The following paper was originally published in the
Proceedings of the Sixth Annual Tcl/Tk Workshop
San Diego, California, September 14–18, 1998

wshdbg - A Debugger for CGI Applications

Andrej Vckovski
Netcetera AG

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

wshdbg - A Debugger for CGI Applications

Andrej Vckovski
Netcetera AG
vckovski@netcetera.ch

Abstract

This contribution discusses `wshdbg`, an interactive, remote debugger for CGI applications written in pure or derived Tcl-based environments such as `websh`. The discussion covers a short overview of the `websh` environment and an analysis of current techniques and impediments of CGI debugging. The debugger presented consists of a client-server architecture, where the server is running on the same host as the Web browser, while the client is included in the CGI application that needs to be debugged. The resulting environment does not provide the level of sophistication known from typical source level debuggers, yet it presents a significant step forward compared to typical CGI debugging techniques which rely on tracing the execution by many log or debug messages. The system has been in operational use in the last two years and proved to be a great help in debugging large Web-based applications, allowing high-level software engineering for Web applications on nearly the same level as it is being done in traditional software development.

1 Overview

This contribution presents an interactive debugger for CGI (Common gateway Interface [1]) applications written in pure Tcl or derivations thereof. CGI has been the first standardized technique that has been available to extend Web-servers to provide dynamic content and it is still - despite of some drawbacks - the most frequently used method to interface specific applications for dynamic content creation.

Debugging such CGI applications has always been a difficult and cumbersome task. CGI applications are growing in size and complexity and suitable debugging techniques need to applied to support the software engineering process. The lack of useful tools motivated the development of a simple yet powerful remote debugger for CGI applications which has been primarily targeted for the use in the `websh`-Framework which will be discussed below. However, the debugger can be used within other Tcl-based environments for Web-Applications such as Don Libe's `cgi.tcl` [2] or Neosoft's `NeoWebScript` [3].

The first section will give a short overview on the `web++`/`websh`-framework and motivate the use of CGI as server extension mechanism. This is followed by a short discussion of the generic difficulties in debugging CGI applications and a review of debugging techniques typically used. Then, the architecture and some implementations aspects as well as examples are discussed. The contribution is concluded by an outlook to our future work and some lessons learned using the debugger in various projects.

2 The `web++/websh` framework

The `web++/websh` framework is a software development and runtime environment for Web-based applications. It consists of two major components:

`web++`

`web++` is a C++ class library which provides utility classes for many standard tasks such as managing the CGI protocol, logging, session management, (HTML) template processing and so on, much similar to the well-known packages such as `CGI.pm` [4] or `cgi.tcl` [2].

`websh`

`websh` (pronounced *web-shell*) is a Tcl-shell which is based on `web++` and provides most of the functionality of `web++` on a script level. The objects and methods within `web++` are made visible on the Tcl-level as Tcl-commands.

When web++ was developed a few years ago the main incentive was to build Web-based applications as a specialization of the web++ class library, using the embedded Tcl interpreter merely as a very comfortable and powerful way for application configuration and template-based page construction. However, the experiences have shown, that a classical C++-development approach with relatively long turnaround cycles and high level of sophistication by the developers does not meet the requirements of typical web-based applications. Therefore, the entire functionality of the web++ class library has been exported as Tcl-commands in a specialized shell (websh) and the application development happens by writing websh- or Tcl-code, respectively.

Unlike usual CGI applications in Tcl, websh processes the code after evaluating the script file(s). The code merely declares callback procedures or code blocks which are called by websh after initializing and parsing the CGI input. This allows a very flexible yet simple way to write applications with *multiple states*. A command dispatcher selects the state requested by the corresponding URL (given by an encoding in the so-called *query string*) and calls (or evaluates) the appropriate callback code.

Other features of websh include:

Session management

websh implements a session management on top of the connectionless HTTP-protocol. Sessions can be associated with persistent *dictionary* objects (key-value pairs) on the Web server.

URL encryption

websh provides a simple URL encryption scheme to hide parameter passing details. This is necessary because session keys and state information are passed within the URL's query string.

Flexible logging

websh implements a two-level logging scheme. Every log message which is generated within an application or the interpreter itself consists of a *message text*, a *severity level* (debug, info, error, alert) and a free-string *facility code*. In the first stage, the severity level and facility code are used to filter the messages according to a user defined set of rules, e.g., "pass all messages with level info and debug messages for facility foo". A message that passes the filter is then routed to a set of log output channels. A log output channel can be either a file, any opened Tcl channel (e.g., pipe, socket, file) or, on Unix systems, the syslog service. Every log channel is also associated with a rule that defines which messages to accept. E.g., it is possible to route only alert messages to syslog and having all other messages in a log file. The logging mechanism is insofar very important as most Web-based applications are expected to run unattended without permanent operator control.

Template processing

The template processing used in websh allows templates (e.g., HTML page templates) to be processed with substitution of embedded directives. Templates can be entire HTML pages or only fragments thereof. The mechanism adopted is unlike most other approaches in that it uses a tagging which is *orthogonal* to SGML or HTML, respectively. This allows nesting within the templates and avoids possible name clashes with future HTML extensions.

Database connectivity

websh and web++ provide a lightweight database connectivity layer called webdb++. It allows simple (mostly synchronous) database connections to relational, object-relational and inverted-list systems. The database connectivity layer is implemented using database connectors that are available in the Tcl community such as Oratcl or Sybtcl [5].

More than CGI

The main usage of websh is to develop CGI applications. However, most large CGI application do contain components which need to run independent of a Web-server, e.g., housekeeping processes, import and export tools and so on. websh allows to reuse modules both within the CGI and non-CGI part of a software system.

Development tools

A set of development tools, such as a pseudo-linker (merely a module merger), a pseudo-compiler ("compiles" the websh/Tcl-code into shared objects), pretty printers and so on are available.

The powerful logging mechanism does provide a step towards *organized* and *controlled* CGI debugging. However, it is still far away from the usability that is known from traditional source level debuggers, e.g., debuggers for C/C++ applications. For that reason, we designed and developed a more powerful debugging approach

which is somewhat close to the ease-of-debugging found in state-of-the-art source level debuggers. The next section discusses some of the difficulties that have to be overcome with a successful CGI debugging approach.

3 Debugging CGI-Applications

3.1 Why CGI?

Before discussing CGI debugging in more detail it is worth considering the question whether CGI is at all appropriate for the development of Web-based applications. In the last years many other approaches for dynamic content creation have been proposed and developed:

Server-specific APIs

Many HTTP-servers provide specific application programming interfaces (API) to extend their functionality and embed dynamic content creation, such as NSAPI (Netscape [6], ISAPI (Microsoft) [7], Apache modules [8] or *Servlets* for Java-based servers.

Server-side includes (SSI)

Some HTTP-servers provide a simple template processing mechanism. SSI has been available with very early releases of the NCSA-web-server. Microsoft IIS uses a similar approach called *active server pages*.

Approaches particular to the back-ends

Web-applications that need to access mainframes or large databases can use specific HTTP servers that are particular to the back-end system, e.g, Oracle Web Server [9].

Compared to these approaches CGI has a major drawback: It requires the HTTP server to create a process ("fork/exec") for every request that has to be handled. Process creation has been very expensive in older operating systems and is still expensive compared to, e.g., a function call or thread creation which is needed in the case of a server-specific API. That is, CGI is not an optimal solution if performance is the major issue. However, there are two fundamentally important reasons which make CGI still the technique of choice for Web-based applications:

- CGI application are isolated in a separate process space. Ill-behaving CGI applications cannot impair a HTTP server's stability (assuming a *real* operating system). For many applications, long-term stability is much more important than a few performance issues that can be solved by better and faster hardware.
- The CGI is well-defined and standardized and therefore, portable across many server platforms. Dependency on a specific HTTP server products limits scalability and flexibility.

These two issues have motivated us to stick with CGI for most of our Web-based applications. The development framework is, however, not *fundamentally* bound to CGI. In the next section we will now discuss some of the impediments when debugging CGI applications.

3.2 Impediments

CGI applications receive their input using two separate mechanisms:

- Context information such as server information and so on are made available by the server using a set of defined variable in the process' environment.
- Request-specific data generated by the browser such as the content of a HTML form are passed on the process' standard input channel.

The response is send by the CGI process to its standard output channel, which is either collected by the server and sent to the browser or, already connected to the browser socket by the server (buffered vs. direct replies). CGI applications usually have a very short life time. The process dies as soon as a request is handled (i.e., the response is sent to the standard output). A multi-state application (e.g., a shopping bag application) typically consists of many request/response pairs, i.e., of many calls to the CGI application. Every instance might expect some other data on the standard input and a different context in the environment.

Therefore, a realistic debugging session needs both the expected data on standard input and the environment variables to be set according to the request that needs to be handled. Assume a CGI application written in a 3rd-generation language such as C. Using standard debuggers for C there are basically two alternatives for debugging:

- The CGI is simulated using a few environment variables and some captured data which are fed into the process' standard input. The process is started manually or by the debugger, respectively, and not by the HTTP server. This approach is very cumbersome because it needs manual maintenance of the input data (which can be different with every request) while having very short debugging sessions by the nature of CGI processes [10].

- The CGI application is started as usual by the webserver. The application is extended in a way that it stops and waits for any signal to proceed (e.g., loops until some condition is true). During this wait period, a suitable debugger can attach to the process and force the condition to become true. There are also tools that can be used to connect to the debugger from the application (i.e., the other way round). The drawback of this approach is that the application needs to be significantly extended to allow such remote debugging. It is, however, still the best method to do it. However, it requires powerful debuggers which allow, e.g., to attach to a running process. Also, this approach requires in the most cases that the debugger runs on the same system as CGI application. On productive systems, this is most often inhibited due to security reasons.

For scripted CGI applications (Perl, Tcl, Python etc.) the technique used most often is still the classic `"printf()"` debugging style. Many log messages are generated in the process giving information about the state of variables and flow of control. The log messages are either sent to a specific log file, that process' standard error (which is often copied into the server's error log) or even sent to the CGI application's output, i.e., intermixed with the "real" output (which works only, if the output is ASCII-text or HTML and not some other, binary-encoded MIME-type such as GIF images).

Often used are also wrappers that are called instead of the CGI application. These wrappers call the CGI applications in turn and provide a formatted output of the application's standard input, environment and the results of the CGI application.

Compared to "real" debugging environments these techniques seem to be anachronisms and far from supporting productive software development. The `wshdbg` (`websh`-Debugger) presented in the next section tries to overcome some of these impediments and provide a better way for debugging Tcl-based CGI applications.

4 websh-Debugger

4.1 Architecture

The `websh`-Debugger `wshdbg` is designed as a *remote* debugger which consists of a client-part and a server-part (see figure 1). The server part is a `wish`-Application which displays the debugger's user interface and waits for CGI applications to connect. The client part is a small stub which is included in the CGI-application when in debugging mode. When a CGI application is launched by the HTTP-server, it connects to the debug server and transfers all context information (environment, decoded data from standard input, decoded query string). The debug servers typically runs on the same platform as the Web browser and controls the further execution of the CGI application. After the first connection to the server, the debug clients awaits further commands from the server (e.g., continue execution). The CGI application may contain a set of break points and trace conditions (variable traces). Whenever a breakpoint is reached or a trace condition is met, the execution is stopped and control is "transferred " back to the server. In a stopped state, the CGI application accepts various commands form the server. These commands can be, for example, a valid Tcl command that can be used to query variables, temporarily evaluate expressions and so on.

This design is similar to corresponding approaches for remote debugging known from standard debuggers available on most platforms. Remote debugging is especially useful if disturbing the debugged platform needs to be minimized, such as kernel programming or applications with high GUI requirements (e.g., the debugger GUI should not interfere with the debugee's user interface). However, to our knowledge, `wshdbg` is the first operationally used debugger for CGI applications which uses a remote debugging technique. Other debuggers have been successfully used in interactive Tcl/Tk environments such as for example [13]

Breakpoints and trace conditions are inserted in the debugged application - being a Tcl or websh-script - directly in the source code. This is a drawback which is imposed by the interpreted nature of the environment. Practical use has shown, however, that this does not pose major impediments.

The debug server can simultaneously control several CGI applications, i.e., maintain several connections to CGI applications. This is useful if there are, for exam-

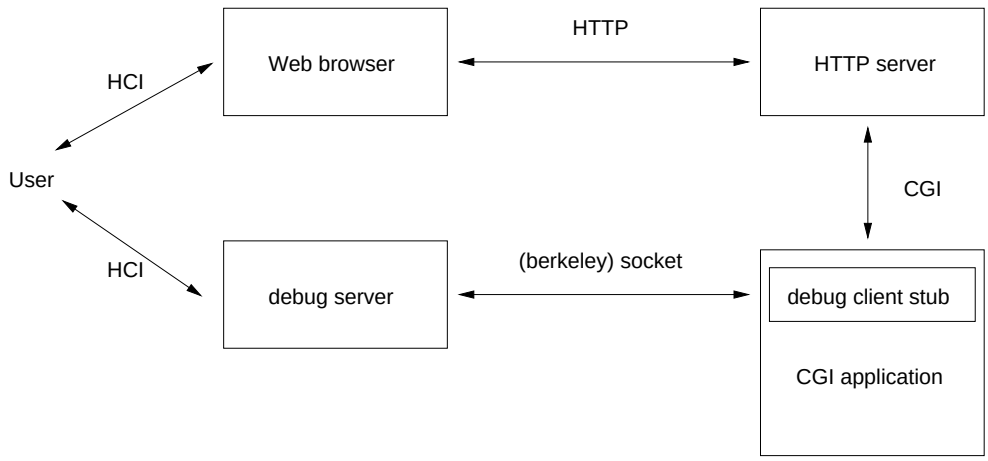

Figure 1: Architecture overview

ple, multiple requests needed for a response such as if both a HTML page and an embedded image are result of different instances of the same CGI application.

4.2 Implementation

The implementation is based on a simple bi-directional, message-oriented protocol. The protocol is implemented upon (berkeley) sockets and distinguishes two states, namely

1. the CGI application is stopped, and
2. the CGI application is running.

The *stopped* state is entered at start, at end (optional) and whenever a breakpoint is hit or a trace condition is met. The stopped state is left by selecting the *continue* action in the debug server. In a stopped state (i.e., the CGI application is awaiting commands from the debug server), the debugger provides following actions:

- The CGI environment (environment variables, decoded standard input etc.) can be inspected
- Tcl-code can be evaluated in the current context of the CGI application, allowing variables to be inspected and set, expressions evaluated and so on.
- Profiling can be enabled or disabled. The profiling can be used if the TclX [9] extension is available on the HTTP server platform . However, unlike typical profiling it collects profiling information over many instances of the CGI application, allowing better statistics for short-lived CGI applications.
- The CGI application can be prematurely terminated.
- The CGI application's execution can be continued. The CGI application continues to run until the next breakpoint is encountered or a trace condition is met.

On transition from a running state to a stopped state the client sends the server information about *where* it stopped. The location indication is a user-defined text which has been given when defining the breakpoint or trace condition.

Breakpoints and trace condition are defined as previously mentioned directly in the source code. A `brk` command (which takes optional arguments naming the location) defines a breakpoint. the `tr` *`variable`* command declares a trace condition which is implemented using the variable tracing mechanism of Tcl. In a typical debugging session, the source modules are kept open in an editor and break points are inserted as needed. This requires, however, that the code is debugged in an non-compiled or "obfuscated" state.

The following code snipplet shows the enabling of the debug client as well the definition of some breakpoints

```
wpp_command showdata {

    # use debugger (by the default,
    # connect to the address where
```

```
    # the request came from)
    usedbg

    set foo {some1 data1 some2 data2}

    foreach {key value} $foo {
brk having key $key
        wpp_puts "$key = $value"
    }
tr myvar "trace on myvar"
    # ... some processing

brk here is breakpoint 2

    # ... some more processing
}
```

Figure 2 shows a screen shot with the main debugger control panel, the CGI inspector and expression evaluator.

4.3 Log-Viewer

In the `websh`-environment, there is another useful tool available which supports debugging with additional information. The `websh` logviewer `wshlv` is a log server which runs similar to the debug server on the same host as the browser. When enabling the debug mode for the CGI application, a log channel to the log viewer (if available) is opened automatically. Every log message generated in the application and framework is sent to that log viewer. The log separates and stores log messages from various instances of the CGI application. Various runs can be compared without having a huge log file that contains intermixed messages from many requests. Figure 3 shows a sample screenshot from the log viewer.

Together with the log viewer, a typical debug session contains therefore of:

- a browser
- a debug server instance,
- an editor with the source modules, and
- a log viewer.

5 Conclusion and future work

The `websh`-debugger `wshdbg` has been successfully deployed in many large projects during the last two years. We have experienced a substantial improvement in debugging productivity compared to classical `printf()` style of debugging. Especially, if the application is dependent on the real server (and browser) environment, this approach has an important advantage. Often, such information can not be easily simulated as it involves many other components (e.g., authentication information provided by authentication services, mainframe connections and so on). In these cases, it is very helpful to debug applications in their real framework, yet having full control and inspection at runtime of the application. Compared to these advantages, it is acceptable that the debugged application needs to have a few additions (such as the inclusion of the client stub).

It might seem unsatisfying that the definition of break points and trace conditions has to happen directly in the source code. Especially if the application consists of several source modules which are merged into a large, single executable, this involves a "make"-style step after every breakpoint has been included or removed. However, these steps are usually rather quick in script-based environments since a build of an application merely consists of the merging source modules (unless there are packaging concepts used - which is usually not acceptable in production environments).

The entire debugging system is in fact a very small piece of code - the debug server consists of less than 500 lines of Tcl-code, the client stub some 200 lines of Tcl-code. Yet, it provides most of the functionality needed for the debugging of large Web applications.

The future work on the `websh`-debugger could consist of extending the protocol between debug server and client to allow additional break conditions and a better user interface for inspection. However, it might be sensible as well to integrate the commercial debugger `TclPro Debugger` from Scriptics, Inc. [12], as it certainly provides much more general debugging functionality than `wshdbg`.

In general, we would like to have a source level debugger that allows break points to be set without modifying source code by specifying file name or procedure name and line number and still is suitable for CGI development. In an interpreted environment, however, the help of the Tcl interpreter is needed. An possible extension to `wshdbg` would be the possibility to use the Tcl

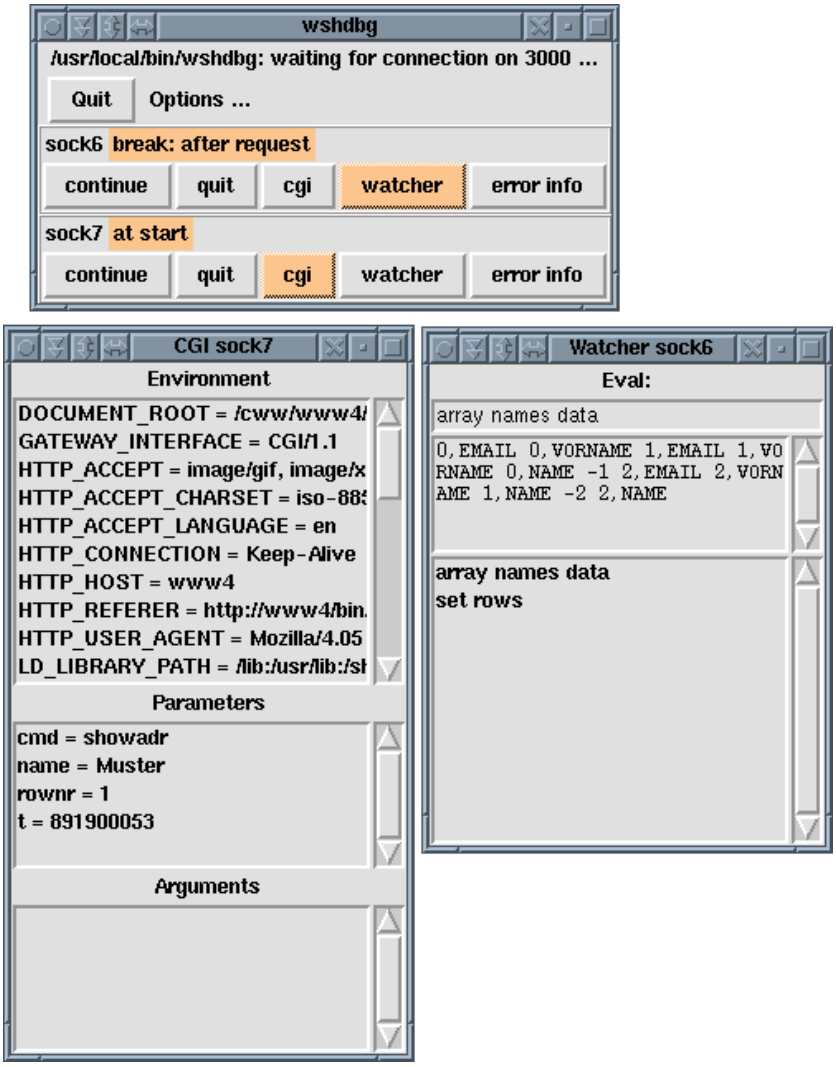

Figure 2: Debugger screen example

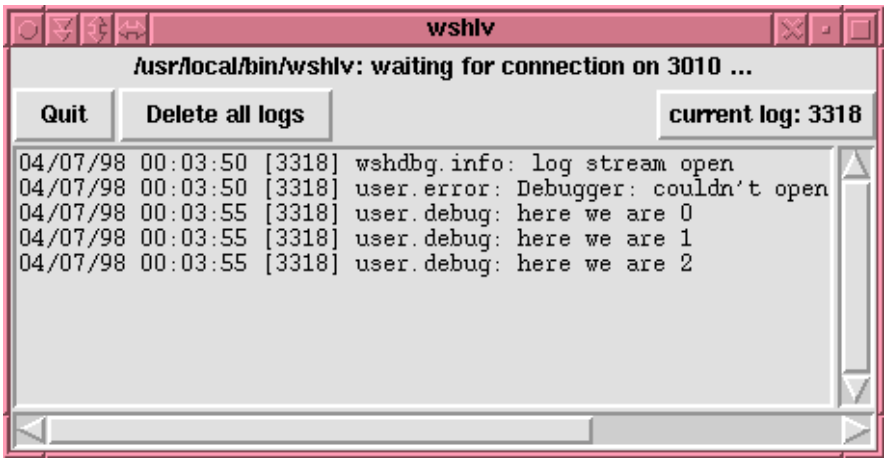

Figure 3: Log viewer screen example

library function `Tcl_CreateTrace()` as it is used, e.g., in TclX [11] to provide call backs that test for break conditions. Very useful would be, however, to provide a mechanism within Tcl that provides additional information such as the line number within the upper stack frame that executed the corresponding command. This would allow the debugger to recognize break conditions on module and line level. However, this would require the Tcl byte code compiler to operate in a specific debug mode where line number information is not only retained in the case of an error.

6 References

1. *The Common Gateway Interface Specification* <http://hoohoo.ncsa.uiuc.edu/cgi/interface.html>
2. *The cgi.tcl Home Page* <http://expect.nist.gov/cgi.tcl>
3. *NeoWebScript* <http://www.neosoft.com/neowebscript/>
4. *CGI.pm - a Perl5 CGI Library* <http://www-genome.wi.mit.edu/ftp/pub/software/WWW/>
5. Harrison, M. (ed.), 1997, *Tcl/Tk Tools*, O'Reilly and Associates, Cambridge (MA).
6. *NSAPI Programmer's Guide* <http://developer.netscape.com:80/docs/manuals/enterprise/nsapi/contents.htm>
7. *Taking the Splash, Diving into ISAPI Programming* <http://www.microsoft.com/mind/0197/isapi.htm>
8. *Apache API notes* <http://www.apache.org/docs/misc/API.html>
9. *Oracle Web Application Server* <http://www.oracle.com/st/o8collateral/html/xweb6ds.html>
10. Boutell, T., 1996, *CGI Programming in C & perl*, Addison-Wesley, Reading (MA).
11. *Extended Tcl* <http://www.neosoft.com/tclx/>
12. *TclPro Debugger* <http://www.scriptics.com>
13. Libes, D., 1993, A Debugger for Tcl Applications, *Proceedings of the 1993 Tcl/Tk Conference, (PostScript)* <http://www.mel.nist.gov/msidlibrary/doc/libes93c.ps>