The Igor System Administration Tool

Clinton Pierce
Decision Consultants Inc.

The Igor System Administration Tool

Clinton Pierce – Decision Consultants Inc.

ABSTRACT

This paper describes the system administration tool we call Igor. Igor is a tool for administrating a large number of UNIX systems in a diverse, networked environment. Igor consists of two parts, an interactive GUI which is controlled by an operator, and a daemon which is run on the UNIX target which actually executes the commands. Igor provides very fast operation, and quick post operation analysis of the results. In normal operations we have run commands on over 600 hosts simultaneously in 60 seconds.

History

Igor was created, because in Ford's highly distributed and diverse environment, the system administrators often found themselves needing to run something on several hosts very quickly. Some examples include holiday shutdown, system re-configuration, installing patches, fixing bugs, surveying systems for patches and software usage, and emergency damage control (see Script Examples below). Normally, this would be accomplished with something like this shell script:

```
#!/bin/sh
VICTIMS="hosta hostb hostc hostd"
for cur_host in $VICTIMS
do
 rsh $cur_host "/usr/bin/cmd -arg"
done
```

This has many potential problems, not the least of which are:

- 'rsh' not being able to portably return the status of remotely executed commands. This means that in order to find out if a command worked, sometimes elaborate shell-scripting is involved.
- stdin/stderr being mixed. This makes checking for errors even more difficult on complex scripts.
- Anything beyond simple commands may involve a full-blown shell script meaning complicated rsh commands, or rcp commands, or use of an NFS filesystem to transport the scripts.
- Slow hosts can bog down operations
- Hosts whose inetd/rshd has gone to lunch cause rsh not to work properly.
- This operation is serial. If rsh hangs, your whole list of hosts goes unprocessed.
- The "batch" style of operation is quite non-satisfying. Especially after a long run, you discover a subtle bug in the script on either end, causing you to have to repeat the lengthy run.
- rsh uses whatever "mystery shell" is on the other side – and whatever environment comes with it.

A very cleverly written shell script can get around almost all of these limitations. Certainly a C program could. However, Igor gets around all of these problems and provides a neat, clean user interface for getting these kinds of jobs done. In addition, other schemes such as running administrative scripts through cron do not allow for a really interactive approach to solving these problems.

Detailed Description

Igor solves these problems by creating a fast, robust, portable, and flexible method to distribute these kinds of jobs. Igor can be described simply as a multiplexed rsh. With Igor there are two parts, a target and an controller. Security is handled through traditional "rexec"-type security (/etc/hosts.equiv and rhosts files).

Igor accepts commands from a GUI on the controlling system, and passes those commands to a set of Perl scripts which distribute the commands to the remote (target) systems. These Perl scripts maintain several remote connections simultaneously, and handle situations such as timeouts, network connection problems and terminating connections. The Perl scripts then collect the information and return it to the GUI.

The GUI itself has additional features to help the operator debug remote systems such as:

- Double-click (left) on a hostname will open an xterm on that host.
- Single-click (right) on a hostname will bring up the most recent results from that host (or group of hosts) in an editor for viewing.
- A scrollable list of hosts is always available, showing the status of the last commands run, any output or errors that resulted from the last run, and the current known state of that host (unreachable, running a job, completed, etc.)
- Information on the host-type of the various systems.

The Code

Igor is written in TCL/Tk [1] (as a wish script) and Perl 4 [2], and the target end is entirely written in Perl 4. The only prerequisites for administrating a target with Igor is that the target have available to it Perl (version 4 or 5), and a host which it implicitly trusts (preferably a centrally located, tightly controlled host). The daemon which runs on the target system uses no Perl library code, and only requires that the interpreter be present. This means that all of the networking code is rolled into the daemon itself. This design was based on the decision that we wanted the daemon to rely on as little as possible to run. For example, if the perl library modules were not mounted, then we did not want the daemon disabled.

Igor works as an interactive tool, which makes it different from tools such as DSmit for AIX [5] and easier to setup than tools such as Systcl [3]. Igor allows a skilled operator to write shell scripts or perl scripts and have them executed very quickly.

The Controlling system's software consists of a GUI which is entirely written in *wish*(1) and a series of backend perl scripts. The GUI simply manages the list of hosts, current set of commands, current set of regular expression matches and some tunable preferences. The rest of the controlling system's software is a set of perl scripts that produce reports on the output data (and use the regular expression data to determine if the run was successful), setup socket connections to the remote hosts; rcp, rsh and ping the remote hosts to setup the Igor daemon,

The Target System

The target is any UNIX host which runs Perl and which trusts a central host (with /.rhosts or /etc/hosts.equiv). Igor runs as a daemon monitoring a pre-determined TCP/IP port. This daemon can either be started by the controlling system using a "spawn" function available to the operator or it can be started by conventional means, such as rc scripts.

The "spawn" function uses ping to contact the target host, rcp to move the Igord daemon script to the target system and then rsh to run the script. The script daemonizes itself and continues running in the background. The Igor daemon (Igord) then listens to the port and when a connection is made, forks and the child process receives commands from the controlling system. These commands can be shell scripts, perl scripts, or special built-in functions to send files to the

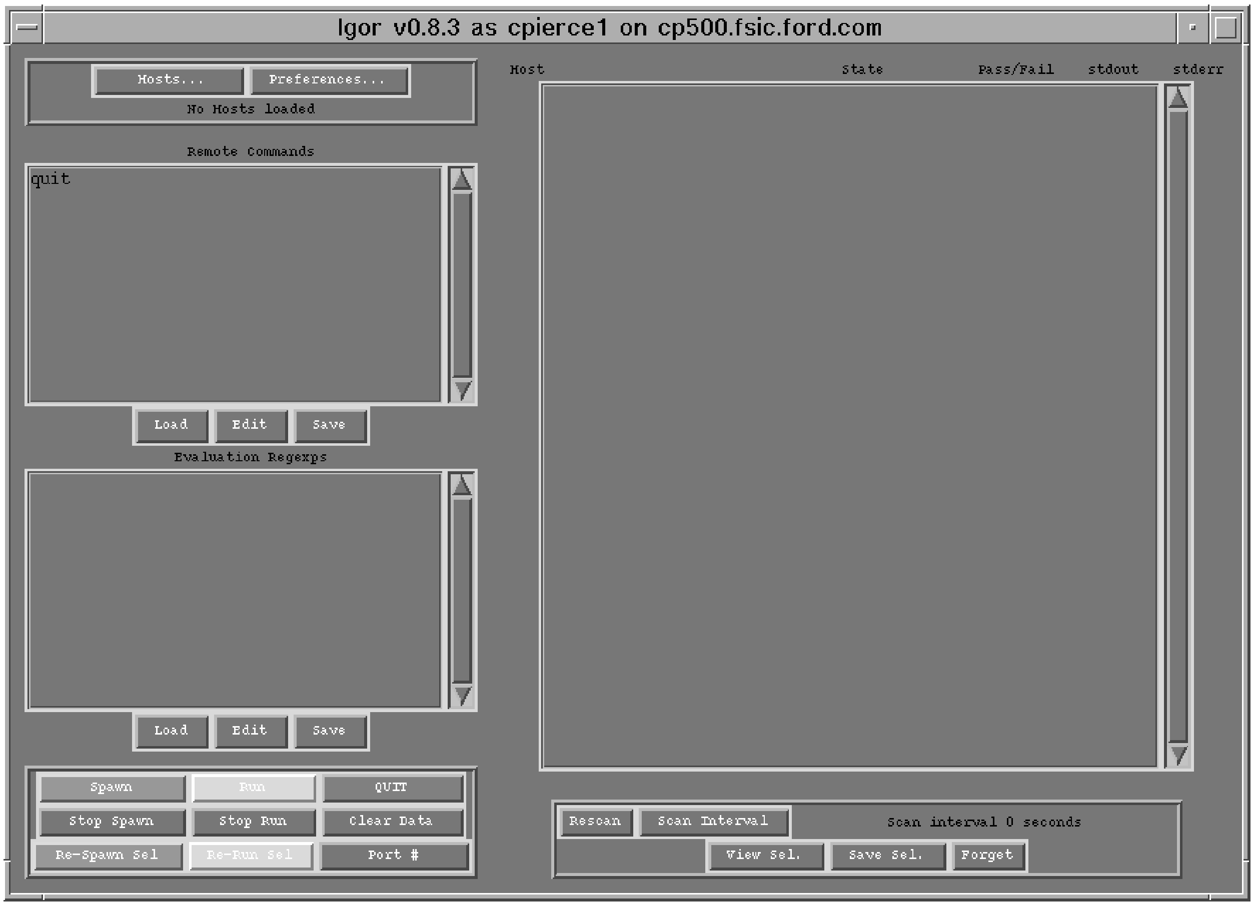


Figure 1: The graphic user interface

target. The Standard Output and Standard Error of all of the executed commands are carefully collected, and put into a boilerplate and sent back to the controlling system for analysis. The child then dies.

The preferred method of starting the daemon is to have the target system start it as part of it's initialization. This way, the daemon is always available to run commands, and does not have to be "re-spawned". Normally, at Ford, if we find a system that is not starting the daemon at boot time, we spawn a daemon on the host, and then run an Igor job to install itself on the target host, and start itself as part of the next boot.

The Controller

The operator runs Igor from a well-trusted host. This host should (if possible) have a very thick connection to the targets and should have as much CPU as you can spare; the more CPU and network, the more jobs you can run in parallel. Also, the controlling host should be able to open many sockets at once. Under certain OS's (Solaris) this requires a kernel tunable parameter to be set. The amount of resources used by the Controlling system is controlled with a "throttle" adjustable in a preferences dialog. The throttle controls how many remote hosts will be communicated with at any one time. Setting this number high uses more resources. On a Sparc-center 1000 with 1 CPU, a throttle limit of 40 will keep the load-average of the system near 10.

On the well-trusted (Controlling) host, the operator first loads in a list of hosts to operate on. This list is simply a flat-ASCII text file, one host per line, and are loaded with a point-and-click file browser [Figure 2].

Once the hosts are loaded, you can perform "run" or "spawn" operations on those hosts or a selected subset of those hosts. Spawn is used to start Igord on the remote hosts. The host is pinged, the script is rcp'd to the host and rsh is used to get it running. Traditional BSD-style network commands are used so that the target host is almost assured to have the necessary utilities already in place to start the daemon. Once the daemon is started, you generally do not have to restart it. If a daemon is already running, and the system is "respawned", then the new daemon will kill off the old one, and run in its place.

Once the "RUN" button is pushed the script that the operator has entered is transmitted to the remote systems, their output collected and sent back to the

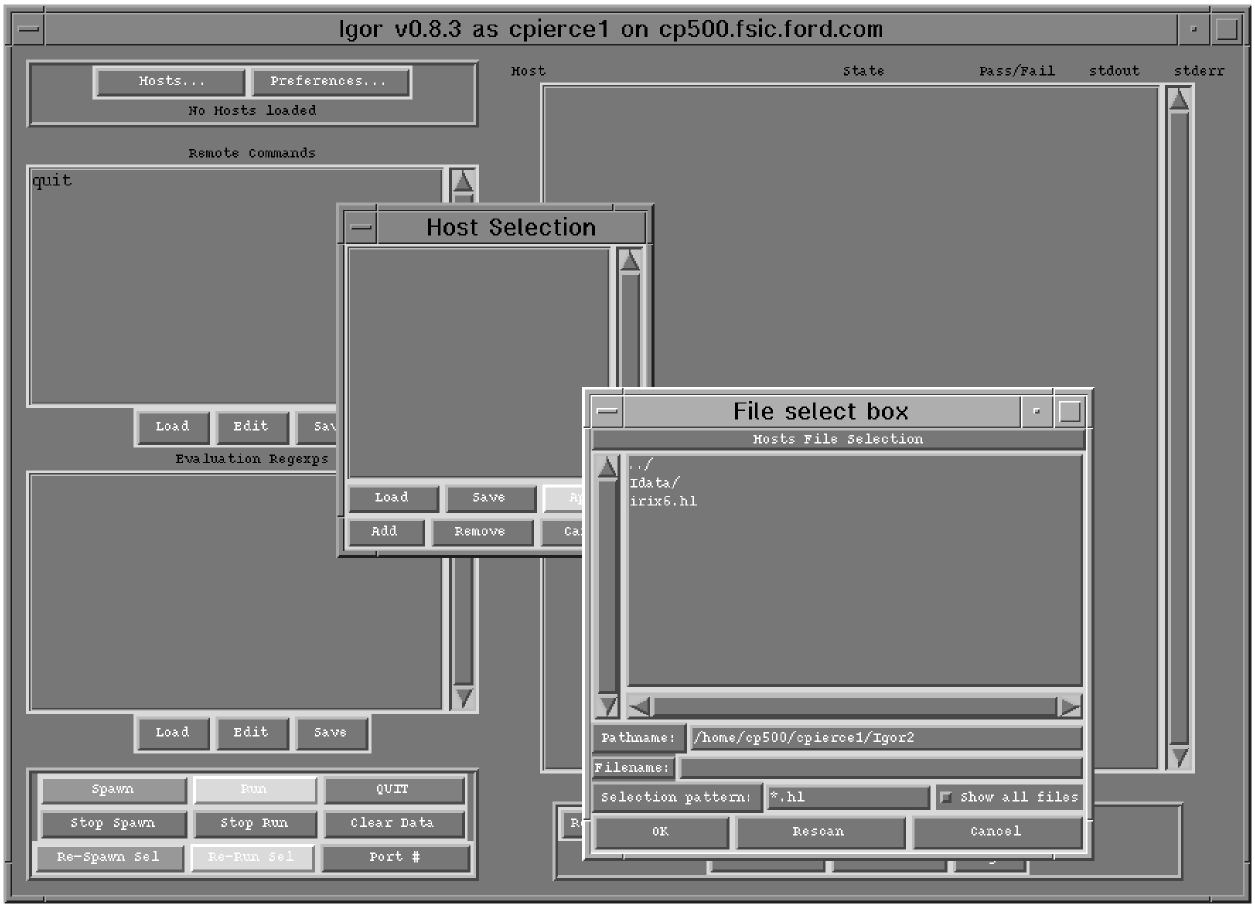


Figure 2: Point and click GUI

controlling system. For both Run and Spawn the GUI starts a back end perl script. That process forks as many times as needed to reach the "throttle" limit. Then each child takes a system name from the common hostname pool, and tries to contact the Igord on that host and execute the job. When the job is completed a particular host, the child grabs another hostname from the pool and starts again. Each of the communication processes will timeout if necessary, using a value set in the preferences dialog of the GUI, and then take another host from the pool.

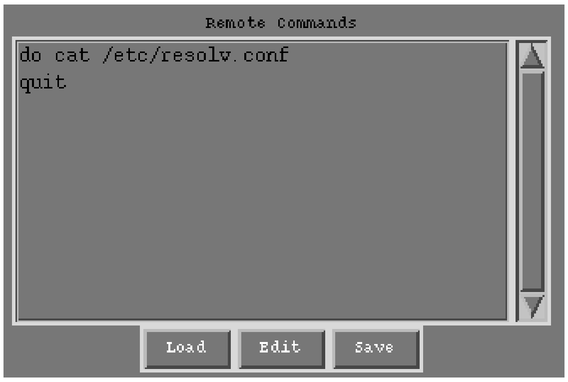


Figure 3: Script

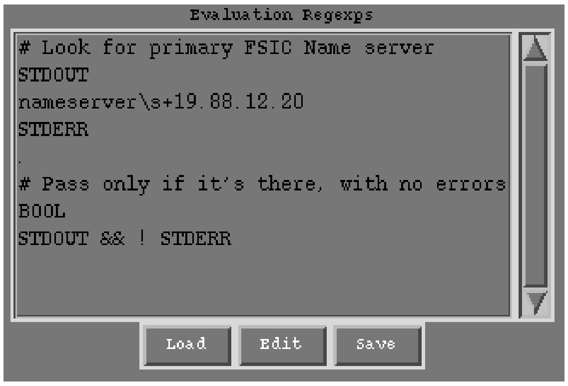


Figure 4: Regular expression list

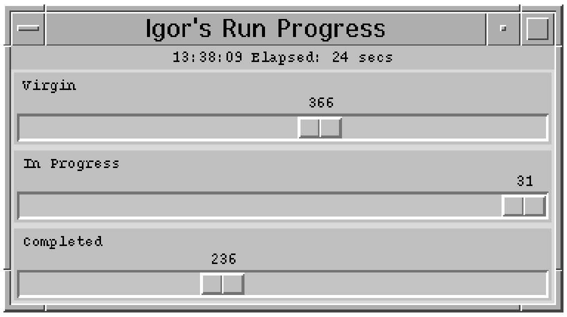


Figure 5: Progress indicator

The backend scripts and the GUI operate independently of each other. The GUI simply starts the backend scripts and then can retrieve their output by looking in a hard-wired subdirectory (./Idata) for results from each host. If the user requests another job be run while the first job is still running, that isn't a problem. Another set of backend scripts are started, and the results are left in the same subdirectory. If for some reason the GUI needs to communicate with the backend scripts (to abort a job, for example) either token files are left in a common area which the scripts look for occasionally, or a specialized perl script can communicate back and forth between the GUI and the backend scripts using other IPC mechanisms.

Controlling GUI Tour

The buttons on the GUI [Figure 1] do the following:

- "Hosts...": Opens the Host Selection dialog [Figure 10] and allows you to add/change/load the hosts to be worked on.
- "Preferences": Opens the User Preferences screen [Figure 11]. The fields are:
  - Throttle – Maximum number of hosts to work on at once.
  - Timeouts – How long to wait for any one host to respond to Igor's query. Once the host is contacted, the timeout is no longer in effect.
  - Editor Options – When "view selected" is picked for multiple hosts, this selects whether you want to see one host at a time (i.e., "vi hosta hostb hostc etc..") or all of the host data concatenated.
  - Voyeur Options – The Progress Indicator can be brought up automatically when the "Run" or "Spawn" buttons are pushed. Normally the Indicators are not shown.

  Preferences are stored between sessions in .igorrc
- "Spawn": Starts Igord on the entire set of hosts.
- "Run": Runs the current set of commands on the entire set of hosts.
- "Re-Spawn Sel": Starts Igord on the selected hosts
- "Re-Run Sel": Runs the current set of commands on the selected hosts.
- "Stop Spawn": Stops a spawn in progress. Any "spawn" that is currently being tried on a host is finished first.
- "Stop Run": Stops a run in progress. If a host is already active, the run is finished on that host.
- "Clear Data": Clears the ./Idata directory and removes all of Igor's information on its contacted hosts.
- "Port #": Multiple Igord's can be run on a system simultaneously. This allows you to control which one you're talking to.
- "Rescan": Retrieves the current set of data for each host, refresh the host status window, reapply the regular expressions to the output.

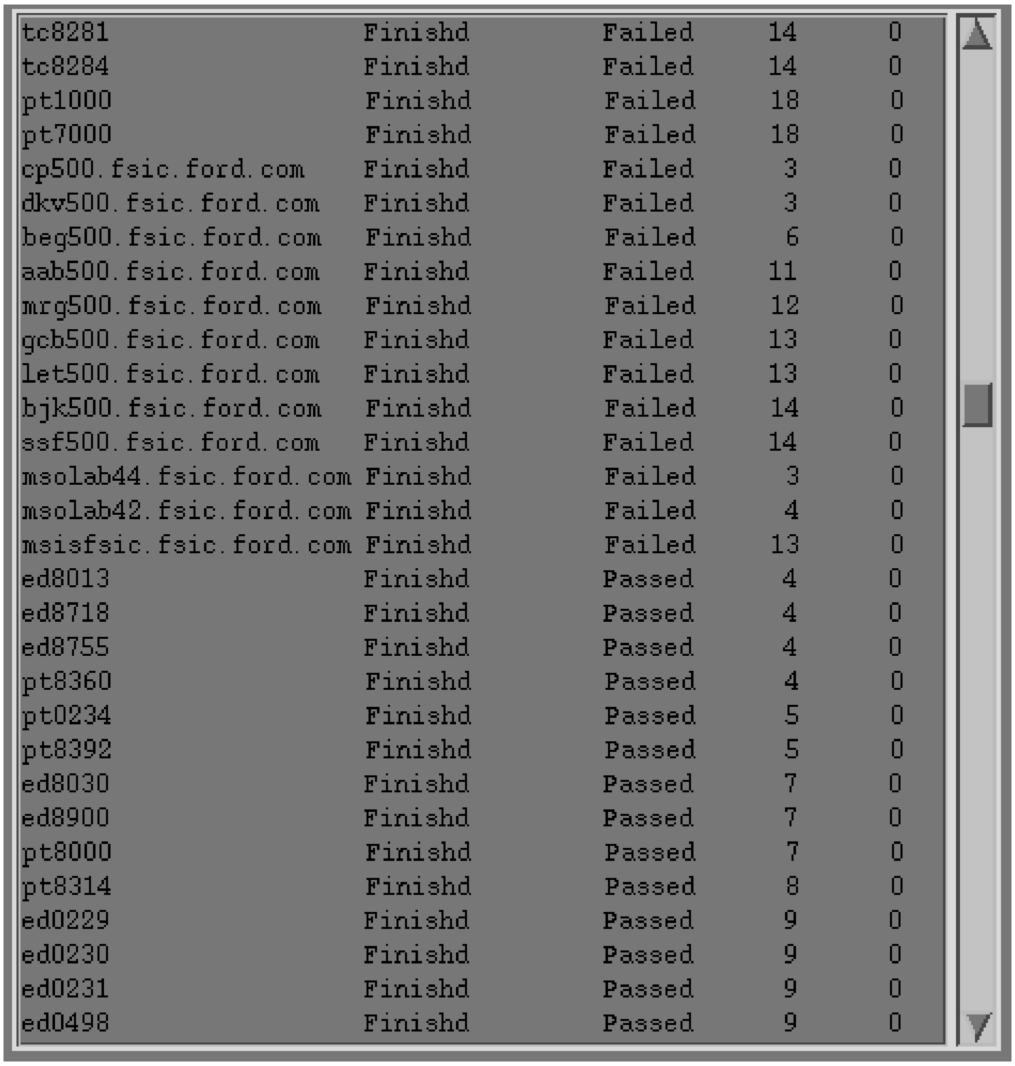
- “Scan Interval”: A rescan can be done at a regular interval. An interval of 0 stops the auto-rescan.
- “View Sel.”: Allows you to view the current data [Figure 9] retrieved for each host selected.
- “Save Sel.”: Will save the list of selected hosts to a file. This allows you to create lists of hosts split up based on the results obtained. For example, saving all of the hosts which fail a certain test so that they can be corrected later.
- “Forget”: Remove the selected hosts from the current host list.

Pressing the right mouse button in Igor will bring up another panel with additional buttons [Figure 12]:

- “Archtype Sel”: Shows the architecture type of the selected hosts.
- “Watch Run”: Pops up a Progress Indicator for each Run currently in progress.
- “Watch Spawn”: Pops up a Progress Indicator for each Spawn currently in progress.
- “Kill Spawn w/Prejudice”: Stops a Spawn immediately. Does not finish the hosts currently being worked on. This can leave the remote hosts half-done. (Daemon is there, but not running for example.
- “Kill Run w/Prejudice”: Stops a Run immediately. Does not finish the hosts currently being worked on. This can leave the hosts having executed only some (or none) of the commands sent to it. Still, if you make a mistake, this button is your friend.

Igor’s Security

Igor’s security is based on the BSD rexec(3N)-style security of ˜/.rhosts file for each operator they wish to trust, or a /etc/hosts.equiv file listing all of the hosts and users that they trust for Igor activity. The daemon, upon connect from a Controlling system will verify that the remote system is trusted. Having verified that, will accept



tc8281	Finishd	Failed	14	0
tc8284	Finishd	Failed	14	0
pt1000	Finishd	Failed	18	0
pt7000	Finishd	Failed	18	0
cp500.fsic.ford.com	Finishd	Failed	3	0
dkv500.fsic.ford.com	Finishd	Failed	3	0
beg500.fsic.ford.com	Finishd	Failed	6	0
aab500.fsic.ford.com	Finishd	Failed	11	0
mrg500.fsic.ford.com	Finishd	Failed	12	0
gcb500.fsic.ford.com	Finishd	Failed	13	0
let500.fsic.ford.com	Finishd	Failed	13	0
bjk500.fsic.ford.com	Finishd	Failed	14	0
ssf500.fsic.ford.com	Finishd	Failed	14	0
msolab44.fsic.ford.com	Finishd	Failed	3	0
msolab42.fsic.ford.com	Finishd	Failed	4	0
msisfsic.fsic.ford.com	Finishd	Failed	13	0
ed8013	Finishd	Passed	4	0
ed8718	Finishd	Passed	4	0
ed8755	Finishd	Passed	4	0
pt8360	Finishd	Passed	4	0
pt0234	Finishd	Passed	5	0
pt8392	Finishd	Passed	5	0
ed8030	Finishd	Passed	7	0
ed8900	Finishd	Passed	7	0
pt8000	Finishd	Passed	7	0
pt8314	Finishd	Passed	8	0
ed0229	Finishd	Passed	9	0
ed0230	Finishd	Passed	9	0
ed0231	Finishd	Passed	9	0
ed0498	Finishd	Passed	9	0

Figure 6: Results of running against various hosts

commands from the Controlling system. If the trust check fails, no commands are accepted and an error message is printed on the socket. Please note that the distributed version of Igor does not use a "trusted" port, and is for experimentation only. Simply changing the port usage on the daemon and the GUI will make Igor use a trusted port and a little more secure.

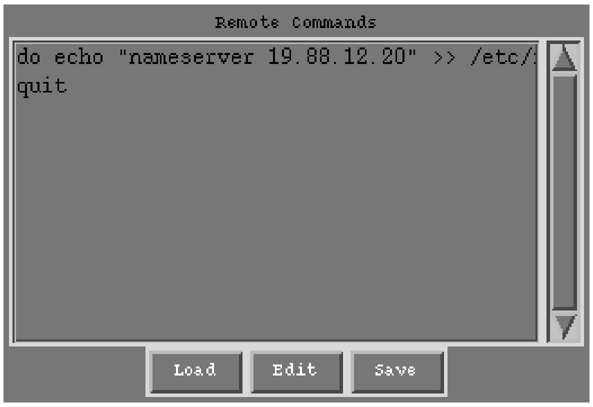


Figure 7: New set of commands

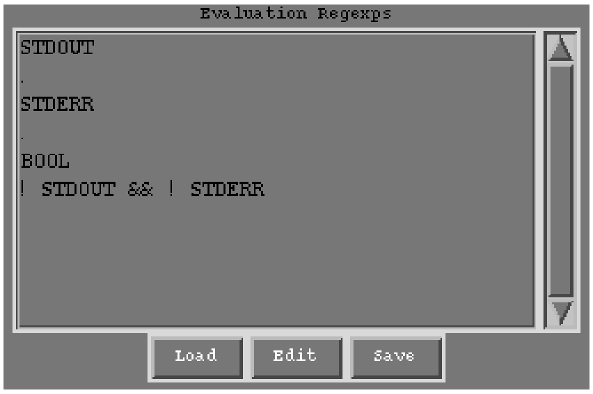


Figure 8: New set of regular expressions

This security, although old and not state-of-the-art is no less secure than what is used for rsh. Internally to Ford, this is generally adequate. Each workstation trusts a centrally located server, and our network security is handled by third party sources. Because we trust our network services and the centrally administered host, rexec security is adequate for our purposes..

There are certainly other ways to make Igor more secure. For example, using a PGP encrypted copy of the script to transmit to the remote daemons. The operator at the GUI could be queried for the encryption key. The decryption key could be located by querying another (or the same) host and having obtained the key, you could decrypt the Igor commands, ensuring that they came from the correct host.

Igor's code is fairly straightforward, and could be easily changed to accept these modifications.

Igor Scripts

The scripts that Igor runs are nothing more than a way of wrapping up shell scripts, tar files, and simple shell commands so that Igor can make sense of them at the remote side and run them. The various commands are:

- **do** *args* – Run args as a shell command (/bin/sh). Eventually, everything except the "do" is passed to a perl script (on the remote end) and run as a "system" command. Normal shell argument parsing will take place on the remote end. This is the most commonly used Igor command.
- **EVAL** *args* – Run "args" as Perl commands. This can be used to run perl commands directly by the remote daemon. Another use is to add functionality, on the fly, to the daemon by having it "eval" new functions. It can be used to add timeout capability to various Igor commands ("do"). See examples below. This is generally safe to use, because the daemon that's being modified with EVAL is simply a child of the daemon listening to the port on the remote system. Any potential defects in the child daemon do not affect its parent.
- **openfile** *file mode* – Open file as a "here" file with mode specified. This is one method of transmitting lengthy shell scripts to the remote system. Binary data being sent must be uuencoded because the Igor "script" exists for a while in a TCL list, which can't contain binary data. Other ways of transmitting scripts, patches, programs, etc... include using an NFS mount publicly exported Read Only from a common system, retrieving the data through an ftp script, or accessing it from a webserver with a URL. (There is a short Igor script using EVAL which can enable Igor to do HTML retrievals.)
- **closefile** – End of openfile block.
- **id** – Igor will report its version number, local hostname, remote hostname (controller), date, time, local system architecture type, and other useful information.
- **quit** – Terminate the remote Igord, transmit results. This command is REQUIRED at the end of a script, and will be inserted if you do not use one.

The commands are given to the GUI by pressing the "Edit" button in the Remote Commands window and using your favorite editor to enter commands. Pre-assembled lists of commands can be loaded using the "Load" button. Scripts are saved with a common file extension (.cmds) to distinguish them from other files.

Script Examples

To transmit a small shell script and run it:

```
openfile /tmp/fixbugs.sh 0755
#!/bin/sh
echo "Then a miracle occurs here"
install_miracle_patch
closefile
do /tmp/fixbugs.sh
do rm /tmp/fixbugs.sh
quit
```

To check disk space in / and /tmp:

```
do df /tmp /
quit
```

Using the EVAL function, some additional functionality can be added to scripts:

```
EVAL sub timeout { next MAINLOOP; }
SIG{'ALRM'}='timeout'; alarm(10);
do function_that_may_hang
EVAL alarm(0);
quit
```

This adds a timeout to the "function_that_may_hang". If the program doesn't return, Igor catches an alarm signal and continues executing the script after the questionable function. This required some knowledge if the innards of Igor, but these tricks are well documented.

Output Analysis

One of Igor's most important features is analyzing the output as it comes back from the remote system. In the GUI, each system is shown, with a count of the number of lines of STDOUT and STDERR reported. Sometimes this is enough to tell if everything worked OK. Also in that window is a field labeled "Pass/Fail". This field can also be used to tag each system with a Passed/Failed status. That is done by using the Regular Expression matcher.

This area takes input in the form:

```
STDOUT
Exp1
Exp2
Expn
STDERR
Exp1
Exp2
Expn
BOOL
Boolean Expression
```

The Expressions are Perlish regular expressions (without the //'s). Slashes and special characters must be quoted. These expressions are matched against successive lines of Standard Output or Standard Error and so long as the expressions match, the associated tokens (STDOUT, STDERR) will evaluate to true. If a regex does not match, the token gets set to false. The regular expressions are associated with the token they follow. The BOOL token indicates that the next line will contain an expression that will evaluate to true or false. The Boolean Expression is a perlish thing that is going to get EVAL'd. "STDERR" gets substituted with 1 for a match and 0 for a nonmatch "STDOUT" gets substituted with 1 for a match and 0 for a nonmatch. Depending on the outcome, the system will be marked as "passed" or "failed" in the status screen.

```
vi
-------------------------tc1194---------------------------
==================REMOTE OUTPUT=====================
---Opening connection
---Getting ID info
ID 0 0 V:2.0 00 17113 Tue Aug 13 13:45:03 EDT 1996 tc1194 solaris ed8030.pto.for
d.com
---Version=2.0
---Transmitting script
---Waiting for reply
START STANDARD ERROR
END STANDARD ERROR
START STANDARD OUTPUT
; $Id: resolv.conf,v 1.1 1994/02/28 20:34:58 jbell4 Exp $
;
; $Log: resolv.conf,v $
; Revision 1.1  1994/02/28  20:34:58  jbell4
; Initial revision
;
;
; Real resolver file for Domain Name Service (DNS)
; domain is pto.ford.com
domain pto.ford.com
search pto.ford.com ford.com
nameserver 19.50.8.20
nameserver 19.3.230.20
nameserver 19.52.6.100
END STANDARD OUTPUT
---Reply finished
~
~
```

Figure 9: Sample of retrieved data

This sounds complicated, but in practice is a rather simple way of checking output. For example, to consider all systems that report something on STDOUT and nothing on STDERR as ''Passed''. you could use this arrangement:

```
STDOUT
.
STDERR
.
BOOL
STDOUT && !STDERR
```

STDOUT gets set to 1 (true) if any single character is matched. STDERR gets set to 1 (true) if there's any STDERR output. If the BOOL expression evaluates to true, the system is tagged as ''Passed'', otherwise ''Failed''. Something more complicated could be used like this:

```
STDOUT
9[1-9]%
STDERR
.
BOOL
!(STDOUT || STDERR)
```

For the set of commands:

```
do df
```

Would report ''Failed'' if the ''df'' command reported any filesystem more than 90% full, or ''df'' reported anything on STDERR.

There's two other special tokens that can be used in the Boolean expression in addition to STDERR and STDOUT. These are STDERRCNT and STDOUTCNT. These represent the number of lines of output on each file descriptor. For example:

```
STDERR
.
BOOL
( STDOUTCNT > 3 ) && ( ! STDERR )
```

This would return true (passed) if there were more than three lines on stout, and nothing on stderr. This would be useful if the expected output could have a variable number of lines, but no errors should be expected.

The Pass/Fail indicators shown in the hostlists are generated every time the host list is displayed. So if you decide that a different pass/fail criteria is necessary for your hosts, you can change the regular expressions and rescan the host list. You do not need to re-run the commands on the remote hosts.

Sample Walkthrough

To demonstrate Igor's true usefulness, what follows is a walkthrough of a sample Igor session. The hypothetical problem will be adding a resolver to each workstation's /etc/resolv.conf. The operator would first login to a trusted host, and start the GUI [Figure 1]. Next the operator can load the list of hosts to be operated on, or can enter them in [Figure 2]. First, in order to make our scripting a little easier, to find out which hosts already have the correct resolver we'll use the script shown in Figure 3. The regular expressions list is shown in Figure 4. These are loaded from a file browser.

Figure 10: Host selection dialog

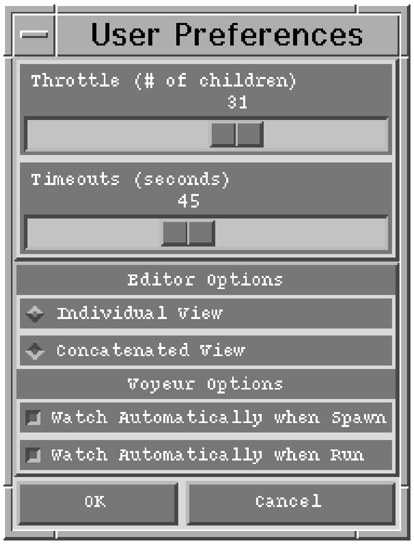


Figure 11: User preferences screen

Now we're ready to contact the hosts. Clicking on the ''Run'' button will cause Igor to contact the remote hosts, and end them the script to be run. A progress indicator [Figure 5] lets the operator know how many hosts are untouched, being-worked-on, or have completed the commands. When the sliders indicate everything is done (or even before that) the operator an click on ''Refresh'' and that will cause the RE's to be run against the results obtained. The results are shown in Figure 6. Some hosts passed OK (they already have the resolver in the file) some did not. The hosts which pass can be selected with the mouse, and then the ''forget'' button pressed.

This will drop these hosts from the host list. We do this, because only the hosts that need work are left in the list.

The remaining hosts are left onscreen. We can then load in a new set of commands [Figure 7], and a new set of regular expressions [Figure 8]. These will actually add the new resolver into the resolv.conf file. Pressing ‘‘Run’’ will cause the progress indicators to reappear, and when Igor is all done, we can see which hosts were modified successfully, and which were not.

If problems appear during the run, there’s quite a few things that can be done to diagnose what happened. To actually look at the data retrieved from the hosts by selecting the hosts we’re interested in, and then clicking ‘‘View Selected’’. The raw data retrieved from the remote host is shown in a vi session. A sample of the retrieved data is shown in Figure 9. The initial information shows the connection being established, the standard output and standard error are shown, separately. From this information you might be able to tell what’s wrong on the host.

If more diagnostics (or repairs) are needed to individual hosts, the operator can double-clicks on a host in the list. An Xterm will open (running ‘‘rsh host’’) so that he can check things out manually. If a large number of hosts failed, the operator can rewrite the script and try it again.

Cautions

If Perl is the ‘‘Swiss Army Chainsaw’’ of UNIX, then Igor is a Gatling Gun loaded with Swiss Army Chainsaws – a useful tool or a terrible weapon of destruction. Igor is the fastest way we know of to fix problems on our 700 hosts – it’s also the fastest way to cause them. It should probably not be used by anyone who doesn’t understand how it works. For example, if you were to mistype a Igor script to do ‘‘/bin/rm -rf /tmp/*’’ and had typed ‘‘/bin/rm -rf /tmp /*’’ then ALL of your systems would be instantly erased. This is a normal pitfall for system administrators, it’s only magnified with Igor.

The thought of implementing Operator safety features to Igor has entered our minds (‘‘Are you sure?’’ type questions, etc...) and then swiftly left. One of the most powerful features of Igor is the fact that it doesn’t get in your way. Once the structure of the commands are learned (there are only 4) and you’ve collected enough post-processing templates Igor just lets you do whatever is necessary...quickly. No fuss, no muss.

Author Information

Clinton Pierce is a System Administrator for Decision Consultants, Inc. and is currently assigned to Ford Motor Company. He is currently involved with integrating Solaris, AIX and IRIX workstations into a common look-and-feel environment. In addition, he teaches UNIX and Perl to consultants for DCI. Clinton can be reached via e-mail at cpierce1@ford.com, or U.S. Mail at Ford Systems Integration Center, 1000 Republic Drive, Suite 600, Allen Park MI 48101.

Figure 12: More buttons

Bibliography

[1] Welch, Brent 1995 Practical Programming in Tcl and Tk Prentice Hall, Englewood Cliffs, NJ.

[2] Wall, Larry, and Schwartz, Randal L. 1991 Programming Perl O'Reilly & Associates, Sebastopol, CA.

[3] Lombardi, Christine, and Desimone, Salatore 1993 "Systcl," Proceedings of the 1993 USENIX LISA Conference pp. 133, Monterey, CA.

[4] Stevens, Richard W. 1992 Advanced Programming in the UNIX Environment. Addison-Wesley, Reading, Mass.

[5] AIX DSMIT Guide and Reference Version 2.2 1994, Pub. Number SC23-2667