Finally an Interactive Shell

Nicole D. Terry and James S. Jennings
Department of Electrical Engineering and Computer Science
Tulane University, New Orleans, Louisiana, 70118
{terry | jennings}@eecs.tulane.edu

Abstract

The goal of this work is to support the development of more intelligent and more interactive text–based interfaces for general computer use by system administrators, programmers, students, professors, and so–called "power–users". In today's networked environment we are faced with an ever–increasing level of complexity. Our departmental network alone provides public access to 250,000 files, 1,700 programs, and 22 workstations. When you add the Internet as a source of additional data, programs, and hosts for remote computation, these numbers grow exponentially. Automation is the key to dealing with this complexity and users know how to automate their tasks. Unfortunately, current shells present many challenges to users who wish to incorporate more automation into their interface. We find that a new command shell architecture is needed.

Finally an Interactive Shell (a.k.a Fish) is a testbed for techniques to support the next generation of truly interactive text–based interfaces. It is a fully customizable and extensible shell. The architecture of Fish is a radical departure from that of traditional shells. Perl was chosen as both the implementation language and Interface Programming Language *of Fish for many reasons, most notably to allow rapid prototyping of new techniques and ideas.*

1 Introduction

Users of current text–based interfaces, namely Unix shells, do entirely too much typing and too much searching. They have to be very specific in order to get the machine to do what they want. Figure 1 presents examples of common frustrations. These problems can be alleviated in two ways:

- Enable users to provide more context to their shell. This will reduce search spaces and make interaction more concise.
- Give users a full programming language and principled access to shell internals so that they may customize and extend their shell. This will allow users to automate their tasks.

Unfortunately, we cannot re–work a traditional shell, such as the Bourne shell or its derivatives, to achieve our goals for two main reasons:

- Concurrently running shells are separate entities and do not communicate with each other. *For example: If you define a new alias, variable, or shell function at the prompt, it is not available in your other concurrently running shells.*
- Their behavior can only be changed by the user in a few predetermined ways. *For example: You can change the value of the* PATH *variable but you can't change how *.h is expanded. What if you wanted filenames considered in not only the current working directory but also /usr/include and /usr/local/include?*

Therefore, we find that a new command shell architecture is needed.

2 Proposed Solution

Our approach is to centralize knowledge (context) in a persistent way while parallelizing execution. The proposed solution consists of:

- A centralized knowledge repository in which data and procedures are subdivided in a hierarchical fashion and automatically stored to provide persistence.

Problems With Traditional Text–Based User Interfaces	**Current Engineering Solutions**	**Reasons that Current Solutions Do Not Scale**
Too much searching	globbing, path, tab completion	How does the system know what the user wants?
Example: On the local network the command `java` *exists in nine different directories. Which* `java` *I want depends upon what version I need and whether I want to use an integrated development environment. How does the interface know which implementation I want when I issue the command?*		
Too much typing	aliases, variables, functions, scripts, etc...	With only one namespace for everything, confusion occurs when there are many of each type of abbreviation.
Example: I have three print aliases that I use depending upon where I am and/or what I am doing. If I have not used them for a while I get them confused and have to look at their definitions.		
Programming the shell is hard	better scripting languages provided by *rc*[Duf90], *es*[HR93], and *scsh*[Shi94]	There is no persistent memory except for file system. Therefore delayed or remote computation cannot be done in a principled way since dependencies are not maintained (i.e. the total context at the time the computation is issued is not the context when the computation is executed).
Example: I write a `cron` *job in Perl in my current shell and when I test it, it works fine. The current setting of my PATH variable finds* `perl` *in* `/usr/local/opt/perl5.005/bin/perl`*. When my* `cron` *job runs at 3am it regenerates the value of my PATH variable from my .cshrc file which yields* `/usr/local/bin/perl` *instead which is a different version and may have unexpected results.*		
No easy access to previous results	file redirection, pipelines, command substitution	Current techniques to capture results require forethought on the part of the user. If the user runs a command without capturing its results and then realizes at a later time that she wants to use those results, she must re–execute the command to regenerate the results.
Example: Sometimes when I run the command `find`*, there are so many error messages interleaved with the results of the command sent to standard output that the window buffer overflows and I cannot scroll back far enough to see the initial results. I must then re–run this time consuming command and redirect standard output to a file in order to see the results.*		

Figure 1: Problems With Traditional Text–Based User Interfaces

- The latest version of Perl provides threads.
- Perl facilitates rapid prototyping.
  - Perl is regarded as easy to use for many reasons, most notably that it provides automatic memory allocation, garbage collection, and data typing.
  - Perl users have written hundreds of modules and made them publicly accessible which facilitates "plug and play" prototyping.
- Due to the nature of shell usage, an interpreted language is appropriate.
- Procedural abstraction, modularity, and exception handling are supported.
- Perl is familiar to a large portion of the Unix community.

Figure 2: Advantages to Using Perl

- Perl is slower than compiled languages such as C.
- Threading is new to Perl, therefore:
  - Most modules have not been re–written so as to be thread safe.
  - The killing of running threads is not yet supported.
  - It is recommended that no production code include Perl threads because race conditions still exist in the implementation.
- There is no consistency or quality guarantee for modules.

Figure 3: Disadvantages to Using Perl

- An *Interface Programming Language* which can be used to customize and extend the interface.
- An architecture in which the shell is a long–lived, multi–threaded server that supports concurrent requests from thin clients.

Our prototype is named **F***inally an* **I***nteractive* **SH***ell*, or simply *Fish*. It was written entirely in Perl version 5.00503 with threading enabled. It has been successfully tested on Linux and Solaris platforms. Figures 2 and 3 outline the advantages and disadvantages to using this language. Despite the disadvantages of Perl, it facilitates rapid prototyping which was critical in this work. *Fish* is meant to serve as a test bed for techniques to improve text–based interfaces. A crucial quality of any test bed is the ability to rapidly develop and try new ideas.

2.1 Persistent Knowledge

In *Fish*, *context* consists of command aliases, variables, functions, command history and *result history*. The result history is similar to the command history in traditional shells except that it contains the text sent to standard output and standard error by commands rather than the text of the invoking command. This enables results of previous commands to be available for filtering and/or processing in the future. *Fish* allows this context to accumulate over time. The user and his agents create new context by running programs, defining new functions, etc. After that, *Fish* makes sure the context endures. Such knowledge lives in the server so that it is centralized and, unless explicitly removed, will persist. To ensure that it *outlives* the server process, it is checkpointed to the file system periodically in order to recover from power outages and reboots. The Storable module is used to provide this persistence.

Just as subdirectories help us deal with the complexity of the file system, subdivision of shell context can also help. Not all context is relevant all of the time, it is task–dependent. To organize a large amount of context accumulated over time, *Fish* maintains a hierarchy of *project scopes*. A project scope is a namespace used to store related items. Users create and delete project scopes, and can switch at any time from one project scope to another, much as they change directories now. Each project scope has its own context. The related items that they contain comprise the context shared between the user and the shell that pertains to a particular project or task. *Fish* users create their own project scope hierarchy to suit their needs just as they create their own subdirectory tree in the file system. Currently, only single inheritance is supported in the design. Multiple inheritance might in fact be more appropriate here.

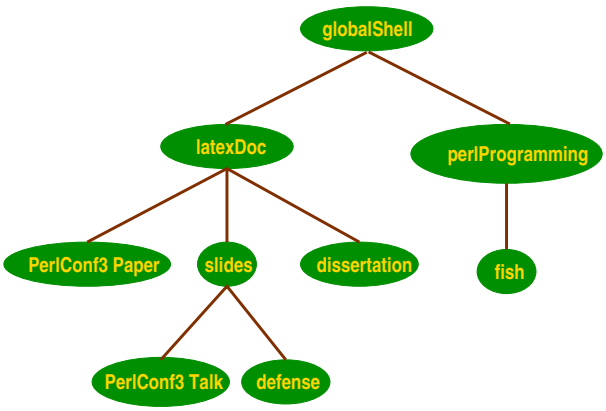

Figure 4: Project Scope Hierarchy

Figure 4 illustrates the author's current project scope hierarchy. The project scope *latexDoc* contains context specific to `latex` document processing such as an alias that deletes all files automatically generated by `latex` and `bibtex` and a function to perform the processing. The project scope *PerlConf3 Paper* has variables containing the name of the directory in which the files for this document are stored and the name of the main `latex` file. Project scope *globalShell* on the other hand, contains context that is useful regardless of the current task such as an alias to run an `xterm` with a scroll bar and a certain font size and a function to print a file to a particular printer depending upon the file type.

2.2 Language

To achieve the goals of customizability and extensibility, we must provide users with the means to do so to *Fish*, namely an *Interface Programming Language* (IPL) and principled access to internals of the shell. Since it is a shell, it also needs some sort of concise command language for brief interaction at the shell prompt. Instead of creating a single language to do both jobs, as with many other shells, we chose to use two separate languages. Figure 5 illustrates the relationship between the command language, the IPL, and results of command execution. The Parse::RecDescent module is used to parse the command line. Figures 6 and 7 present the syntax and interpretation phases of the command language of *Fish*.

The Interface Programming Language (IPL) of *Fish* consists of Perl and the Builtin and DataAccess functions of *Fish*. The Builtin functions implement the

Figure 5: Language Conversion

interface between the command language and the DataAccess functions of *Fish*. Since they are meant to be interactively called, they do extensive error checking on arguments. The Builtins can also be called directly from the IPL. The Builtin functions are intended to be added to or replaced by the user. The DataAccess functions implement principled access to the internals of *Fish*.

To write a user–defined function that can be invoked from the command language, the programmer need only know how to write Perl code and how arguments expressed in the command language are converted to arguments in the IPL. Arguments not inside a GROUPING are passed directly as scalars in the @_ array. Arguments inside a GROUPING are converted into an array and then a reference to that array is passed in the @_ array.

2.3 Architecture

The core of *Fish* is a long–lived process which we refer to as the *backend*. The first time a user uses *Fish*, the backend process starts. In general, it never stops; it is already running and users merely connect to it via a thin client which we refer to as the *frontend*. The user types commands to the frontend but the actual execution happens in the backend. The user may start as many frontends as desired, all of which connect to the single backend. The backend is multi–threaded for concurrent processing of frontend requests.

The backend is intended to persist (run forever), all the while accumulating context. Ideally, the user would have one backend running on each host that he uses regularly. Frontends are intended to be short–lived. They are meant to be used in the same way as multiple traditional shells — each in its own `xterm`. The user starts up a few *Fish* frontends and switches between them as he works. If he executes a time consuming command in the foreground in one frontend, he switches to another to continue his work. When the user is finished his work for the day, he kills all his frontends and logs out. The frontend uses the Term::ReadLine::Gnu module for interaction

WORD $\Longrightarrow$ ghostview *or* .xinitrc *or* 1276
Most combinations of letters, digits, and symbols but no white space

LITERAL $\Longrightarrow$ "Tulane University" *or* "$12.50"
Any combination of WORDs and white space enclosed in balanced double quotes

PERL CODE $\Longrightarrow$ { $Backend::DEBUG = 0; }
Any Perl code inside balanced braces

variable REFERENCE $\Longrightarrow$ $workingDir *or* $parentProject
A $ followed any valid Perl variable name

command history REFERENCE $\Longrightarrow$!32 *or* !dvip *or* !"tidyUp 4" *or* !(tidyUp 4)
A ! followed by a WORD, a LITERAL, or a GROUPING

result history REFERENCE $\Longrightarrow$ #128 *or* #"This is TeX, Ver"
A # followed by a WORD, a LITERAL, or a GROUPING

GROUPING $\Longrightarrow$ (juno pegasus rex choctaw) *or* (ls -la)
Balanced parentheses around WORDs, LITERALs, REFERENCEs, PERL CODE, and/or GROUPINGs separated by whitespace

IN–PLACE EVALUATION $\Longrightarrow$,WORD *or* ,"LITERAL" *or* ,(GROUPING) *or* ,{PERL CODE}
A comma in front of a WORD, a LITERAL, PERL CODE, or a GROUPING

Figure 6: Informal Description of the Command Language of *Fish*

1. **Alias Expansion**: The word in the operator position of the command is expanded if it is an alias.
2. **Syntax Expansion**: Any command or result history references are expanded.
3. **In–Place Evaluation**: Any in–place evaluations are performed and replaced in the command line by their results.
4. **Evaluation**: Any variable references are expanded.
5. **Execution**: The command line is executed.

Figure 7: *Fish* Command Language Interpretation Phases

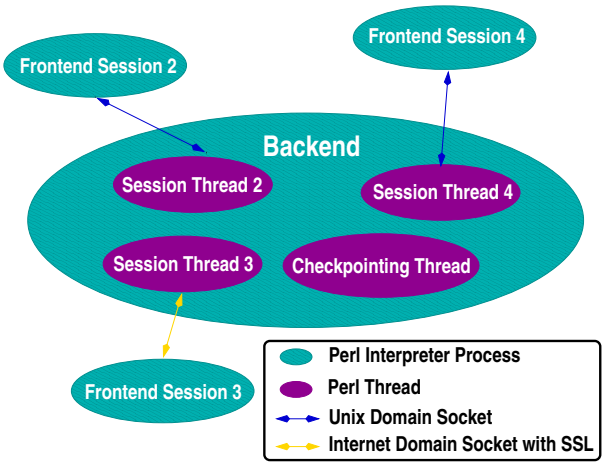

Figure 8: The Architecture of *Fish*

with the user.

Each time a frontend connects to the backend, a thread is spawned in the backend to handle this connection. This thread executes all commands issued to the corresponding frontend except for those specified to be backgrounded. Any backgrounded commands are run in a new thread. All knowledge is stored in the memory shared by all of the threads. Additional threads run in the background to listen for local and remote connections from frontends and to perform checkpointing.

The backend is able to execute *Fish* builtin functions, user–defined *Fish* functions, straight Perl code issued to the command prompt, and any binaries or scripts residing on the file system. *Fish* forks a subprocess to run external binaries and scripts. Perl code, which includes the builtin functions and user–defined functions, is run directly by the same Perl interpreter executing the backend. The architecture of the *Fish* prototype is depicted in Figure 8.

2.4 Security

Frontends can connect to local or remote backends. Since the backend runs as a process owned by its user, any commands or agents executing in the backend have access to all the user's resources just as the user does. Therefore, it is necessary to restrict access to the backend. Frontends communicate with a local backend via a Unix domain socket. Access to this socket is restricted by its permissions and enforced by the operating system. Internet domain sockets are used for communication between frontends and a remote backend. We chose to use the Secure Sockets Layer[FKK96] for secure communication over Internet domain sockets. The prototype uses the Net::SSLeay module for authentication and encryption. Each user of *Fish* must have a public/private key set and a certificate. The user must provide his pass phrase each time he starts the backend (a rare event) or any frontends connecting to a remote backend. Connections between backends and frontends are restricted to those owned by the same user.

2.5 Modes of Computation

Other programs can communicate with the *Fish* backend process and take advantage of its store of knowledge. The frontend can take almost any form, the backend does not depend upon the frontend at all. As far as the backend is concerned, the frontend, what ever it may be, send commands and receives results. It does not matter to the backend how the frontend acquires those commands or what it does with the results.

Since the *Fish* backend process is owned by its user, it has access to all that the user has access. Threads running in the backend can do work for the user while he attends to other tasks, thereby supporting computation by proxy. The longevity of the *Fish* backend supports autonomous computation. These threads can live forever doing work at certain intervals or upon certain conditions. The secure remote connections allowed by the backend support remote procedure call and mobile code. Connections to remote backends can be initiated by frontends, backends, and any other processes that 1) observe the communication protocol and 2) are authenticated to be owned by the same user as the remote backend. Since backends allow multiple connections, distributed computation is available with no extra effort on the part of the user. *Fish* provides a single unified interface to all of these various modes of computation.

3 Examples

In each of the *Fish* transcripts THE PROMPT, **the user's input to the shell**, `the output of the shell`, and `asynchronous messages from agents` are displayed in different styles. In the transcripts

the output of several commands was edited for presentation purposes; the user's input to *Fish* is shown as typed.

3.1 Exploiting Structured Information in Results

A primary influence on *Fish* is the notion that real interaction between humans and computers can only take place when the conversation goes two ways. Most commands in Unix send their results of interest as text to standard output. This textual output often contains useful structured information such as file names, telephone numbers, dates, times, stock symbols, equations, citations, etc. In traditional shells this output is displayed but otherwise discarded. *Fish* not only captures this text for potential later use but also provides parsing capabilities to extract this structured information. The user can then write functions that take this text, pass it through the appropriate parser, and make use of the extracted information. This not only eliminates the step of re–typing/cutting/copying the useful piece(s) out of the larger body of text but also allows for automation of the "pasting" of this information, i.e, doing the right thing with it. Additionally, since these results are temporally local to the interaction with *Fish*, indexical reference can be used as a means of abbreviation. Figures 9 and 10 illustrate these points.

The `elmAlias` user–defined function takes as input text, parses it for any email addresses, presents them to the user, and creates an `elm` email alias for the chosen address. When given an indexical reference of "it" rather than text, `elmAlias` retrieves the previous result and uses that as input. The `visit` user–defined function operates in much the same way. `visit` parses its input for URLs, presents them to the user, and sends a message to the concurrently running `netscape` to go to the selected web address. If the `visit` function receives an indexical reference of "it", the previous result is used as input. Appendix B presents the code of `elmAlias` and Appendix A presents the code of `visit`.

3.2 Distributed Log–Checking

Figures 11 and 12 show an agent that performs distributed, autonomous computation by proxy. It runs in the background, periodically checking the /var/adm/sulog file on multiple machines for suspicious entries. The /var/adm/sulog file is a record of all attempts by users on the system to execute the `su` command. Each time the agent finds new, suspicious entries, it notifies the user by sending a message to the next frontend he uses. The agent does this by placing a message in the *backend message queue* which is a facility for asynchronous messaging by autonomous computational entities. It is an abstract communication mechanism; the agent merely places the message in the queue and *Fish* does the right thing with it. The queue is discussed further in Section 5.

The code of the agent is shown in Appendix C. The agent makes use of the `runDistrib` *Fish* builtin function which takes as input a list of hosts and one or more commands and runs those commands on each machine. To do this, the current backend pretends to be a frontend and issues those commands to the backends on each of the hosts for remote execution. The remote backends return the results to the current backend and those accumulated results are then returned as the result of `runDistrib`.

4 Related Work

We are inspired by the phenomenon of Emacs and Elisp. Users are able to interact with Emacs concisely using key–bound functions and function names but are also able to extend its capabilities with a full programming language. Emacs was created to be a text editor but it has evolved as people have written various extensions for it in Elisp. Those extensions range from editing commands for many languages, to a web browser[EW3], and even to a robot client[JKW98] which allows mobile robots to be controlled by key–bound functions in Emacs. Although only programmers are able write these extensions, anyone can use them afterwards. This collaborative software development has driven Emacs in directions never dreamed of by its creators.

Attempts to improve the shell interaction language include *Rc*[Duf90], *Es*[HR93], and *Scsh*[Shi94]. Attempts to ease the burden of complexity on the user range from Essence[HS94] which works on top of existing technology (namely Unix) to alleviate file system complexity, to Softbots[EW94] which allow the user to specify *what* information he wants rather than *how* to retrieve it, and to Plan 9[PPD+95] which is a revolutionary operating system that pro-

```
SESSION4:PERLCONF> cat notification
From gnat@localhost.frii.com Fri May 21 00:18:07 1999
Received: from  (hdsl34.dnvr.uswest.net [209.180.255.34])
       by rex.eecs.tulane.edu for <terry@eecs.tulane.edu>
       Fri, 21 May 1999 00:18:04 -0500 (CDT)
Received: (from gnat@localhost)
       by localhost.frii.com (8.8.7/8.8.7) id XAA19703;
       Thu, 20 May 1999 23:17:23 -0600
Date: Thu, 20 May 1999 23:17:23 -0600
Message-Id: <199905210517.XAA19703@localhost.frii.com>
To: terry@eecs.tulane.edu
From: Nathan Torkington <gnat@frii.com>
Subject: 1999 Perl Conference Paper
Content-Length: 1072
Status: OR

Thanks for submitting your paper titled:
  Finally an Interactive Shell
to the 1999 Perl Conference.

It's my pleasure to inform you that it has been
accepted!  In the coming weeks you'll need to produce
the final copy of the paper for us to put on the
conference CD and in the printed proceedings.  For
this we'd like you to follow the Usenix guidelines on:
  http://www.usenix.org/events/usenix99/instrux/
              instructions.html
(of course, where it says "Usenix Annual Technical
Conference", we'd like you to pretend it says "The
Perl Conference").

Please mail your HTML and Postscript files to
<pc99submissions@perl.org>,
along with any questions you might have.

   ***The deadline for submissions has moved!***
We now need your submissions by June 15.  This will
let us proofread and identify any last-minute
corrections to be made before they go to the printer.

If you'd like help with your paper, please contact
the conference committee at <conference@perl.org>.
```

Figure 9: In this transcript, a recently received paper acceptance notification is `cat`-ed out. There is good deal of structured information in the text of the message: several email addresses, hostnames, URLs, dates, times, etc. The transcript continues into Figure 10.

```
Congratulations!

Nathan Torkington
<gnat@frii.com>
Chair, 1999 Perl Conference
Program Committee

SESSION4:PERLCONF> elmAlias it
Multiple Email Addresses were found:
[0] gnat@localhost.frii.com
[1] conference@perl.org
[2] gnat@frii.com
[3] terry@eecs.tulane.edu
[4] XAA19703@localhost.frii.com
[5] pc99submissions@perl.org
Please select one by number: 5
Enter the alias: pcsub
Enter the real name or organization:
   Perl Conference Submissions

Elm email alias pcsub made for
    Perl Conference Submissions
    pc99submissions@perl.org

SESSION4:PERLCONF> visit #From
Visiting http://www.usenix.org/events/
         usenix99/instrux/instructions.html
```

Figure 10: The `elmAlias` function is run with the indexical reference of "it". Six email addresses are found within the previous result. The paper submission email address is chosen and an email alias is created for that address. The `visit` function is also run on the same result. `#From` is a reference into the result history of the current session for the most recent result beginning with the string "From". Only one URL is found within the message which `visit` sends to `netscape` for display.

```
SESSION4:GLOBALSHELL> loadFile UserDefFuncs/CheckSulog.pm
File UserDefFuncs/CheckSulog.pm has been
loaded into scope globalShell.

SESSION4:GLOBALSHELL> setVariable CheckSulogSleep 240
"CheckSulogSleep" has been assigned to
"240" in globalShell.

SESSION4:GLOBALSHELL> AgentCheckSulogs (juno pegasus jupiter) 240 &
Thread=SCALAR(0x9dae3c)

SESSION4:GLOBALSHELL> pwd
/home/terry/research/Fish5

--- SU Violations on juno ---
 05/27/1999 18:37 - pts/9 terry-root
 05/27/1999 18:27 - pts/9 terry-root
 05/27/1999 17:04 - pts/11 terry-terry
 05/26/1999 20:45 - pts/2 terry-root
 05/26/1999 20:37 - pts/2 terry-root
 05/26/1999 20:32 - pts/2 terry-root
 05/26/1999 20:31 - pts/2 terry-root
 05/26/1999 20:28 - pts/2 terry-root
 05/26/1999 20:26 - pts/2 terry-root
 05/26/1999 20:19 - pts/2 terry-root
 05/26/1999 20:17 - pts/2 terry-root
 05/26/1999 20:16 - pts/2 terry-root
------------------------------
```

Figure 11: In this transcript, we run an agent in the background that periodically checks the sulogs on serval machines for any suspicious entries. The first command loads the definition of the agent into the backend. Next we set the variable CheckSulogSleep which the agent specifically checks to determine how often to run its algorithm. We then invoke the agent with the list of hosts that we want it to monitor. By default, the first time it runs its algorithm, it reports all suspicious entries for the past 24 hours. After that, it only reports any new suspicious entries since its last check. Its first report indicates that terry, who is not a priviledged user, has been trying to become root on juno.

```
SESSION4:GLOBALSHELL> cd UserDefFuncs/
/home/terry/research/Fish5/UserDefFuncs

--- SU Violations on jupiter ---
 05/27/1999 18:26 - pts/7 terry-root
------------------------------

--- SU Violations on pegasus ---
 05/27/1999 18:28 - pts/10 terry-root
------------------------------

SESSION4:GLOBALSHELL> runDistrib (juno pegasus jupiter) (date) (hostname)
juno
 Thu May 27 19:03:35 CDT 1999
 pegasus
 Thu May 27 19:03:43 CDT 1999
 jupiter
 Thu May 27 19:03:47 CDT 1999

SESSION4:GLOBALSHELL> who
saavedra console May 27 16:40
mb       pts/0   May 27 15:51 (thoth)
terry    pts/1   May 27 16:07 (choctaw)
terry    pts/3   May 27 16:22 (choctaw)
saavedra pts/6   May 27 16:58 (     )
saavedra pts/8   May 27 16:58 (     )
saavedra pts/9   May 27 17:30 (     )
terry    pts/7   May 27 18:24 (juno)

--- SU Violations on juno ---
 05/27/1999 19:04 - pts/9 terry-root
------------------------------

--- SU Violations on pegasus ---
 05/27/1999 19:04 + pts/10 terry-terry
------------------------------
```

Figure 12: We continue on with our work. Meanwhile, terry, or someone who has broken into her account, is still trying to su to root but now also on jupiter and pegasus. This example is contrived, but in a real instance we would imagine that the user (with powers of system administration) would have frozen terry's account by now.

vides the user with a customized view of the local network. Structured information found in everyday documents has been exploited by Apple Data Detectors[NMW98]. Once a document or portion of a document is parsed with the available grammars, a list of actions relevant to the type(s) of information found is presented to the user. The user may then select an action to be performed on the data found.

5 Discussion

> Note that threading is a highly experimental feature, and some known race conditions still remain. If you choose to try it, be very sure to not actually deploy it for production purposes. README.threads has more details, and is required reading if you enable threads.
> Build a threading Perl? [y]
> *excerpt from Perl* Configure *script output*

We began the implementation of *Fish* in Perl, experimented with Python briefly when we realized that we needed threads, and then switched back to Perl when threading became available shortly thereafter in the Summer of 1998. We were wary when we read the warning in README.threads but decided to go forward anyway. Although using Perl made some parts of this endeavor move along swiftly, other parts were quite frustrating. The newness of Perl's threading facility and the scarcity of adequate documentation and examples made us feel like pioneers in many respects. In our original plan for persistence of knowledge, we wanted to use a database for fine–grained access to secondary storage rather than just dumping the entire environment to a file. We attempted to use the MLDBM module with the Storable and SDBM_File modules. It crashed every time we tried to access the database. We soon discovered that the problem was with the SDBM_File module and that none of the DBM modules would work with multi–threaded Perl code. This fact is now mentioned in the current README.threads.

Although the current prototype well–serves the purpose of "proof of concept", it is not ready for real–world use. The code sorely needs an optimization pass to bring its speed up to a reasonable level. Performance measurements were taken on a SPARC-system 600 to compare the resource usage of a *tcsh* process to the *Fish* backend. They were run simultaneously over a period of 90 hours, along with many other processes running on the same machine. Both shells were in use during the same time periods. CPU usage is compared in Figure 13 and memory usage is compared in Figure 14.

In general, the backend is not as robust as it should be. Occasional segmentation faults are due to to inadequacies in Perl's thread implementation and the use of modules that are not thread–safe. Also, if the user starts any threads in the backend that get caught in an infinite loop or are waiting for an event that will never happen, they hang forever. Perl's threading facility provides no means to kill a running thread. Therefore, although the backend is meant to run forever, it must be shutdown occasionally to clean up these threads.

We are currently contemplating what language(s) we should use to implement the next version. In order to be able to use Perl, the following things must be done:

1. Perl's threading facility must be improved.
2. Modules in Perl's core distribution must be made thread–safe.
3. Some of the functionality provided by external Perl modules must be specifically re–implemented for use with *Fish* for the purpose of optimization.

Refer to [Ter99] for more information about *Fish*. It presents many interesting features of *Fish* that were not within the scope of this paper such as human–assisted deletion algorithm to deal with monotonically growing context. The human–assisted deletion algorithm suggests not recently used and/or large context items to the user for possible deletion. Future plans for the backend message queue are also detailed. A wide variety of notification and delivery methods for prioritized messages will be supported. For example, when the Check Sulog Agent detects a serious security breach, it will place a high priority message in the queue. Once handed to the shell, if the user is logged out it could page the user with a numeric code or, if he has an alphanumeric pager, *Fish* could page him with the message itself. A learning agent is presented in [JT99] which watches the user's input across multiple frontends to automatically write and instantiate *Fish* user–defined functions to perform repeated sequences of commands. Papers and source code are available on the web http://www.eecs.tulane.edu/Terry and also via ftp ftp://ftp.eecs.tulane.edu/pub/terry.

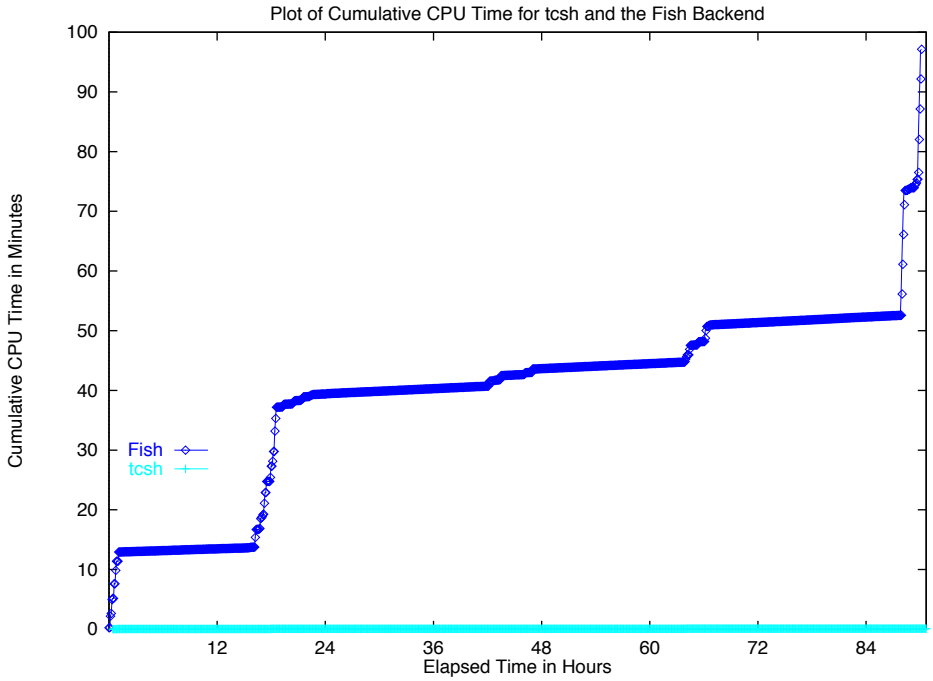

Figure 13: Cumulative CPU Usage: The plateaus in the plot of *Fish* data correspond to a periods of time in which neither of the shells were in use. The sharp inclines correspond to periods of usage of both shells. The final value of cumulative CPU time for *tcsh* was 6 seconds and for the *Fish* backend was 97 minutes.

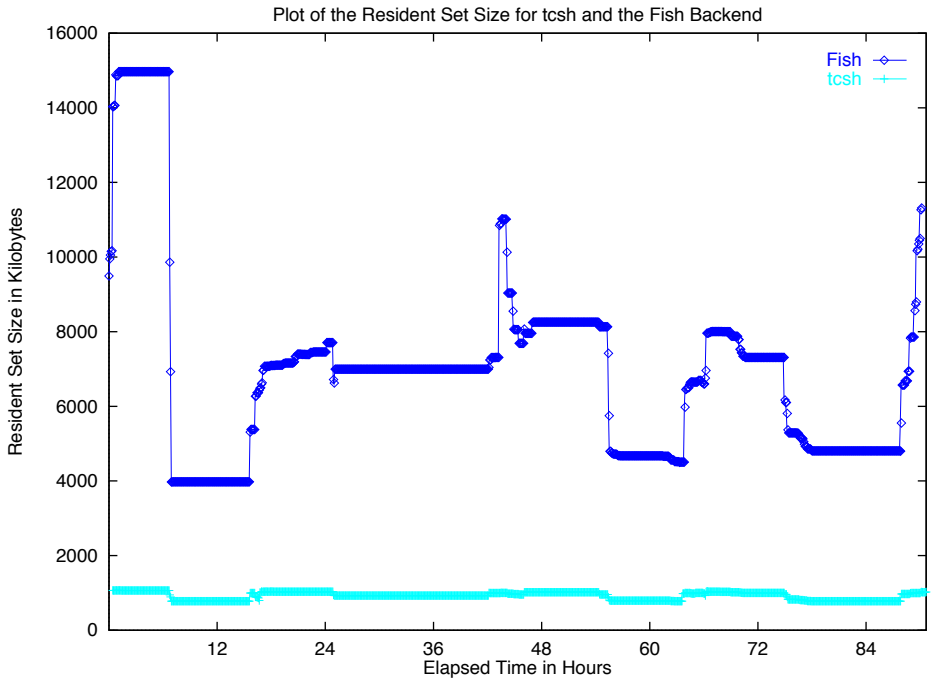

Figure 14: Memory Usage: The valleys and plateaus in both plots represent periods of inactivity and activity, respectively.

6 Conclusion and Future Work

The goal of this work was to provide a testbed for experimentation with various powerful user interface features, notably:

1. the automatic parsing of results;
2. a coherent interface programming environment (for customization and extension); and
3. an organizational structure for contextual information.

Because of the emphasis on flexibility and rapid development, Perl was well–suited to this work. Its module system was especially powerful, and its recent (though fragile) thread support essential. Whether the next version is implemented in Perl or not depends upon how Perl evolves.

Planned development of the IPL requires the inclusion of much of the functionality of the sample code shown in the appendices of this paper. For example, the ability to specify *typed* function arguments will allow Fish to automatically invoke the appropriate parser on the input (argument) to the function. A default exception handler would prompt the user when more or less than one item of the appropriate kind is parsed from the input. This would reduce the functions shown in the appendix to very brief and more high–level definitions. The design of such an IPL is an active area of continuing work on Fish.

This work takes many existing technologies and combines them in ways we have not found elsewhere. *Fish*'s architecture is based upon the client/server model which has not been applied to shells until now. By providing a long–lived multi–threaded server as the core of the shell, tasks traditionally run as `cron` jobs are taken to the next level. In an important sense, `cron`'s world (input and output) is limited to the file system, whereas a *Fish* task has access to much more information, especially information about the user's current status and activities. Additionally, the secure remote connections to *Fish* backends allow for a unified interface to various modes of computation. Therefore, tasks such as system administration can be done distributively with ease.

It is a well known fact that many people do not make use of the advanced features of currents shells. It is our hope that when *Fish* is ready for real world use, people will take advantage of its features. We imagine that with automatic persistence eliminating the need for users to edit a shell start up file and with centralized knowledge eliminating the need for users to `source` their shell startup file in each concurrently running traditional shell when the file changes, users will provide more context to their interface. Given a full programming language and principled access to shell internals, we envision many users customizing and extending *Fish* to suit their needs. With open source, users will be able to share this functionality with others.

References

[Duf90] Tom Duff. Rc-a shell for Plan 9 and Unix systems. In *UKUUG Conference Proceedings*, pages 21–33, 1990.

[EW94] Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, July 1994.

[EW3] Emacs/w3. http://www.cs.indiana.edu/-elisp/w3/docs.html.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol. http://home.netscape.com/eng/ssl3/-draft302.txt, November 1996.

[HR93] Paul Haahr and Byron Rakitzis. *Es*: A shell with higher-order functions. In *1993 Winter USENIX Technical Conference*, pages 53–62, 1993.

[HS94] Darren R. Hardy and Michael F. Schwartz. Customized information extraction as a basis for resource discovery. *Transactions on Computer Systems*, 14(2):171–199, May 1994.

[JKW98] J. Jennings and C. Kirkwood-Watts. Distributed mobile robotics by the method of dynamic teams. In 4^{th} *International Symposium on Distributed Autonomous Robotic Systems*, Karlsruhe, Germany, 1998.

[JT99] James S. Jennings and Nicole D. Terry. Towards more intelligent and interactive interfaces. In Amruth N. Kumar and Ingrid Russell, editors, *Twelfth International Florida AI Research Society Conference*,, pages 24–31. AAAI Press, 1999. Orlando, Florida, May 3-5.

[NMW98] Bonnie A. Nardi, James R. Miller, and David J. Wright. Collaborative, programmable intelligent agents. *Communications of the ACM*, 41(3):96–104, March 1998.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan

9 from Bell Labs. http://plan9.bell-labs.com/plan9/doc/9.html, 1995.

[Shi94] Olin Shivers. A scheme shell. Technical Report TR-635, Laboratory for Computer Science, MIT, 1994.

[Ter99] Nicole D. Terry. *Gaining Wisdom with Age: A Long-Lived Shell.* PhD thesis, Tulane University, New Orleans, Louisiana, August 1999. forthcoming.

A visit.pm

```
use strict;
no strict 'refs';
use diagnostics;
use integer;

$Fish::USAGE{'visit'} = "Usage1: visit <text>\n" .
  "Usage2: visit it\n" .
  "Purpose: to tell the concurrently running " .
  "Netscape to go to a particular web page\n";
sub visit {
  my (@text, @urls, $numFound, $url);

  if (!$_[0]) { return($Fish::USAGE{'visit'}); }

  if ($_[0] =~ /^it$/) {
    @text = &Builtins::getLastResult();
  }
  else { @text = @_; }
  @urls = &Builtins::parseURLs(@text);
  $numFound = scalar(@urls);

  if ($numFound == 0) {
    return("Error: Did not find a url in @text.\n");
  }
  elsif ($numFound == 1) { $url = $urls[0]; }
  else {
    my $sock = &Helpers::beginInteract();
    send($sock, "Multiple URLs were found:\n", 0);
    my $i;
    for ($i = 0; $i <= $#urls; $i++) {
      send($sock, "[$i] $urls[$i]\n", 0);
    }
    send($sock, "Please select one by number: ", 0);
    $i = <$sock>;
    &Helpers::endInteract();
    $url = $urls[$i];
  }
  `netscape -raise -remote \'openURL($url)\' 2>&1`;
  return("Visiting $url\n");
}

return 1;
```

B elmAlias.pm

```
use strict;
no strict 'refs';
use diagnostics;
use integer;

$Fish::USAGE{'elmAlias'} =
  "Usage1: elmAlias <text>\n" .
  "Usage2: elmAlias it\n" .
  "Purpose: to create an elm alias for a " .
  "email address\n";
sub elmAlias {
  my (@text, @addresses, $numFound, $address,
     $alias, $name);

  if (!$_[0]) { return($Fish::USAGE{'elmAlias'}); }

  if ($_[0] =~ /^it$/) {
    @text = &Builtins::getLastResult();
  }
  else { @text = @_; }

  @addresses = &Builtins::parseEmailAddresses(@text);
  $numFound = scalar(@addresses);

  if ($numFound == 0) {
    return("Error: Did not find an email address ".
      "in @text.\n");
  }

  my $sock = &Helpers::beginInteract();
  if ($numFound == 1) { $address = $addresses[0]; }
  else {
    send($sock,
         "Multiple Email Addresses were found:\n",
         0);
    my $i;
    for ($i = 0; $i <= $#addresses; $i++) {
      send($sock, "[$i] $addresses[$i]\n", 0);
    }
    send($sock, "Please select one by number: ", 0);
    $i = <$sock>;
    chomp($i);
    $address = $addresses[$i];
  }
  send($sock, "Enter the alias: ", 0);
  $alias = <$sock>;
  chomp($alias);
  send($sock,
       "Enter the real name or organization: ",
       0);
  $name = <$sock>;
  chomp($name);
  &Builtins::appendToFile(
              "/home/terry/.elm/aliases.text",
              "$alias = $name = $address\n");
  &Helpers::endInteract();
  return("Elm email alias $alias made for " .
         "$name $address\n");
}

return 1;
```

C CheckSulog.pm

```
use strict;
no strict 'refs';
use diagnostics;
use integer;
use POSIX;
use Date::Manip;

$Fish::USAGE{'AgentCheckSulogs'} =
  "Usage: AgentCheckSulogs (<host> ...) " .
  "[<sleep time in seconds>]\n" .
  "Purpose: to monitor sulog on various hosts\n";
sub AgentCheckSulogs {
  my ($newtime, @violations, $sleeptime, $host);
  my $starttime = &Date::Manip::DateCalc
      ("now", "1 day ago");
  my $dftSleeptime = 1200;
  my $hostRef = $_[0];
  if (!ref($_[0])) {
    return($Fish::USAGE{'AgentCheckSulogs'});
  }
  if ($_[1]) { $dftSleeptime = $_[1]; }

  &Builtins::runDistrib($hostRef, ["loadFile",
    "/home/terry/fish/UserDefFuncs/CheckSulog.pm",
    "globalShell"]);
  while (1) {
    $newtime = &Date::Manip::DateCalc("now",
                              "1 minute ago");
    ### Run it on each machine
    foreach $host (@$hostRef) {
      @violations = &Builtins::remoteRun($host,
                        ["checkSulog", $starttime]);
      if (@violations) {
        &Helpers::sendMessage(
            "\n--- SU Violations on $host ---\n",
            @violations,
            "------------------------------\n");
      }
    }
    $starttime = $newtime;
    ($sleeptime) = &Builtins::getVariable
                            ("CheckSulogSleep");
    if ($sleeptime =~ /^Error/) {
        $sleeptime = $dftSleeptime;
    }
    elsif ($sleeptime == 0) { last; }
    sleep($sleeptime);
  }
}

$Fish::USAGE{'checkSulog'} =
  "Usage: checkSulog <YYYYMMDDHH:MM:SS>\n" .
  "Purpose: to check sulog for suspicious entries\n";
# originally written by Mark Hershberger
# adapted for use by Nicole Terry
sub checkSulog {
  my ($starttime) = @_;
  if (!$starttime) {
    return($Fish::USAGE{'checkSulog'});
  }

  # Read the file in, find the sys admins.
  my @admingroups = ("staff", "sys");
  my $groupfile   = "/etc/group";
  my ($group, @line, @admins);
  open(GROUP, "<$groupfile") ||
    return("Couldn't open $groupfile: $@\n");
  while(<GROUP>){
    chomp;
    @line = split /:/;
    foreach $group (@admingroups){
      if($line[0] eq $group){
        push (@admins, split(/,/, $line[3]));
      }
    }
  }
  close(GROUP);

  # Read in the SUlog in reverse into @sulog
  my $sulogfile   = "/var/adm/sulog";
  my @sulog;
  open(SULOG, "<$sulogfile") ||
    return("Couldn't open $sulogfile: $@\n");
  while(<SULOG>){
    unshift @sulog, $_;
  }
  close(SULOG);

  my $lastdate = &Date::Manip::ParseDate("today");
  my $year = &Date::Manip::UnixDate("today", "%Y");
  my ($user, @violations, $day, $time, $perp,
      $thistime, $flag);
  DATE: while ($_ = shift @sulog){
    @line = split(/\s+/, $_, 4);
    $day  = $line[1];
    $time = $line[2];
    $perp = $line[3];

    # A hack.  Only month/date is stored in sulog.
    # We need a way to decrement the year..
    $thistime = &Date::Manip::ParseDate($day);
    $flag = ($thistime cmp $lastdate);
    if ($flag == 1) { $year = $year - 1; }
    $lastdate = &Date::Manip::ParseDate($day);

    # Make sure we have the right date
    $thistime = &Date::Manip::ParseDate
                    ("$day/$year $time");
    $flag = ($thistime cmp $starttime);

    if ($flag >= 0) {
      # Get rid of all admins (backup, root, etc.)
      foreach $user (@admins){
        if(/\s+$user-/ and $user ne ""){ next DATE; }
      }
      # found a violater!
      push(@violations, "$day/$year $time $perp");
    }
    else { last DATE; }
  }
  return(@violations);
}

return 1;
```