland to be programs that run inside a traditional shell, including the widely used Bash (Bourne Again Shell) or zsh (Z shell). Additionally, while the concepts we explored in this project are broadly applicable to most Unix based systems, and even to Microsoft Windows, our development and primary testing was all done on a few flavors of Linux, namely Arch Linux and Ubuntu.

1.2 Motivation

Unix was designed with the idea that complex tasks could be accomplished by splitting up work among simpler programs and combining them together. Programs in a traditional Unix userland should "do one thing and do it well." [9] This is called the Unix Principle. However, due to this principle, programs need to communicate with each other in order to for the user to accomplish more complex tasks. A good example of this trying to determine whether a process is running when you only know part of the process name. If you know the full name of the program then it is possible to use the pidof command, which will return an error if the program is not running. Since we do not know the full process name, however, we need to use the following two programs:

- ps: ps lists all of the processes currently running on the system. It only lists those processes that are running under the currently logged in user by default, but command line options are available to customize the output.

- grep: grep searches its input for matches to a pattern specified as an argument to the program. Additional flags control how it prints matches and how it searches. Files can also be specified as arguments, and grep will search them for matches. If no files are specified, it will look in stdin (i.e. standard input).

In order to search for the process, we need to pipe the output from ps to grep. In Unix-based systems, every processes has three file descriptors called stdin (standard input), stdout, and stderr (standard error). These can be manipulated by the terminal emulator, including joining, or piping the stdout and stdin of two programs. Assuming that we are searching for the process "foo" our final command for this is:

```
ps -e | grep foo
```

Figure 1: Bash command for finding process "foo"

This is a trivial example, and Bash style piping is significantly more powerful than this. However, it does have some disadvantages as well, and it is these disadvantages that we seek to address in this project.

More complex use cases for piping involve formatting the output of one program into an expected format for another program. For example, a user might want to take a listing of processes, their process ids, and what user is running them, and convert them into a comma-separated values (csv) file for easy analysis in a program like LibreOffice or Excel. They might also only want a subset of processes, such as processes that start with a certain prefix. We can again use ps and grep for this. However, here the user will also need to use sed in order to extract desired information from the stream and put it into csv form. sed (stream editor) is a small Unix utility that is used to parse and manipulate text. In order to be able to parse out the information, the user will need to write a regular expression (regex) for sed, which can be a tedious processes, and the resulting regex is usually not easily understood by other users. Additionally, it is also usually not reusable except for manipulating this specific output specifically with sed, as the regex was created for this one input. If further manipulation of the source text or resulting csv is needed, the user will likely need to use awk to do so. awk is a small utility with a self-named programming language designed for working with files and text, and while it can be very powerful, it also shouldn't be necessary.

Usage of grep, sed, and awk is a recurring theme with most bash scripting, and most users who ever want to do serious bash scripting will likely find themselves needing to use the three programs at some point or another. The AUU MQP seeks to effectively remove the need for sed and awk from most situations by exploring

different methods of program communication. In current shells, userland programs communicate through unstructured raw text. Because this text is unstructured, an individual line of output from a program has no semantic meaning until we as humans look at it, or until it is run though some kind of parsing tool designed to extract semantic information from it. This can be as simple as the aforementioned sed regex, or it could be something as powerful as a machine learning model. However, that extra step of extracting semantic information from the text always needs to be there, because all of that information was lost when the generating program converted the data into text. We seek to remove this loss of semantic information by removing unstructured text from the equation; programs can and should communicate with modern standards that convey more information than unstandardized raw text.

1.3 The AUU Project

In this project, we looked at two different methods of removing unstructured text from program communication. The first is through use of the Javascript Object Notation (JSON) format and a organization methodology we call streams, which we describe in Section 2. We implement this design as a proof-of-concept, and then examine the strengths and weaknesses of the protocol. Next, we take the lessons learned from our initial concept and apply them to another unstructured text alternative, which we are calling the Python Shell (PSH). This second solution is described in Section 3. We implemented a smaller proof-of-concept for PSH, and we then examine its usability, strengths, and weaknesses. Finally, we examine the work we've completed, and what the next steps of this project are.

2 JSON Shell

In this section, we explain our initial motivation for designing a protocol for the Augmented Unix Userland based on JSON. We then analyze both the advantages and shortcomings of this design.

2.1 Motivation: Why JSON?

JSON was originally designed to be used for JavaScript[7], but it has since become more of a general purpose object serialization format. The syntax of JSON is simple, and it is for this reason that JSON has become so ubiquitous. Its ubiquity makes it a good choice for this communication design. Many popular programming languages have libraries available for marshalling and unmarshalling objects to and from JSON including Python (json), Golang (encoding/json), Java (google-gson/jackson), C (jansson), and Ruby (json). Additionally, due to the simplicity of the language and similarity to many forms of pseudocode, it can be easily understood by others, even if they are not initially familiar with JSON.

2.2 JSON Protocol Design

The JSON-based Augmented Unix Userland relies on the idea of building core utility programs that interface with each other via JSON. In JSON, data is organized in a hierarchical tree. No matter what programming language or library is used to serialize the data, JSON requires that the marshalled content be fully evaluated before it is serialized. This imposes significant limitations on the functionality of any shell which wraps these userland utilities.

In a traditional Unix system, the Unix pipe can be used to join the standard output of one program with the standard input of another. A well-written shell (e.g. bash) will launch both processes and allow the operating system to run them in parallel, and data is processed as soon as it becomes available to the receiving program. With standard JSON as the interface, the sending program must first evaluate the entirety of the output before marshalling and printing to standard out. Our JSON protocol introduces a concept called 'Streams', which allows us to add process-level parallelization to our utility programs.

Streams

We are taking the traditional concept of Streams and sending updates from program to program in frames. This allows us to easily enable parallel processing by turning

standard input and output into constant Streams of data instead of having to buffer all of the output from one program, wait for it to finish, and then pass it along to the next program. We do this by having the output of a program be an array of output blocks, each with its own StdOut and StdErr. These are arrays themselves, and each named object becomes a substream.

Standards and Definitions

Our JSON Stream protocol is designed as a set of standards on top of the existing definitions provided by RFC 7159.[7] These standards and definitions exist for the benefit of anyone who would like to implement the standards. We follow the terminology standards specified in RFC 2119.[6] We define the following rules:

- All output must be standard JSON as defined by RFC 7159.
- The top-level JSON value must be a standard JSON array as defined in section 5 of RFC 7159.
- The elements of the top level array (henceforth called `JSH frames`) must be standard JSON objects as defined in section 4 of RFC 7159.
- `JSH frames` must contain two members whose names are "StdOut" and "StdErr" and whose values are standard JSON arrays.
- A top-level `JSH Stream` is an array of standard JSON values. The concatenation of arrays from the JSH frames constitute the elements of a top-level JSH Stream. Only the arrays whose associated names are the same across all frames.
- A `substream` is an array of standard JSON values. A substream has a parent Stream and an associated name, and any Stream can have an arbitrary number of substreams. These substreams elements are derived by iterating over the elements of the parent Stream. For each element of the parent Stream, the following process is applied. If the element is an array, then its values are concatenated to the substream. If the element is an object and contains a value corresponding to the substream's associated name, then that value is added to the substream. If the element is neither an object nor an array, then the element itself is added to the substream.

For a more intuitive understanding of these concepts, it is easier to explain with an example.

Example Use Case: ps

In this example, we explore the output of a Unix program called `ps`. For the sake of brevity, we use a simplified version of `ps` which only provides the names and process IDs of running processes. Figure 2 illustrates the sample output.

```
[{
  "StdOut": [
    {"processes": [
        {"pid": 1, "name": "init"}
    ]},
    {"processes": [
        {"pid": 2, "name": "bash"},
        {"pid": 3, "name": "ps"}
    ]}
  ],
  "StdErr": []
}]
```

Figure 2: Sample Output of ps with a Single Frame

In Figure 2, the outermost array contains exactly one JSH frame. This array is specified under the second rule in Standards and Definitions. There are two top-level JSH Streams present: "StdOut" and "StdErr". "StdOut" has one substream, denoted by the name "processes", which can also be expressed as a JSON array as shown in Figure 3. "processes" has two further substreams called "pid" (an array of integers) and "name" (an array of strings).

```
[
  {"pid": 1, "name": "init"},
  {"pid": 2, "name": "bash"},
  {"pid": 3, "name": "ps"}
]
```

Figure 3: Processes Stream As Array

Because of the Stream protocol's flexibility, the same information can be represented as a single array under the "processes" name within the JSH frame as shown in

Figure 4. In fact, the information can be split across multiple JSH frames as long as the same names are used and the same hierarchical structure of the process objects is maintained. Figures 2, 4 and 5 all represent different, but equally valid, ways of representing the same information.

```
[{
  "StdOut": [
    {"processes": [
      {"pid": 1, "name": "init"},
      {"pid": 2, "name": "bash"},
      {"pid": 3, "name": "ps"}
    ]}
  ],
  "StdErr": []
}]
```

Figure 4: Sample Output of ps with a Single Frame in One Array

```
[{
  "StdOut": [
    {"processes": [
      {"pid": 1, "name": "init"},
      {"pid": 2, "name": "bash"}
    ]}
  ],
  "StdErr": []
},
{
  "StdOut": [
    {"processes": [
      {"pid": 3, "name": "ps"}
    ]}
  ],
  "StdErr": []
}]
```

Figure 5: Sample Output of ps with Multiple Frames

2.3 Tributary

In order to manipulate the JSH format, we define a new utility which we call "tributary", which can be used to query substreams from the top-level standard output format. For example, running it with the query "stdout" is basically equivalent to "echo". But if we query for the "processes" subfield using the syntax "stdout>processes", then we can pull out all the values corresponding to the objects containing a "processes" key in StdOut. This essentially allows for the extraction of substreams from their parent Stream, allowing the user to manipulate the data into an expected format for other programs. However, using tributary is significantly easier than sed or awk, since no regular expressions need to be created.

Tributary Selectors

In order to manipulate a Stream, tributary accepts an argument called a selector. This selector tells tributary which Stream to promote. Selectors strings are surrounded by the "" symbol, and are broken up by the ">" symbol. Individual Stream names do not need to be surrounded by anything except the outer double quotation marks, unless there are spaces in the name. If there are spaces, they can either be escaped with a "\" character, or the whole Stream name can be surrounded by single quote marks. In order to use single quotes, double quotes, or backslashes in a selector, they must also be escaped.

Figure 6 shows an example of the output from Figure 2 piped to tributary. Here, we are using a selector to promote the contents of the processes substream to the top level, just below stdout.

```
$ ps | tributary "stdout>processes"
[{
  "StdOut": [
    {"pid": 1, "name": "init"},
    {"pid": 2, "name": "bash"},
    {"pid": 3, "name": "ps"}
  ],
  "StdErr": []
}]
```

Figure 6: Sample Output of ps Piped to Tributary

Some programs might have a substream that needs to be extracted in order to be

used properly by programs further down the pipe. In Figure 7, we show the output of an example ls program that does not follow our ordinary standards. Using tributary, that output can be manipulated into the format shown in Figure 8

```
$ nonstandard_ls
[{
  "StdOut": [
    {
      "foo": {
        "files": [{"name": ____}]
      }
    }
  ],
  "StdErr": []
}]
```

Figure 7: Sample Output of Nonstandard ls

```
[{
  "StdOut": [
    {
      "files": [{"name": ____}, ...]
    }
  ],
  "Stderr": []
}]
```

Figure 8: Expected Input Template for Standard Cat

Piping out the nonstandard format of ls through the `tributary` program would allow the user to normalize the data by simply typing "`nonstandard_ls | tributary "stdout>foo" | cat`."

Using Tributary to Standardize Output

While there is a tentatively defined format for all of our first party utilities, the expectation is that others will also use our output format when writing new utilities

and may not always follow the spec to the letter. If some third party application nests some kernel of standardized output within fields that are unknown to any of the standard utilities, using our tributary function will enable users to easily access that standard info, and essentially peel it out into its own Stream. [7]

2.4 Utilities Implemented

We redesigned and built a number of Unix programs around this principle. This list of programs includes:

- cat–concatenates multiple files specified with the command line arguments and output their contents in a JSON frame
- df–prints a listing of the amount of free disk space on the partitions currently mounted to the file system. Each file system structure contains a Name, Size, Used, Available and MountPoint.
- free–prints a listing of how much memory is available on the current system using a structure we define called MemStat which groups the Stat with the Units.
- ls–prints a listing of the current working directory contents, where each File is a (Name, size, Mode, Inode) and whose Permission nodes are (Read, Write, Execute).
- ps–prints information about the current list of processes including the User, Pid, PercentCpu, Percentmem, Vsz, Rss, Tty, Stat, Start, Time and Command.

2.5 Stream Library API Design

In this section, we will go over the design decisions we made for different Stream API libraries we made for some existing languages. These libraries allow users to output in the Stream format in several different languages. First, we talk about the general concepts that apply to all of the libraries, regardless of target language. Then we talk about the implementation for each chosen language, going over how the design was adapted to use the features of the language to their full benefit, while still maintaining compatibility with the existing Stream spec.

High Level Design

A user of the Streams library, in any language, goes through the following general steps to initialize the Stream and start sending data. First, the user initializes the Streams API. This exact process varies from language to language, but it generally involves setting up the file descriptors to use for representing stdin, stdout, and stderr, and telling the library to start outputting Stream frames. Therefore the library outputs the beginning array marker, "[", and then waits for output from the program. Users then obtain a reference to the Stream they want to output on. Users then call a method to give the chosen Stream the output value. The Streams API buffers a certain amount of data before sending it, to avoid constant writes. This value is configurable during setup, but by default it is 10 values. If the user wants to obtain a substream from the existing Stream, then they call a method, either on the Stream itself or passing that method a reference to the Stream, depending on the language. The requested substream is returned, which can then be used just like any other Stream. At the end of the program, the user calls a close method on the Streams Library, which flushes the remaining elements in the buffer and writes the closing "]" character, closing the top-level frame array.

Implementation: Go[5]

The Go API was the first implementation we created, and we designed most of the general structure as we implemented this API. We also used the API in our own utilities to make sure that it was usable, and it directly influenced the final design of the API. The Go API has a few modifications from the high level design. First, we use a Go thread to run the Stream buffer in the background, and we use channels to send new data to the Stream server, as well as to tell it when to shut down. This makes the code much more in keeping the traditional Go design, and helps simplify some of the issues surrounding synchronization, since Go channels can receive data from multiple endpoints in a thread-safe and synchronous manner.

Implementation: Python[3]

The original reason we selected JSON as our primary interface for the core utilities was that JSON parsing is available as a standard library feature for many other programming languages. Python is one of them. Unfortunately, Python's global interpret lock, which prevents multiple threads of Python code from being executed at the same time, makes it difficult to build an API that is thread-safe. However, we were able to build a `Stream` class whose methods implement the same general

interface as the high level API. Its methods include `start`, `stop`, `output`, `flush`, and `new_stream`.

Implementation: Ruby[4]

The Ruby API was built as an almost line-for-line port of the Python version.

Implementation: C[1]

Because C is not an object oriented language, we had to work a little bit differently. The C Stream API relies on an existing open source library called Jansson for parsing JSON. The C API's header file includes a collection of functions that are similar to the methods of a high-level `Stream` class: `jsh_new_stream`, `jsh_close_stream`, `jsh_start_stream`, `jsh_end_stream`, `jsh_new_substream`, `jsh_output`, and `jsh_flush`.

Implementation: Java[2]

Java was the only language we implemented the Streams API in that had both a strict typing system and generics, and we tried to use generics to their full power during implementation. This means that every Stream is generically typed, which adds very nice properties of type safety for output. However, to support this in Java, we needed to separate the concept of a terminal Stream, which is some raw value such as an integer or a float, and a Stream that is non-terminal, which can have substreams but no concrete value of its own. We also leveraged Java's built-in monitors and synchronization primitives to make the whole API thread-safe and concurrent.

2.6 Analysis of Design

This section weighs the benefits and drawbacks of our JSON-based design.

Strengths

The core utilities we built are much simpler and more flexible than traditional core utilities such as GNU coreutils and Busybox. In terms of number of lines of code, our programs are simpler since they interface directly with the kernel's 'proc' interface instead of relying on the C system calls API. The augmented core utilities do not require any formatting flags since formatting is simply pretty-printing the JSON,

which can be done with a number of separate programs. The Stream API implementations provide a practical means of building new JSH-compatible programs without having to worry about conforming to the JSON Stream standards we designed.

Weaknesses

As we designed and built the JSON utilities and API, it became increasingly difficult to come up with practical use cases for all of the new features we were implementing. For example, the Stream API works well, but we did not build any utilities in any of the programming languages we used that actually require the parallelization that the API offers.

Another issue with the Steam API is how clunky and difficult to use it is. One issue we have in all implementations at the moment is that the Stream must be closed explicitly by the user at the end of the program. This behavior is very different from how standard Unix printing currently works, where the expectation is that all output will properly be printed in a program that terminates successfully. Having to remember to terminate the Stream API is an error-prone task, as it will likely be forgotten. Additionally, all the setup required to print to a specific Stream is very tedious and space consuming: where previously a print operation could take one line, it now might take two or more depending on the language.

Finally, the JSON based protocol itself is confusing in the way that it is parsed. This is partially due our definition of Streams, but also because JSON is not a good format for this application. Some Streams are treated implicitly as arrays, where encountering more of their elements later in time just appends to an array which can lead to unintended lists of lists. Also, users of the protocol might want elements to be treated not as implicit arrays, but as singular objects. The JSON based protocol does not allow for this type of distinction, and that can make dealing with input difficult for both the user, and for the Streams API implementation itself.

2.7 Lessons Learned

In creating the JSON protocol, we drew a few key lessons from its strengths and weaknesses. First and foremost, we observed that raw text still has a place in an improved userland, especially when communicating terminal information to the user. The JSON format, while useful for retaining semantic information for programs, can obscure that information for users with its verbose syntax. A good example of this is output formatted as a table in traditional Unix utilities. This output has no easy translation into JSON, and the output certainly won't be as immediately obvious to the end user. Therefore, in our next implementation, we made sure to include the

ability to pass effectively raw text, and communicate information to the user in a more informative way.

Additionally, we learned a key lesson about imposing additional restrictions on top of existing data formats, such as JSON. We specified that a JSON Stream starts with the "[" character and ends with the "]" character. This means that the output of a program does not constitute a complete JSON value until the last "]" character is transmitted. This meant that we could not use existing JSON parsers for our project, as they need complete JSON. Rather, we had to fully recognize a frame before handing it off to an existing parser. We therefore knew that our next idea for an improved userland needed to use an entirely different method of data communication, and not an existing method of data serialization designed to be read in complete chunks.

3 Python Shell

In this section, we discuss our additional research and prototyping of our revised idea, the Python SHell (PSH). In Section 3.1, we discuss additional research we reviewed after evaluating the issues with JSH and the Streams protocol. Next, in Section 3.2, we outline the initial concept for our revised shell, how a shell utility might be written, and how shell utilities would talk to each other. We then describe our specific design for PSH in Section 3.3, including why Python was chosen to implement the concept, what issues with the initial concept prototyping exposed, and how we worked around them. Finally, in Section 3.4, we examine the usability of our prototype, and compare it to the usability of JSH.

3.1 Additional Research

After realizing some of the issues with our original JSON-based design, we decided to do more research on existing redesigned shell solutions. While there have been many different shell implementations since Bash was created, only a few have had both a radical redesign of communication between programs and also have wide use. We examine a few here, and point out the useful features that we wish to incorporate into our shell or the shortcomings of the solution.

PowerShell

PowerShell is the juggernaut in this realm of redesigned shells. It began shipping with operating systems starting with Windows 7.[14] This means that, as of April 2015, PowerShell is installed on over 70% of desktop computers.[8]

PowerShell was initially a Microsoft research project. In 2002, Jeffery Snover published a paper called the Monad Manifesto that described much of the initial vision of PowerShell, including high-level concepts, how utilities would be written, and how they would interface with each other.[13] The initial paper called out some of the same complaints that we have with the way that shells currently work. The paper specifically mentions *Prayer Based Parsing*, which is how it describes the communication methods between different programs. Specifically, the paper says "Because the executable outputs text, the downstream elements must use text manipulation utilities to try to get back to the original objects to do additional work... *Prayer based parsing* is when you parse the text and pray that you got it right."[13] The way that PowerShell seeks to fix this is by leveraging the power of the .NET run-time to pass objects between programs, instead of passing text. This allows programs to operate on the output of other programs using .NET Reflection APIs. Finally,

PowerShell provides additional support for parsing and validating user input and arguments to the program, and leverages this support to help programmers create documentation and make it accessible to users.

In PowerShell, an individual program is called a CmdLet (pronounced Command-Let). Programmers can write CmdLets in any language supported by the .NET run-time, which is usually C#, Visual C++, or Visual Basic. PowerShell also allows for the creation of scripts, which use a Domain Specific Language (DSL). CmdLets themselves can also be written in this language. All of these languages have the notion of an object which has fields inside it, and that allows them to call different methods on these child objects, as well as get any information they have in fields or properties without having to manual figure out the semantic meaning of text based on the surrounding text. An example of a PowerShell CmdLet, written in C#, is shown in Figure 9.

```
[CmdLet("Get", "EventLog")]
public class EventLogCmdLet : Cmdlet
{
   [Parameter(Position=0)]
   public string LogName = "system"; //Default to the system log

   Protected overrride void ProcessRecord()
   {
      WriteObject( new EventLog(LogName).Entries);
   }
}
```

Figure 9: C# PowerShell CmdLet Example[13]

Here, PowerShell leverages C# attributes to help the programmer read input from the command line. The first attribute, which is on the class, defines the name of the CmdLet, and when the user imports the CmdLet, PowerShell automatically defines the name Get-EventLog for the CmdLet. Next, the LogName class field has the Parameter attribute on it. This tells PowerShell that if the user supplies an argument on the command line, to set LogName equal to that value. It also has a default value of "system" if there is no argument supplied. The parameter can also be specified by using the -LogName flag, which PowerShell automatically defines from the field name and parses for the CmdLet. PowerShell provides additional attribute fields to specify things such as shell aliases, whether a parameter is manda-

tory, and valid input ranges and types. Finally, PowerShell uses the information gained from these attributes and C# documentation comments to generate help files for the CmdLet automatically, which can be found with the built-in CmdLet Get-Help. This is a significant improvement over traditional Unix programs, where each program is responsible for parsing all input, providing help prompts, and installing all documentation pages correctly. Further, PowerShell can use these attributes to provide auto-completion help on the command line, where traditional shells need custom extensions written for them such as bash-completion.

PowerShell also provides a set of functions that CmdLets override to allow for processing input. These commands have the following signatures:

```
protected virtual void BeginProcessing ()
protected virtual void ProcessRecord ()
protected virtual void EndProcessing ()
```

Figure 10: PowerShell Life-Cycle Methods

These methods and their uses are defined in more detail in the official PowerShell documentation, but to briefly summarize, they are called at different points in the life cycle of a CmdLet. BeginProcessing is called before any output from preceding programs has been processed. ProcessRecord is called every time a preceding program outputs an object. Finally, EndProcessing is called after the preceding program has terminated, and all of its output has been processed. This model is immensely powerful. Consider a program that operates on a list of files, performing a discreet action on each file. In a traditional Unix Shell, one might obtain that list of files by using the ls command. That list then has to be passed into the receiving program, which could happen in one of two ways. If the program accepts the list of files from stdin, it has to read in each file character by character, and handle doing so until stdin is closed. If it accepts the files as arguments, then another utility is needed, such as xargs. Either case needs to correctly handle escaping any spaces in file names, which could mean a trip through awk or sed. The PowerShell object model makes this easy: the Get-ChildItem command outputs each file separately. The receiving program sets a parameter to be received from the pipeline (done through attributes) and implements ProcessRecord. There is no need to worry about correctly escaping spaces in file names, since each record is a discrete file, and there is no need to worry about waiting until there is no more input. This is a very powerful model, and one that we are very interested in adopting for the second version of our shell.

PyShell and Scsh

While PowerShell was the main source of inspiration for our revised Unix Userland, we also took a look at other shells that entirely removed traditional Bash-style scripting and instead used an existing language. The first is PyShell, which is a shell interpreter written in Python.[11] This interpreter has some of the advantages of PowerShell: namely, it has powerful argument parsing abilities. However, PyShell still relies on traditional text as the input and output between programs, rather than enabling new commands to be written that allow for Python objects to be passed between them. Further, the manner that it does argument configuration is through an extra configuration file which must be specified in the program, meaning that every program has to have at least 2 files. Overall, while this does provide some nice enhancements, it ultimately fails to meet our original goal, which is to make passing information between programs easier.

Another shell is scsh (SCheme SHell).[12] This shell is based on the existing language Scheme. Scsh programs are written directly in Scheme, with bash syntax ported directly to Scheme in most cases. This means that all of the original disadvantages of bash still exist, including using text as a medium to pass between programs and no assistance in parsing arguments. Therefore, scsh does not meet our requirements.

3.2 Revised Shell Concept

In this section, we talk about the overall concepts for our revised shell. This includes our revised goals for writing programs, a general specification for how programs communicate with each other, and how output might be displayed to the user.

Our research into PowerShell was very promising, especially after the issues we had in JSH with cross-language compatibility. PowerShell is achievable because everything internally is .NET objects, and the .NET run-time was built in such a way as to allow objects from one language, such as Visual Basic, to be used by other .NET run-time languages, such as Visual C++ or C#. We therefore borrow several core concepts from PowerShell for our second implementation, which gives us a very powerful base on which to work. While we had not yet decided on an implementation language, we determined that we wanted scripts to all be written in the same language, so that we could directly use objects as the input and output of programs. We talk about language choice in Section 3.3.

Another feature that we want in our implementation is argument definitions abilities. Again, these would enable functionality similar to that found in PowerShell. Ideally, the shell would be able to pull the information about what parameters are

available for programs right out of the programs themselves, and be able to provide completion and help file generation for the programs, as well as a standard method of input validation. How exactly the shell does this is a matter that we leave for implementation, as each language will likely have a different method of annotating properties. C# has attributes, Java has annotations, Python has decorators, and so forth. Still other languages might not have direct annotation methods, so we might have to create configuration files or special methods to return this information.

Formatters

Additionally, we built on another concept in PowerShell, which are formatters. These are the programs responsible for taking the final output of a set of programs and formatting it for reading by the human running the programs. Default formatters and formatter options should be able to be specified on a per-program basis, as different programs will display better with different formatters, and will want to display certain fields by default. For example, a directory listing might be best displayed as a tree, whereas an echo would be best displayed simply as raw text. We consider the following formatters to be generally useful, and should be available in a full implementation. All formatter examples use the output of an ls command on a directory with 3 files.

Table Formatter A table formatter prints the fields of a series of objects in a table. The output from a table formatter would look like Figure 11.

```
Name        Permission   Owner   Group

file1       rw-rw----    fred    fred
file2       rw-rw-rw-    fred    wheel
executable  rwxrwx---    root    root
```

Figure 11: Table Formatter Example

List Formatter A list formatter prints the fields of each object it receives, one field/name pair per line, as shown in Figure 12.

Tree Formatter A tree formatter prints a recursive series of objects as a tree. This is more complex, and implies that the returned objects can have a hierarchy. We decided to leave this as an implementation detail for our prototype step, and

```
Name:        file1
Permissions: rw-rw----
Owner:       fred
Group:       fred

Name:        file2
Permissions: rw-rw-rw-
Owner:       fred
Group:       wheel

Name:        executable
Permissions: rwxrwx---
Owner:       root
Group:       root
```

Figure 12: List Formatter Example

to implement this functionality in the best way we saw fit in our final language or languages, rather than impose this constraint at the high-level design stage. Another interesting feature that the tree formatter might have is the ability to shrink and expand nodes in the console. This would allow for a full file-browser to be implemented in the terminal, complete with interactivity. However, while this is a nice thing to have, it is currently not a target feature for initial implementation. An example of a tree formatter is shown in Figure 13. This example assumes that the objects are in a folder called folder.

```
N: folder, P: rw-rw-rw-, O: fred, G: fred
|- N: file1, P: rw-rw----, O: fred, G: fred
|- N: file2, P: rw-rw-rw-, O: fred, G: wheel
|- N: executable, P: rwxrwx---, O: root, P: root
```

Figure 13: Tree Formatter Example

There is certainly room for other formatters, and they should also be user-definable. One can conceive of a formatter that outputs to JSON, for example, or a formatter that outputs to a CSV. However, we believe that the list, table, and tree formatters are the basic formatters that should go into a basic implementation of our new shell design.

3.3 PSH Design

In this section, we talk about the implementation details of our revised shell, which we are calling Python Shell, or PSH. This differs significantly from PyShell, which we mentioned in Section 3.1. We will go over the reasons we chose to use Python, how we implemented the object pipeline, our proof-of-concept formatters, and a few simple Unix utilities that we re-implemented in PSH as proof-of-concepts.

Language Choice

We looked at a few languages for the implementation of our redesigned shell, and settled on Python for a few reasons. We first considered using Go again, but we decided against it for a few reasons. One very important part of our redesigned shell is that the language needed to have either a strong reflection system or a dynamic typing system that allows for easy access to arbitrarily named fields or methods in an object. A strong reflection system allows for inspection of objects, and determining of their types and fields. For example, inspecting a file object would likely show fields such as name, location, permissions, and owner. A dynamic typing system allows for treating an object as any type we choose and allowing us to call methods on the object, failing at run-time if the methods are not present. These errors can be caught by the program and useful feedback can be given to the user. We used Go's reflect package during our implementation of the JSH API and found it to be clunky to use, which was discouraging for a shell where we pass around objects. We also needed a language that would allow for rapid prototyping as we started redesigning our project towards the end of the MQP. It's possible that it might be usable for the final implementation, but we discovered several advantages of Python during implementation that lend themselves towards making PSH powerful and easy to use.

Since we decided to not use Go, we looked at several other languages. We mostly focused on scripting languages with large ecosystems, such as Ruby or Python. We specifically wanted to make sure that the language we chose would have lots of preexisting libraries that we could use to simplify development, as we only had a few weeks to do the redesign. We ended up settling on Python over Ruby due to group experience with Python, as all members of the group have had at least a decent amount of experience with Python. Some members of the group had never used Ruby before, which would not prove conducive to rapid development.

Shell Program Organization

Shell programs in PSH can be one of two different types of programs: either generators or formatters. We'll go into more detail on each of these individual types later, but for the moment, we'll talk about the overall concepts that tie them together. Generators generate output for programs further down the line, and are so named because they are literally Python generators. Python generators are lazily evaluated iterators that make use of Python's "yield return" features. This allows a Python function to resume execution where exactly where it stopped during a previous call. This type of function automatically returns an iterator, which iterates over these objects returned in succession. Similar functionality is available in other languages, though terminology and syntax vary. Formatters take the output from generators and transform it into human-readable text. Generally, per command executed on the console, there will be at least one, possibly more, generators, and there will be one formatter at the end. Additionally, the shell will add the default formatter to the command if the command does not already have a formatter. If a PSH program is piped to a legacy program, then a formatter is added between the two programs if there is not already one.

PSH programs are created by inheriting from one class: BaseCommand. This includes both formatters and generators; the type of the program is specified by whether it returns an output generator or not. Generators do return this, and formatters do not. The general process for running commands is as follows: the programmer enters commands on the command line. The syntax for this is a very limited subset of bash: the "|" character is used for piping output between programs, and the ">"/">>" characters support writing to files in either non-append or append mode, just like bash. However, that is it for basic command line functionality. More advanced scripts can be entered directly as Python by using the ">" character as the first character of a command. If this is done, all following input on that line will be interpreted as Python.

If the command entered is not Python, we first attempt to match the individual programs being called to registered shell commands. If this succeeds, we set up the outputs to chain from one program to the next, terminating in a formatter. Again, if there is no final formatter, the default is appended here. If there are traditional Unix commands, they are run by opening them as sub-processes.

PSH structures output by having all programs return a generator of TreeNode objects. Again, TreeNodes are explained in more detail below, but in a nutshell, they are N-ary trees, where each node has some data value, and some number of child generators, potentially 0.

Generators

A generator is roughly analogous to an individual legacy program such as ps, ls, or free. They generate some data in the form of TreeNodes. However, sometimes things are different: there are no formatting flags on a generator command. There are still flags that control the type of information that is output: in the "free" utility, for example, there will still be flags for controlling what memory information is output, such as page count, virtual memory used, etc. However, the actual formatting of that information, be it in a tree, a list, or a table, is decided by the specific *formatter* used, not by free itself. Some programs will likely need to have a few flags. ps, for example, would have to change the output format if it is going to be displayed as a tree instead of as a table. If the final output is going to display as a tree, then all processes need to be be output as a true hierarchical tree, whereas if the output will be displayed as a table, then all processes need to be output at the same tree depth. For the most part, however, this separation of formatters allows individual programs to remove their many, many formatting flags, and leave the complicated layouts to the final formatter. Additionally, since a TreeNode retains semantic information, much of the original need for having different formats that are easily parsable by sed and awk is eliminated, since the final text is only being used by humans and not by other programs.

TreeNodes

Non-hierarchical data, such as lists and sets, can be represented easily with a hierarchical structure like a tree. Because trees are incredibly common in computer systems; file systems, "process trees", etc., and are also capable of representing other common structures such as lists and sets, the atom of object based output was made to be a tree. The actual TreeNode object is very simple, functioning as a container for the data at the current level, as well as producing an iterator over its child nodes.

```
class TreeNode(object):
        """A tree node holds some data and has iterable children."""
        def __init__(self, data):
                """Initializes a data. The data can be any type,
                but it usually is an ordered dictionary."""
                self.parent = None
                self.data = data
                self.children = []
        def add_child(self, child):
                """Adds a child to the node."""
                child.parent = self
        self.children.apppend(child)
        def children(self):
                """Returns an iterable generator for all the children
                nodes."""
                def children_generator():
                        for child in self.children:
                                yield child
                return children_generator()
```

Figure 14: The entire TreeNode class.

As can be seen with a glance at Figure 14, the TreeNode is incredibly simple in design, essentially providing nothing more than a wrapper for a generator of child nodes and a container for arbitrary *stuff*. The data field in the TreeNode class holds either a string or list of values, generally just strings themselves.

Formatters

In PSH, formatters are special applications that are designed to take TreeNodes and print their contents to stdout. With the change to using internal data objects rather than text for piping data from one application to the other, there arises a need to display this data to the user in a comprehensible way, which is exactly what formatters do. The idea behind having multiple individual applications, decoupled from the text they are formatting, to do the formatting is to allow for greater flexibility in how the data is formatted, without needing any increase in complexity of the code of any application. We detailed some of the specific formatters we feel are needed for a base system in Section 3.2.

3.4 Usability of PSH

In this section, we take a look at the usability of PSH, starting first with a quick description of our prototype environment, then walking through an exemplar program, "echo", and finally comparing PSH to the JSON and Streams API we introduced in Section 2.

Prototype Environment

To test the PSH environment, we created a small implementation of the PSH concept, ignoring a few of the more powerful features and not focusing on the security aspects of a shell. We implemented generators, TreeNodes, and basic formatters, allowing us to test this functionality. An important note is that this shell is not production ready, as the Python programs do not have process isolation, and can interfere with each other's memory and global variables, either unintentionally or maliciously. However, it did serve our needs for determine the general usability of the concept.

Exemplar PSH Program: Echo

We re-implemented a popular Unix utility called echo in PSH as one of our first programs, to see how easy our framework is to use and play around with. Echo is a very simple utility that prints any command line arguments it receives and any data it receives on stdin back to the console, hence the name echo. This can be useful for sending messages to the user inside bash programs, and putting text into a file by redirecting the output of the echo command into a file. While we expect the former use to be effectively useless in PSH, since Python has existing methods for printing to the user, the latter use case is still relevant, so we re-implement the program here. The full program can be seen in Figure 15.

This program is very simple and easy to understand, so we'll briefly step through it. Echo is based on the BaseCommand class, which defines the general methods and helper methods for a PSH command. It also calls the register_cmd() decorator, which registers the echo command with the main PSH shell on load, with the name "echo". Next, we define a constructor that takes a list of arguments, with a default of no arguments, and we store the arguments for later use. The next method is the call method. This is the method called by the shell when the user uses the echo command, and all commands must provide an implementation of the method. The call method returns a generator of TreeNodes, which is passed to the next program. In our simple echo implementation, it is very easy to do this. First, we yield all of the arguments given to echo. Next, we check the input generator. Getting the input generator is

```
from psh.commands.core import BaseCommand, register_cmd

from psh.tree import TreeNode


@register_cmd("echo")
class Echo(BaseCommand):
    """Echoes anything from the command line arguments as well as input
    from the previous command."""

    def __init__(self, args=[]):
        super(Echo, self).__init__()
        self.args = args

    def call(self,*args,**kwargs):
        # Get the output generator from the previous command. This is our input
        input_generator = self.get_input_generator()
        def output_generator():
            # First, echo everything in our arguments
            for args in self.args:
                yield TreeNode(args.encode("utf-8"))
            # If there was no previous function, this will be an empty iterator,
            # and will never loop. Otherwise, echo out everything from the previous
            # program call
            for node in input_generator:
                line = node.data
                yield TreeNode(line)
        return output_generator
```

Figure 15: The PSH Implementation of Echo

handled by BaseCommand class, which exposes the helper get_input_generator() that returns the generator if there is one, and the empty list if there is not. Finally, this generator is returned, and is evaluated as needed by the next program in the list.

PSH Strengths

In working with our test implementation of PSH, several strengths of the design have stood out, and some very nice benefits that we didn't originally intend, but are nice to have regardless. First, it is very easy to work with input piped from other programs. The ability to work with raw objects making calling methods or getting different information much easier than a traditional program written for Bash or another shell. The programs, as shown by the echo program in Figure 15, can be very short, and don't have to do much work to get input from another program. Formatters are also very easy to implement, as they just need to get the TreeNodes and print the data contained inside. Since the data inside a TreeNode is a Python object, it has named fields, which makes printing the information very easy.

Another benefit that we did not design for, but is very powerful now that we have it, is lazy evaluation of inputs. Because programs return generators, and in Python generators are only called when another object is needed from them, programs have to do only the minimum required amount of work. Take ls, for example. If we ls a directory, the returned generator will yield a list of TreeNodes. Each node's data will correspond to one entry in the current directory, which is either a file or a folder. If the entry is a file, the list of child generators will be empty. If the entry is a folder, then it will have one child generator that will generate the listing for that directory. If we take the output of the ls command and format it with a tree formatter, the tree formatter will walk the entire TreeNode structure, including children nodes. However, if we format it as a table, the children nodes will never be called, meaning that the ls program will not have to generate directory listings for the sub-directory. While this is a trivial example, more complex operations that take a few seconds might be piped to a search function. If the search finds what it is looking for in the first yielded TreeNode and terminates, that could potentially save a lot of time on the part of the programmer, without having to do any explicit design work to gain this boost.

When compared with our JSON and Streams API, we believe that PSH is significantly easier to comprehend and use. PSH requires no explicit setup or tear down methods like the Streams API does, as it does not have to worry about terminating frames or clearing buffers. Further, the concept of an object is much less foreign to programmers today. According to the TIOBE Index, a widely-used listing of the most popular programming languages in use, 9 of the top 10 languages at least support the concept of an object, if not being directly object-oriented.[15] Here at WPI, objects are introduced early as part of the undergraduate curriculum. Python is taught directly to students in CS 1004: Intro to Programming for Non-Majors, and Java is taught in CS 2102: Object-Oriented Design Concepts. Since users already have

familiarity with the concept, PSH should be easier for them to pick up, as opposed to Streams, which is far more complicated. Additionally, since we are not imposing new syntax on top of an existing serialization methodology, implementation of PSH is significantly easier than the Streams API, and even if we have to serialize Python objects in the future, the pickle library, part of the Python language, is already very robust and well tested, and should need no additional modifications from us to use.

PSH Weaknesses

Even though PSH is much easier than the JSON and Streams API, there is still some overhead in usage. This is having to define a class that extends from a specific other class, and then define a method that returns a generator, and finally design the code to be used as a generator. There is the additional pain point that programs currently must be written in Python, or a language that can output Python objects in a Python generator. It is likely that programs can be written in C with the C Python API, but we have not tested this, and the C Python API can add significant overhead to a C program in terms of structural complexity. There are also bindings available for other languages, and proxy tricks can be used for languages that can call C. For example, Java can call into native code, usually C, using Java Native Interface (JNI). However, JNI itself adds a lot of complexity to a program, and now you need two binding libraries just to talk to the shell. While PowerShell can somewhat get around this by having multiple languages support the same binary object format, we need to explore other options for cross-language usage in the future.

4 Evaluation

In this section, we discuss our evaluation of the two designs. After completing our implementation, we conducted a user study for both concepts with the express goal of getting some initial feedback on the high level concepts of our designs.

4.1 Procedure

We conducted an informal user study among fellow computer science students with varying levels of experience with shells and command line interfaces. All of these participants have at least some familiarity with a Unix terminal. We allowed the participants to dictate their responses, and we recorded their answers in a text file. To mitigate bias, we flipped a coin to determine whether we would explain and ask questions about JSH (3-6) or PSH (7-10) first. In total, we recorded responses from 8 different students.

The questions were as follows:

1. Do you find awk, sed, and grep cumbersome? What do you like or not like about them?
2. Are you familiar with PowerShell?
   - 2a. (Follow up: if they say yes) Have you ever used cmdlets?
   - 2b. (Follow up: if they say no) (Say: PowerShell is a new shell implementation by Microsoft, shipping with Windows 7 and up. It allows for sending .NET objects between programs instead of raw text, and programs can be written in C#, Visual Basic, Visual C++, and a custom scripting langauge very close to bash-syntax.) Does this sound useful?
3. Do you think this would be convenient to work with from the command line?
4. Do you think this would be useful as a tool to make shell scripting easier?
5. Would you prefer to work with traditional bash utilities or JSH, assuming that all utilities you would want to use are JSH-compatible?
6. Are there any features that you see missing from JSH that would make it more usable?
7. Do you think this would be convenient to work with from the command line?

8. Do you think this would be useful as a tool to make shell scripting easier?
9. Would you prefer to work with traditional bash utilities or PSH, assuming that all utilities you would want are PSH-compatible?
10. Are there any features that you see missing from PSH that would make it more usable?
11. Of the two concepts, JSH and PSH, which would you prefer to use?

4.2 Results

Our results come in two parts: quantitative and qualitative. The quantitative responses refer to questions whose answers can fall under discrete categories such as "yes", "no", or "maybe". The only questions we asked which merit qualitative responses inquired about suggestions.

Quantitative Responses

For some of the questions, a few participants gave more nuanced responses rather than definitive a 'yes' or 'no'. Only definitive 'yes's and 'no's counted for the purpose of our tallies. Anything else was counted as 'maybe', with common concerns addressed later in Section 4.2. The answers to these 'yes' or 'no' questions are summarized in Table 1.

Question	# Yes	# Maybe	# No
1	3	4	1
2	4	1	3
2a	0	1	3
2b	4	0	0
3	0	6	2
4	6	1	1
7	6	2	0
8	7	0	1

Table 1: Survey Results–Yes/No Questions

For comparison questions between traditional Unix userland tools and the designs as we explained them, we asked for a more definitive answer: ours, traditional or no preference. We asked the same for the final question about whether they prefer JSH or PSH. The answers to these comparative questions are summarized in table 2.

Question	Ours	Traditional	No Preference
5	4	3	1
9	6	1	1
Question	PSH	JSH	No Preference
11	2	5	1

Table 2: Survey Results–Comparative Question Response Counts

Qualitative Responses

When asked for suggestions to improve the usability of the JSON design, most participants offered no immediate suggestions. Five of the eight participants expressed the desire for traditional or "normal"-looking output as the user interface in JSH. These participants did not like the idea of having to read JSON as the final output. Of these people, one participant would have liked it implemented as "fallback mode" whereas the other four wanted it it as the default implementation.

Most participants had no suggestions to improve the usability of PSH. Among the three people who did offer ideas, the most requested features include support for traditional shell features such as environment variables, customizable prompts and background jobs.

4.3 Analysis and Summary

We learn from the responses to question 2 that participants who are not familiar with PowerShell think that the concept of passing objects between programs via the shell is useful. Four of the four participants who responded "no" to question 2 answered "yes" when asked question 2b. We also learn that participants who claim familiarity with Windows PowerShell have not heard of the concepts of "cmdlets". Three of the four participants who responded "yes" to question 2 answered "no", and the last participant responded with "a little" to question 2a.

The results of question 3 indicate somewhat negative opinions against the usability of JSH as a command line interface. Two participants responded with definitive 'no's, and the six others had expressed feelings. A few participants expressed 'yes' with the condition that some sort of formatter program would convert the JSON into traditional, human-readable text.

Question 4's answers show that most participants feel JSON serialization is a useful technique for improving shell scripts. Six of the eight participants responded with a definitive 'yes'. One responded with 'no', and the other said it "certainly could be" useful.

Participants had mixed opinions over whether they would use JSH over traditional userland tools; specifically, four participants expressed preference for JSH, three explicitly prefer the traditional userland, and one indicated no preference.

Six of the eight participants indicated that PSH would be useful as a command line interface. However, two participants expressed concern over the complexity of defining new functions and commands in the command line interface. However, seven of the eight participants indicate in their responses to question 7 that PSH would be useful for shell scripting. The one dissenter believes that PSH's scripting functionality "feels like another library" and that "there [are] already ways to do this in Python".

In Question 9, we find that six out of the eight participants would prefer to use PSH over traditional userland tools. One participant indicated no preference, and the last would rather use traditional tools.

Question 11 shows that five of the participants would prefer PSH over JSH, while two would prefer JSH. One participant was unsure.

The features suggested in Section 4.2 for PSH (such as background jobs) were outside the scope of our project, but we consider them necessary for PSH to be used as a practical shell. Consequently, we will address these concerns along with other ideas for potential future work in Section 5.

5 Future Work

In this section, we will discuss potential future areas of exploration for both JSH and PSH. We will start by going over what might be done to improve the usability of JSH, and then talk about what still needs to be explored and proven for PSH to be considered a complete solution.

5.1 JSH Future Work

The biggest change that needs to occur for JSH is for it to no longer be based on strict JSON. One potential direction is likely creating a domain specific language (DSL) that has explicit syntax differentiating a stream from an object. This will help solve one of the bigger issues we had when working with the Streams API, which was the difficulty in creating a parser that understood the difference between a stream and an object. Creating a DSL will inherently require creating a grammar for the language in the parser, which can be formalized and published. This direction will need some additional work in defining an API that both powerful and easier to use than the existing JSH API, and it might require some significant redesign of how streams are transmitted between programs.

Another direction for JSH is to drop the idea of streams from the protocol, and instead use JSON or some other data serialization form such as XML to communicate between different programs. This model would much more closely resemble PSH than JSH, but it would have the advantage of being able to use many different languages. It would have a few disadvantages compared to PSH: because Python generators are lazily evaluated, they can do less work when possible. If programs written in multiple languages wanted to do this, then they would have to have bidirectional communication, which might prove difficult.

Either of these solutions is going to need more utilities to prove useful. They would also both likely benefit from having a full shell replacement that can detect when to use the chosen communication protocol and when to use raw text. That shell should also be able to convert the output back to human-readable text at the end, rather than having to pipe all output to a formatting program. While we envision this shell being mostly bash-compatible, it is possible that this shell will have extensions to the bash syntax to support direct manipulation of whatever protocol is chosen for communication.

Finally, both of these utilities will need to have their support libraries updated for all languages. These need to be at least partially redesigned to allow for one-line output of the chosen protocol, without having to worry about set up before use and

close after use. This requirement, as it currently stands, adds a lot of overhead to what a programmer needs to know about before using the protocols, and is generally nonintuitive. If we expect this idea to get support in existing markets, it has to be easier to use than it currently is.

5.2 PSH Future Work

PSH needs quite a bit more work before it can be used as a practical replacement for standard shells. In the limited time we had to prototype, we created a good proof of concept, but it is missing several features and has not been designed in such a way as to make extending and maintaining it easy. It currently has GNU readline support, so past commands can be recalled, but it needs other standard shell features before we can expect it to be used to replace bash or zsh. One simple example is environment variable support. This is likely as simple as creating and assigning Python variables for PSH programs, but we need to be able to set environment variables for when we call legacy programs. The shell also needs to have more configurability, in the same way that bash and zsh do. One possibility for this would the .pyrc configuration file and the .pyrc.d/ directory. The .pyrc file would be a standard Python file, and by calling methods or setting global variables various shell settings can be changed. For example, if the shell supports having a custom prompt, we could define a special function name that the user could create in the .pyrc file. If the function is defined, it is used to create the prompt. Otherwise, the shell uses a default prompt. The .pyrc.d/ directory could be used to store more startup scripts, which are referenced by .pyrc, and also to store user-specific PSH scripts that should be loaded by the shell.

Another big feature needed for PSH is process isolation. Currently, all scripts are run in a single process, which is very bad for process isolation. This allows scripts to assign to global variables and conflict with each other, and opens numerous security holes, not least of which is if a user wants to run a PSH program as root, then they have to run the entire shell as root, and any other program that they happen to run in that session will run as root as well. We believe that this issue should be addressable within the shell relatively easily, using existing interprocess communications (IPC) techniques such as shared memory and IPC semaphores. If implemented correctly, these methods should still be able to give us the power of lazily-evaluated generators, while also giving programs the ability to radically customize the shell session that they are working in if they need to.

Not only are there features still needed in the shell, but there are also new applications to complement the shell that would provide additional utility. A network

sniffer would be incredibly useful as a command line application. Most packet sniffers are packaged into large monolithic applications where storage, filtering, and interpretation are all performed in one place. An application that simply sniffed and passed all of the traffic seen to whatever search and formatting programs are desired would be far more in line with the Unix philosophy.

5.3 Improvements to Evaluation

Because our initial survey was not sufficiently rigorous, we cannot draw any objective conclusions from our results. Once the shells are more feature-complete, a more concrete, hands-on user study would serve well to demonstrate the efficacy of our work. In this user study, users would be assigned simple tasks in both environments and asked questions about their experience with both. These tasks could range from simply launching programs or parsing text to writing a full on program using the APIs we built. Quantitative metrics such as "time needed to implement ls in jsh vs. psh" would provide even stronger comparative evidence.

The opinion study conducted in this paper could be easily reproduced with a more representative sample. If an explanation of the two designs were made available online in either video or text format, the survey could be conducted to represent a population much closer to our target audience: all Unix-like command line interface users.

6 Conclusion

This project originally set out to create a set of core utilities and a new shell in order to modernize the Unix philosophy, specifically using Go as a language and JSON as a data format. We encountered many problems with this initial approach were uncovered, and the initial approach ended up serving more as a study in what does not work, as opposed to being a fully fledged prototype.

The shortcomings of the JSON based protocol did end up helping to steer the project towards looking at what PowerShell had done right, and thus proved very useful in shaping the end result of this project. PowerShell ended up lending some major concepts to our shell, such as the concept of a formatter and the concept of passing objects without any serialization whatsoever.

The resulting Python based shell, called PSH, proved to be much easier to work with both from an end user standpoint as well as a utility developer standpoint. The power of Python based scripting meant that all shell scripts could be written in a language that most users would already be familiar with, which should facilitate ease of adoption.

User testing ended up being brief due to time constraints, and was over a small enough sample size as to be inconclusive, but the initial results seemed to favor PSH over JSH and Bash. So while no hard conclusions can be drawn, PSH shows promise and merits further development.

7 References

[1] C (programming language). http://en.wikipedia.org/wiki/C_(programming_language), 2015.

[2] Java. https://www.oracle.com/java/index.html, 2015.

[3] Python. https://www.python.org/, 2015.

[4] Ruby. https://www.ruby-lang.org/en/, 2015.

[5] The Go Programming Language. http://golang.org/, 2015.

[6] Scott Bradner. Key words for use in rfcs to indicate requirement levels. BCP 14, RFC Editor, March 1997. http://www.rfc-editor.org/rfc/rfc2119.txt.

[7] T. Bray. The javascript object notation (json) data interchange format. RFC 7159, RFC Editor, March 2014. http://www.rfc-editor.org/rfc/rfc7159.txt.

[8] NetMarketShare. Desktop operating system market share. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpsp=2015&qpnp=1&qptimeframe=Y, 2005.

[9] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[10] Eric S. Raymond. Userland. http://www.catb.org/jargon/html/U/userland.html, 2010.

[11] Alex Rudy. Pyshell. https://github.com/alexrudy/pyshell, 2015.

[12] Olin Shivers, Brian Carlstrom, Martin Gasbirchler, and Mike Sperber. Scsh. http://scsh.net/about/who.html, 2006.

[13] Jeffery Snover. Monad manifesto. *Aug*, 22:1–16, 2002. http://blogs.msdn.com/b/powershell/archive/2007/03/19/monad-manifesto-the-origin-of-windows-powershell.aspx.

[14] Jeffery Snover. NY Times Delcares [sic] PowerShell to be 30% of the value of Windows 7 :-). http://blogs.msdn.com/b/powershell/archive/2008/10/29/ny-times-delcares-powershell-to-be-30-of-the-value-of-windows-7.aspx, 2008.

[15] TIOBE Software. Tiobe programming community index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2015.

Appendices

A Building JSH

A.1 Building The Core Utils

The core utilities of JSH are coded in go, and thus the go environment needs to be set up first. Instructions on how to set up a go coding environment can be found here: http://golang.org/doc/install. Make sure to follow the instructions for installing go AND setting up your environment: this Appendix assumes that you have followed both.

After installing go and setting up your environment, put your copy of the JSH core utilities source in `$GOHOME/src`. From there, installing any of the utilities is straightforward. The `go install <utility name>` command will install the utilities to `$GOPATH/bin`, and they can be executed from there. Use the –json flag in order to use the new output format: we pass through to the legacy implementation by default.

The following is a list of the utilities we implemented in this project. These are also the utility names used in the above command for install the utilities.

- `cat`
- `df`
- `free`
- `ps`
- `ls`

A.2 APIs

Go

In order to use the Go API, follow the instructions in appendix A.1 to set up go and install the JSH libraries. In your code, import the `jsh` namespace. References for how to use the go API can be found the implementations of the other core utilities.

Python

The Python API is made available as a setuptools package. To install it with pip, clone the repository and run `cd python; pip install .`. Example usage in available in `example.py`.

Ruby

The Ruby API is an almost line-for-line port of the Python API. To use it, copy `jsh.rb` into a directory on your `$LOAD_PATH`. Example usage is available in `example.rb`.

C

The C API is distributed in the form of a shared object file (library). The library requires `libjansson` to be installed as well as a C99-compatible compiler such as GCC. A `Makefile` is provided under the `c` directory of the API repository. After running `make`, copy `build/libjsh.so` into a directory within `LD_LIBRARY_PATH` and `jsh.h` into `/usr/include`. Sample usage is provided in the test module `libjshtest.c`. To run the tests, `cd build; LD_LIBRARY_PATH=. ./libjshtest`.

Java

The Java API is relatively easy to use. First, install either OpenJDK 8 or Oracle JDK 8. In the Java API directory, execute `./gradlew install`. From there, the Java API can be used as a Maven dependency from the local Maven repository. The group id is `jsh`, the artifact id is `java-api`, and the version is `1.0-SNAPSHOT`.

B Building PSH

B.1 Requirements

PSH requires two main components:

- Python 3
- pip

B.2 Installing

After cloning the `psh` repository, simply run the following: `pip install -r requirements.txt; pip install .`

B.3 Running

To run psh, simply execute `python -m psh.run`.