Invocatables: Large-Scale Application Support Using Wrapper Scripts

Brian Town and Dan Krantz

Agilent Technologies

24001 E Mission Ave

Liberty Lake, WA 99019

509-921-3871 (W) 509-921-4300 (FAX)

brian_town@agilent.com dan_krantz@agilent.com

Introduction

At Interworks 98, we gave a presentation entitled "Rebuilding a UNIX Infrastructure" in which we talked about a world class Unix environment for a division of what used to be Hewlett Packard (now Agilent Technologies). We shared our philosophy on how to migrate 600 systems from a disjoint group of HP-UX 9.05 machines to a carefully engineered architecture based on HP-UX 10.20. Our top priority was to improve the reliability of the infrastructure.

Recognizing that an infrastructure's reliability is visible to the end-users through the application layer, one of our essential goals was to provide a consistent application environment: Every installed version of every supported tool would be available to every user on any workstation regardless of discipline or functional area. With over 100 commercial application suites and a diverse end-user population, our solution needed to be robust, proactive and transparent. Invocatables (wrapper scripts) were chosen as a reliable and extendible methodology for achieving these objectives.

Objectives - The Vision Behind the Implementation

In keeping with the theme of our HP-UX 10.2 infrastructure project, our approach to managing applications follows the same rules of transparency, functionality, consistency, pro-activity and centralized management. We developed a strategy to accommodate existing tool usage patterns and enhance end user control over application invocations while minimizing support overhead.

The vision was to create a thin layer (i.e. onionskin) around each individual application suite installed on our system. This would take the form of shell scripts providing run time binding of all necessary PATH and variable setups. The user's environment was to be completely devoid of application configuration data. By moving the environment configuration to run time, we were able to meet and exceed our objectives.

Objective #1: Transparency

End users rely upon the documentation shipped with applications. Creating a solution that interrupts, alters or otherwise causes deviation from the documented use model for invoking a tool will not be received well by end users. An invocatable's presence in the system should be transparent to users apart from the added value and control they offer. Therefore, if a user can just type "netscape" at the command-line to launch the application, they should be allowed to continue this action.

Objective #2: Functionality

The ability to simultaneously support multiple versions of application suites is a common requirement for a number of reasons. New tool versions often require extensive testing before full deployment. Sometimes we invite a "beta" group of end-users to try out the new version before we advertise its availability to the entire site. There are also business requirements and schedule dependencies that impact tool transition timing. Users with multiple project assignments encounter the need to flip back and forth between two or more versions of their application. In this case invocatables are called upon to simplify the otherwise cumbersome issue of switching between different versions.

When supporting a large number of application suites, there are bound to be some conflicts which arise between them. One area of conflict we have experienced has been with the appropriate settings for environment variables. It is not uncommon for two tools to require different (and mutually exclusive) settings for environment variables. Using invocatables affords the opportunity to delay the setting of environment variables until actual application run-time. Since each invocatable can set the variables just prior to launching the tool, this eliminates the conflict.

Objective #3: Consistency

Nothing generates more headaches and calls for help than inconsistencies. In the area of application support, inconsistencies arise when applications are set up on a system-by-system, workgroup-by-workgroup or user-by-user basis. Users often rely on their peers as an informal first line of support. When something works for one user but not for another, you have lost an opportunity to reduce support calls. It is very difficult to predict exactly who will eventually require the use of a particular application. When you add the movement of users between projects or across disciplines, the task becomes almost impossible. This reasoning caused us to adopt the philosophy of making all applications available on all systems for all users.

As you know, most Unix applications require the definition of application-specific data, such as environment variables and paths. The typical approach is to add this data to individual profiles through the well-known mechanisms found in .login, .kshrc, .profile, etc. Since we want to make all applications available to all users, just imagine the baggage each user would carry in these files. Such a method is begging for inconsistencies. Instead, we localize application configuration data within the invocatables themselves. The application-specific environment is created on the fly at run-time! We have successfully eliminated the opportunity for inconsistencies to rear their ugly head and dramatically reduced the calls for help.

Objective #4: Pro-Activity

Detecting problems before they occur (and correcting them) is what pro-activity is all about. If a conflict can be identified before an end user discovers it, then a thoughtful solution can be crafted which will eliminate a fire drill from occurring later. A predictable conflict often experienced is when two vendors use the same exact name for an executable. An invocatable strategy can provide visibility to these conflicts during the process of installation and configuration. Once identified, the administrator can determine the proper handling of the collision and even codify necessary corrective actions inside the invocatable.

Perhaps the largest opportunity for pro-activity comes from the ability to monitor and report application errors to an administrator **before** the user picks up the phone to report the problem. Again, invocatables afford an administrator the opportunity to identify and report (via email) potential problems as well as actual application run time errors. This advanced information gives administrators a heads up to problems occurring with an application. Imagine the opportunity for surprise when an administrator calls an end user to tell them that the problem they just experienced (but have not yet reported) has been corrected! Early detection and correction of a problem also shields other users from experiencing any related downtime.

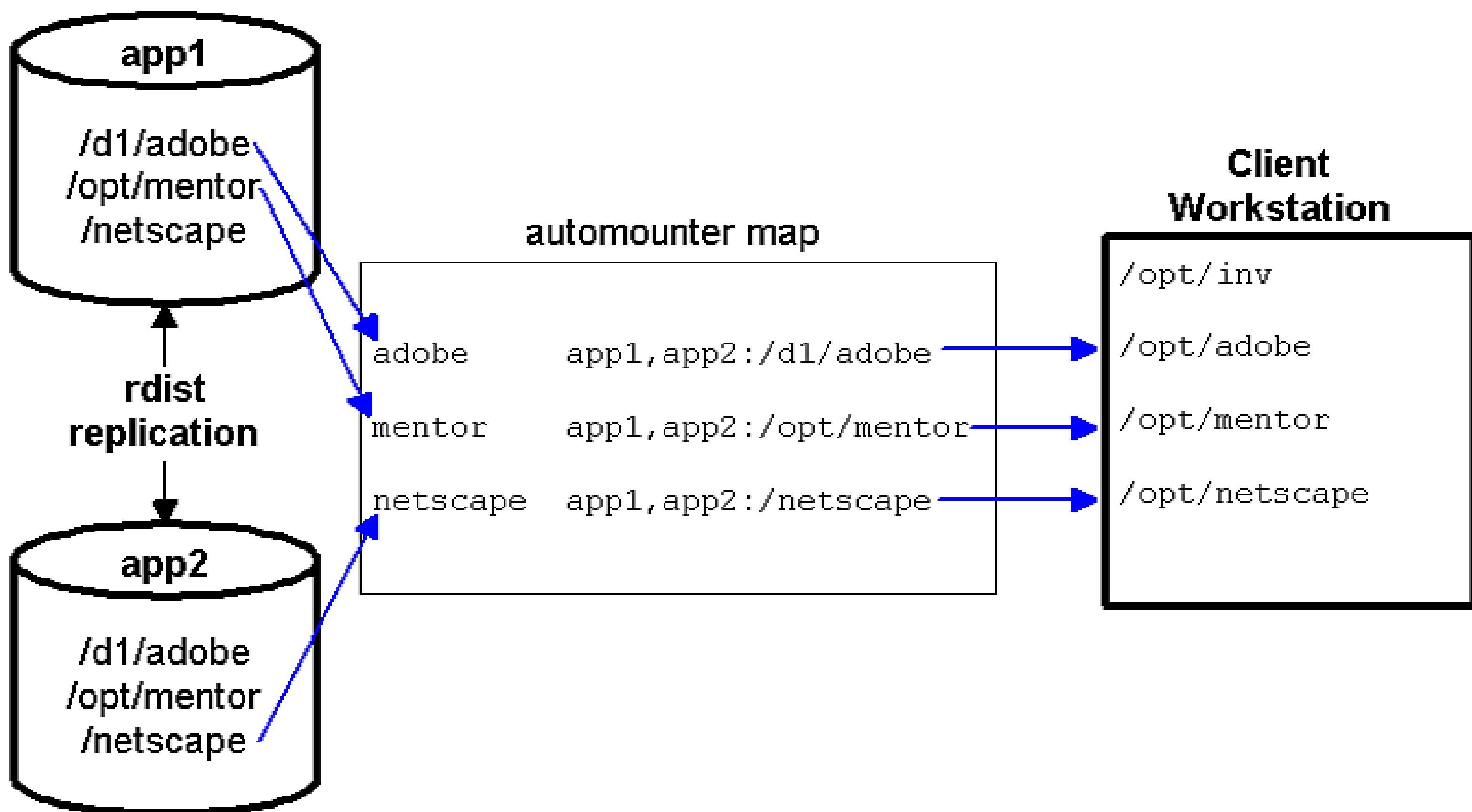
Objective #5: Centralized Management

Invocatables align with our centralized management philosophy because they concentrate application support tasks around a single script. We have eliminated the need for hard coding application dependent environment variables and paths into profiles and system boot scripts. The idea is to create a sort of "one stop shopping" for application support within an invocatable.

Logging and metric gathering is a common theme in any centralized management scheme. Because all application startups run via an invocatable, this is a natural place to add a logging facility to record rudimentary run time information. This data can be invaluable for identifying users, usage patterns and statistics.

Foundation: The Enabler of the Implementation

There are really two distinct pieces to the application deployment puzzle. First, the actual application bits must be made available to the client workstations. Once this is accomplished the appropriate environment variables and paths can be setup and the application launched, but we'll get to that part later. For now, let's examine the components of a robust application infrastructure by looking at the following diagram.



NOTE: Although not shown above, unique version directories exist under each of the /opt/<application> directories. Version directories allow multiple revisions of each application to exist simultaneously. These version directories follow the form:

```
/opt/<application>/<version>/
```

The first thing to note about the application infrastructure is that we load the application binaries, libraries and documentation on high availability servers rather than on individual workstations. To install the latest version of your favorite app, simply place it on the app server and immediately hundreds of workstations have access to it. Better yet, workstations that were off-line at the time of your install will see the new version as soon as they come back on-line. You may be thinking to yourself, "But the reverse logic will cost me my job! What happens when the app server dies? If I can't get it going for several hours, hundreds of expensive, idle engineers will develop all sorts of high-tech torture gadgets *just* for me."

Indeed this is a risk and a very important one to prevent. First, you should bolster the individual server. We have been extremely pleased with the HP Model 12H AutoRAID. Configured with an Active Hot Spare, drive failures are a nonevent for the server (and more importantly, the client workstations). Upon detection of a failed disk, the system automatically switches over to using the Active Hot Spare. Simply swap the failed mechanism with a new spare module at your earliest convenience. Redundant power supplies and redundant fans also contribute to hardware uptime. You can continue to increase the availability of the server by utilizing the proper software. We have chosen the built-in Logical Volume Manager and the valuable OnLine JFS. With OnLine JFS we can de-fragment file systems, increase logical volumes, decrease logical volumes and take file system snapshots all while the data is being accessed by client workstations.

Even after you add this armor to the basket, you still have one basket holding all your eggs. That?s why we?ve deployed redundant application servers: app1 is the master and app2 is the slave. We utilize the standard HP-UX program `rdist(1)` to replicate files from the master to the slave. A simple command-line interface

allows support staff to interactively replicate newly installed apps. A nightly cron job maintains server synchronization even if the admins forget to manually do the work, not that any of YOU would ever forget a post-install procedure!

There is one side benefit to having dual app servers: load balancing. Our two D390/2 Enterprise Servers, each with an AutoRAID and 100Mbps switched LAN connections, have easily supported over 180 versions of 100 distinct commercial application suites representing 60 GB of data for nearly 700 clients. As shown in the diagram above, we utilize an automounter map to provided dynamic load balancing. The map allows us to specify both application servers. When a client workstation needs to mount the application directory, its automounter daemon first "pings" both servers and selects the first one to answer. We have configured automounter to unmount inactive directories that have been idle for more than 30 minutes so that workstations have a chance to pick a new app server.

The automounter map also provides location transparency. End-users and the invocatables they use do not need to know application server names or directory structures. We embed this knowledge in the automounter map and simply advertise to the world that all applications are available in the /opt directory. Now that the application software is made continually, ubiquitously available to the client workstations, we can move on to the task of on-the-fly run-time environment setup. This is where invocatables enter the picture.

Implementation - From Vision to Reality

Having presented our objectives for invocatables and the foundation that delivers the application binaries to the client workstations, we may now move on to the functional description of our invocatables. We will describe the various components and functions of the invocatables without getting bogged down in all of the details of our specific implementation of these concepts.

The PATH to Invocatables

Most software vendors instruct you (as part of their installation process) to add the directory containing their binaries to your PATH variable. This methodology can lead to some extremely long PATH strings. Since invocatables set the PATH at run time, this problem is avoided.

So if the path to the application binaries isn't in the PATH, what is? The path to the directory that contains the invocatables of course! A single invocatable directory is added to every user's PATH. This directory contains an executable for **every** possible command provided by the applications installed on the system. Now there is actually only one invocatable per application suite, so the same invocatable is called for each command available for that suite. This is accomplished by creating a link to the real invocatable for each command provided by an application. This will be easier to understand by way of example.

First we configure all systems on site so that every user has the invocatable directory at the head of their PATHs:

PATH=**/opt/inv**:/usr/dt/bin:/usr/bin:/usr/bin/X11:...

Note the invocatable "switchboard" directory, /opt/inv. This is where we find a link for every possible command provided by all application suites. But there is only one invocatable script per application, and we locate these scripts in a "source" directory, /opt/inv_src. Both of these directories are located on every client workstation rather than being NFS-mounted from the app servers. With /opt/inv as the first entry in the PATH, even simple Unix commands like /usr/bin/ls would result in NFS traffic if /opt/inv were NFS-mounted. Furthermore, by placing the invocatable locally, it can detect stale NFS mounts and present the end-user with an intelligent warning with instructions on who to contact for assistance. Here is a sample of what you might find in /opt/inv_src:

```
/opt/inv_src/acrobat.inv
/opt/inv_src/actel.inv
/opt/inv_src/mentor.inv
/opt/inv_src/netscape.inv
/opt/inv_src/informix.inv
/opt/inv_src/maker.inv
```

Now for the case of Informix, here are a few of the symbolic links used to support it's invocation:

```
/opt/inv/dbaccess ->  /opt/inv_src/informix.inv
/opt/inv/dbexport ->  /opt/inv_src/informix.inv
/opt/inv/dbimport ->  /opt/inv_src/informix.inv
/opt/inv/dbload   ->  /opt/inv_src/informix.inv
```

You can see that when a user invokes a command like `dbimport`, the informix.inv invocatable is what actually gets run. The invocatable sets up the appropriate variables and paths required by Informix and then calls the actual Informix executable (`$INFORMIXDIR/bin/dbimport` in this case). The invocatable knows to run the `dbimport` command because the Posix shell sets $0 to the name of the command that was called by the user. The invocatable simply extracts the name from the $0 parameter allowing the same script to serve every command available for the application.

Now the thought of manually creating a link for every possible executable for each application seems just a bit overwhelming, at least we thought it was. That's why we don't do it manually. Instead, we create the list of links programmatically by looking for executables in the application's bin directory(s) when it's deployed. We then use OpenView Software Distributor (OV-SD) to push the actual invocatable and the appropriate links to each client machine. For our site, we currently have a grand total of 1,319 links that represent the superset of all commands available for our installed application suites - All this for the price of only one entry in the PATH variable.

One final point worth mentioning related to the creation of the links is the fact that there is nothing keeping you from declaring one or more executable links of your own design. This gives the invocatable author the ability to create what we might call *synthetic* commands. These are commands that will not be found in the vendor's bin directory. Instead, these are commands of the invocatable author's own making. When the invocatable detects (via a case statement on the $0 parameter) that one of these synthetic commands has been called, it runs a custom portion of the script to perform the desired task rather than simply calling an executable from the vendor's bin directory. Here are some examples of commands we have created via these links:

`spw_docs`	Launches a tool-specific documentation viewer.
`mentor_docs`	Launches the Adobe Acrobat reader for reading supplemental documentation that we receive from a vendor (Mentor Graphics)
`startWindRiverLicenses`	Gives administrators a single command for starting a vendor-specific license daemon.

Prior to implementing our invocatable strategy, this type of functionality could only be achieved by creating a myriad of small scripts off in some bin directory. Once again, the invocatable solution offers the ability to consolidate support for an application within a single file.

Command Line Options

All of our invocatables support a standard set of command line options. These provide support for a variety of end-user options as well as ones required for administration of the invocatable environment. In keeping with our consistency objective, we wanted to ensure that our command line options would have little chance of conflicting with the command line options supported by the application vendors. Since most options are preceded by a dash, we decided to start all of our command line options with at plus ("+") sign. Here is a brief description of each of the command line options we currently support.

`+env`	Export the environment variables and then exit. This is very useful for troubleshooting because it allows you to "source" an invocatable's environment: `". dbimport +env"`. Running this will cause all of the application configuration variables and paths to be set in the current shell.
`+help`	Display usage, showing a list of all + options.
`+listvers`	Display version information. All invocatables echo their list of application binary directories when they receive this command. OV-SD searches these paths when generating its list of /opt/inv links that need to be created.
`+prefs`	Run the app preference setting GUI. Use this option to set the desired version of an application as well as any other selections provided by the invocatable author.
`+term`	Run the app in a new terminal window. A new terminal window is spawned and the application is run from that window.
`+vars`	Echo variable settings. Shows all application specific environment variables set by this invocatable.
`+version <version>`	Specify application version to run.

Each invocatable sources a common set of library functions which parse these options into environment variables. The invocatables contain tests and branches based on the settings of these variables.

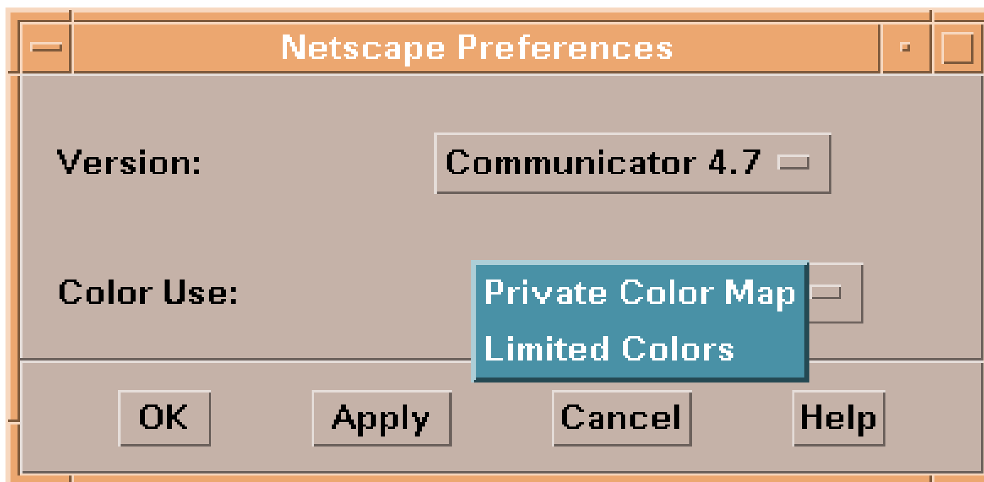
Multiple Version Support and Other Application Preferences

The fact that an invocatable sits between the user and the application means that any number of run-time tests can be performed to control the final application environment settings. Support for multiple application versions is the most common example of this run-time ability. In our implementation, we use a variable within the invocatable to control a case statement that sets the root directory of the desired (version-specific) application tree. The user is given three ways to control which version of an application they run:

1. By setting a variable in their current shell environment: `export INFORMIX_VERSION=7.3`
2. From the command line: `dbimport +version 7.3`
3. By launching a simple GUI to set what we call *Application Preferences*: `dbimport +prefs`

At run time, the invocatable determines which version of the application should be run and sets the environment accordingly, prior to launching the real executable.

While version selection is no doubt the most common usage of the Application Preferences functionality, it is by no means the only one. In the Netscape invocatable, we use a preference setting to give our users GUI based control over the behavior of this X11 color hogging application.



Individual user preferences are stored and maintained by the Application Preferences GUI in end-user home directories, `$HOME/.dt/preferences`. The invocatable calls a standard library function in order to extract the current setting of a preference at run time. This information is used to appropriately set environment variables or command line options to exert the desired control over the target application.

Trapping and Reporting Errors

As stated in the fourth objective (pro-activity), invocatables can be used to trap and report errors. There are two categories of errors that our invocatables can manage in this way. The first group consists of errors that can be identified prior to actually launching the application. Our invocatables can test for and report the following error conditions if appropriate:

- Path to application tree not accessible (e.g. an NFS mount may be down).
- Insufficient hardware to run this application
- Invalid operating system for this application/version
- Insufficient number of free X11 color cells to run application
- Another instance of this application is already running

Each of these tests is accomplished by a call to a function in our library of standard functions. This library is sourced by every invocatable. When the given error condition is detected, the user is notified of the problem via an X11 popup. The invocatable can optionally be configured to send and email notification to the application owner as well.

The second classification of errors available for trapping and reporting are run-time errors, those generated by an application itself. Most well behaved applications report these types of errors via the Unix `stderr` descriptor. Invocatables desiring this level of trapping launch their applications via our in-house X11 program called XLaunch. The trapping is accomplished by passing the run string to XLaunch, which then spawns the application process and monitors `stderr`. Errors received at `stderr` are reported to the user via an X11 popup window and are optionally sent to the application owner as a "heads up" to the fact that a user is experiencing problems. Occasionally, we find applications that report meaningless information via `stderr`. To cover this case, XLaunch has several options that allow the invocatable author to provide various *weedout* strings. These strings are regular expressions filter that which constitutes a real error (i.e., one worthy of reporting).

The XLaunch functionality is particularly useful for applications that are configured to launch via CDE actions. Without XLaunch, errors end up in `/dev/null` or off in an obscure error log file which most users don't know to check. The use of XLaunch and invocatables has eliminated those frustrating experiences where a CDE action is invoked but nothing seems to happen, probably because an error occurred without visibility.

Logging and Usage Metrics

As stated in the fifth Objective, Centralized Management, we have added a simple logging function to all of our invocatables. Every time an invocatable is launched, the following information is captured:

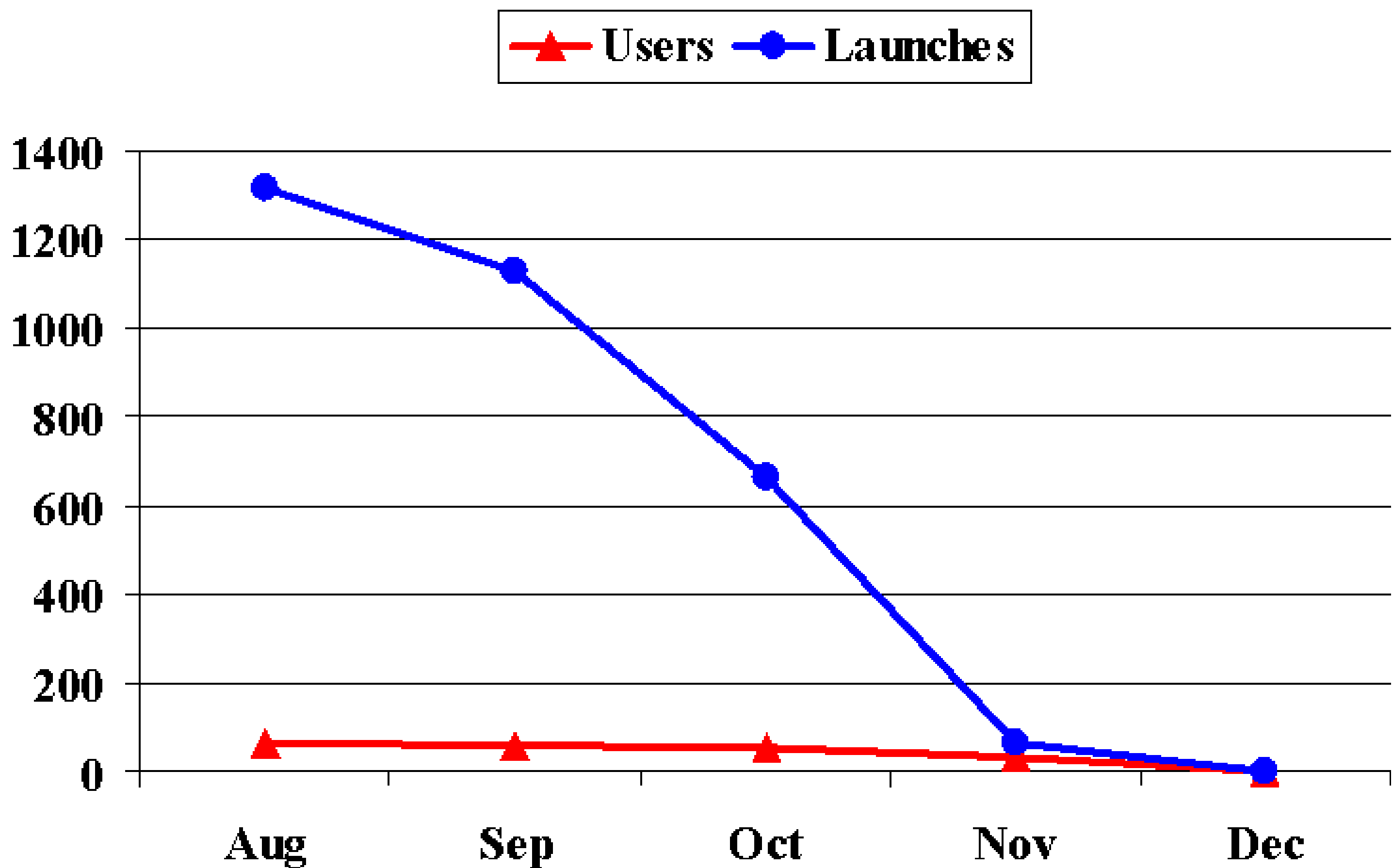
- Host, the system name on which the application was launched.
- Date, current date in a machine-readable format.
- Time, the current time in a machine-readable format.
- Application Suite, the name of the tool-set. Not the actual executable run.
- Executable command, the actual command the user ran.
- User, the user's login account name.
- Optional, this field is available for application owners to log information they deem appropriate such as version info.

A sample snippet from our log looks like this:

```
quad.spk.hp.com;20000111 17:48;SYNOPSYS;dc_shell;joe;version=1999.10
cube.spk.hp.com;20000111 14:44;CDE;login;fred;
spike.spk.hp.com;20000111 19:07;NETSCAPE;netscape;jane;Version: Com 4.7
```

To be honest, we originally added the logging capability because it was fairly easy to implement and we thought it might aide in troubleshooting. We now view logging as a critical piece of our invocatable strategy. Over time, the data collected by this mechanism has proved to be invaluable. As you can see in the sample above, we have even added a call to the logging function as part of the CDE startup even though this does not have anything to do with invocatables. Once we had the logging mechanism in place, we found that there was no reason to limit its use to invocatables. By adding logging of CDE session startups we are able to better track our workstations and users. This is just one example of how the benefits of using invocatables have spilled over into other areas of system administration at our site.

Here is a real live example of how we used the logging data gathered by one of our invocatables in order to track the obsolescence of a non-Y2K compliant application:



Managing Application-Specific System Boot Scripts

If you are going to consolidate an application's configuration requirements into an invocatable, you don't want to stop short of including logic to handle system boot scripts. While most applications do not require any special daemons to be started at boot time, there are those that do, such as client/server database applications. These applications require the database server portion of their code to be properly started and shut down as part of the server's startup/shutdown process. The application's invocatable is the perfect location to place the logic necessary to perform these tasks. The system boot scripts then need only call the invocatable to bring the daemons up or down. This is in keeping with our strategy of localizing all application configuration tasks in the invocatable.

In order to implement this type of functionality one must first determine how the invocatable will be able to detect that it is being called for a system boot script. To date, we have found two different methods that work well depending upon the specific case. The first method is to create a *synthetic* command for the task as already outlined in the section "The PATH to Invocatables". In fact, that is exactly what was done for one of the examples cited in that section, startWindRiverLicenses.

An alternate method may be used in cases where the daemon that needs to be launched is one of the commands in the application's bin directory. In this case, you will have already built a link for this command. All that is needed is a section of custom logic in the invocatable which will detect this command name (via the $0 parameter). In this case rather than simply setting up the environment and running the command, the invocatable's custom logic would call the daemon with whatever options are appropriate for launching the daemon. For example:

```
case ${progname} in
                # Are we starting the doors database daemon?
      doorsd)   eval nohup $EXECUTABLE -serverdata $SERVERDATA -port \
                     $DOORSPORT -logfile $DOORSLOG $parameters &
                ;;
                # Anything else is a doors command-line program...
           *)   eval $EXECUTABLE -data $DOORSDATA $parameters
```

In this example, all of the environment variables shown have been set by the invocatable prior to running this case statement. The $progname variable was set using "progname=${0##*/}" so that it contains the name of the command being executed.

Conclusion

There?s some selling to do anytime one proposes a new way of doing things. Our site certainly was no exception to this rule. At first glance invocatables seemed overwhelming to some administrators/application owners. We needed every application owner on board with this approach if we were going to be successful. Here are the key steps we took to insure a successful invocatable implementation:

- Cast the vision. Make sure everyone understands what the benefits are.
- Develop a library of utility functions to be shared by all invocatables. If a function is used by more than one invocatable, it should exist in the library.

- Create an example template invocatable. Nothing communicates better than a real life example.
- Provide documentation and training resources.
- Offer consulting resources to new authors. If they know there is someone willing to help them through the creation of their first invocatable, the task becomes less ominous.

We have been using invocatables at our site for a couple of years now. With just a few tweaks to our original utility library functions what began as an HPUX implementation now spans across the Linux and Sun platforms as well. We have met our original objectives as well as a few new ones along the way. To date, we have written and deployed about 100 invocatables. Each one has unique issues that it addresses along with the consistent core of functionality it shares with all the other invocatables.

Rather than trying to describe every detail of our implementation, we have presented the "essence" of what our invocatable environment looks like. It is our experience that the greatest opportunity for leveraging across a diverse community tends to occur at the conceptual level. With that in mind, we have attempted to cast a vision of the possibilities and benefits of using invocatables. We hope that you will catch this vision and implement it with your own creative mechanisms rather than simply adopting the ones we prescribe. In this way you can deploy your invocatable strategy in a manner consistent with your environment's unique needs and requirements.

Example Invocatable

Finally, here is an example of one of our invocatables. For the sake of brevity, we have not included the various library functions. Most of the functions are fairly self-explanatory based on their names. This should give you a taste for what an invocatable might look like.

```
#!/usr/bin/sh

############################################################################
#  ABOUT THIS FILE:
#
#       $Source: /opt/spk-src/inv_src/maker.inv $
#       $Revision: 1.33 $ $Date: 1999/07/02 17:55:54 $
#
#       Framemaker invocatable

############################################################################
#  LIBRARY FUNCTIONS AND ALIASES
#
#       This section sets the shell environment appropriately, sources in
#  aliases for all of the commonly used HP-UX utilities, and sources
#  in a set of utility functions developed for use by invocatables.

#  Don't treat unset parameters as an error when substituting.
set +u
. /opt/spk/lib/aliases.lib
. /opt/spk/lib/util_functions.lib
. /opt/spk/lib/msg_catalog.lib

############################################################################
#  ENVIRONMENT VARIABLES (GENERIC)

# The name of the application suite, the owner and email command string will
# be used by various functions and popup messages.
export APP_HANDLE=FRAMEMAKER
APP_OWNER="John Smith, x3456"
APP_MAILCMD="mailx -s '10X Maker err' jsmith"

############################################################################
#  Parse the "+" options and set the variables $progdir, $progname,
#  $parameters, $run_mode, $version.  The user's version preference will
#  also be retrieved from the ~/.dt/prefereces file by making the call:
#       version=$(get_preference $APP_HANDLE "Version")

. /opt/spk/lib/parse_parameters.lib

############################################################################
#  XLaunch weedout settings

# Instruct XLaunch to discard errors until it finds one that matches this
# string.  From then on, display any errors received.
UNTIL_REGEX="Finished Loading"
# Instruct XLaunch to discard any messages that match this regular expression
WEEDOUT_REGEX="LiveLinks"

############################################################################
#  PREFERENCES SETUP (calls GUI if user involked with +prefs option)
# The app prefs GUI will stores all user preferences in ~/.dt/preferences
# for later retrieval via the get_preference function.
if ! run_mode=$(set_app_preferences $run_mode $APP_HANDLE "$APP_OWNER")
  then exit 0
fi
```

```
#############################################################################
#  APPLICATION-SPECIFIC ENVIRONMENT VARIABLES

# Set the paths to all versions currently available on the system
FRAMEDIR_55=/opt/adobe/maker/5.5.6
FRAMEDIR_50=/opt/adobe/maker/5.1.1

case $version in
       5.0)     FMHOME=$FRAMEDIR_50
                ;;
  default*|\
       5.5)     FMHOME=$FRAMEDIR_55
                ;;
         *)     popup_message "Invalid $progname version $version
specified!  Press Close to continue."
                exit 1
                ;;
esac

export FMHOME
export PATH=$FMHOME/bin:$PATH

EXECUTABLE=$FMHOME/bin/$progname

# "spkreader" is a link for a synthetic command set up for use by Mentor.
# When maker.inv is invoked by the name spkreader, it adds some necessary
# command line arguments.
if [[ $progname = spkreader ]]; then
  # This is invoked:   spkreader -f <filename> [<link>]
  if [ "${parameters#* * }" != "${parameters}" ]
    then
      # There are three (or more) parameters, only use the first two
      # within the -fpname command string (reader startup)
      nonlinkparam="${parameters% *}"
    else
      nonlinkparam="${parameters}"
  fi
  # Connect to an existing framemaker session rather than starting a new one.
  parameters="-rpcProp _Frame_Reader_RPC -fpname \\\"reader $nonlinkparam\\\" $parameters"

  # Now that the parameters are all set, change the progname/EXECUTABLE to
  # reflect the program we really want to run in this case.
  progname=fmclient
  EXECUTABLE=$FMHOME/bin/$progname
fi

#############################################################################
#  LIST VERSION BIN DIRECTORIES IF INVOKED BY SOFTWARE DISTRIBUTOR

if [[ $run_mode = listvers ]]; then
  # Echo the bin directory of each version of framemaker available on the
  # system.  Each directory will be searched for executables which need to
  # have corresponding links at /opt/inv.  Each version must be searched
  # so that the superset of all commands available across all versions is
  # generated.

  echo "$FRAMEDIR_55/bin: \-> .wrapper:"
  echo "$FRAMEDIR_50/bin: \-> .wrapper:"

  # Tell configure script to create a link for synthetic command "spkreader"
  echo "LINK:spkreader:"

  # Exit now because we don't want to run anything when called by SD.
  exit 0
fi

#############################################################################
#  USER OVERRRIDE-ABLE VARIABLES

# Using this syntax, the variable only gets modified if it was not already
# set.  This allows users to override its value (by setting and exporting
# FM_FLS_HOST before launching maker)
export FM_FLS_HOST=${FM_FLS_HOST:-app3}

#############################################################################
#  COMMON LOGIC
```

```
# Skip these tests when the run mode is "env" (ie. if the user called us
# with the "+env" option.
if [[ ! $run_mode = env ]]; then
  set_mailit_alias "$APP_MAILCMD"

  #  Check for another instance of this process running.  Framemaker can
  #  open multiple documents and therefore should not be started more than
  #  once (to avoid consuming multiple licenses).
  if ! no_other_process WARN_ONLY "$progname" "$APP_MAILCMD"
    then exit 1
  fi

  # Check for sufficient free colorcells, if required.  The global resources
  # file for Maker has forcePrivateMap set to true.  We only need to check
  # for sufficient colors if the user has set the resource to False in their
  # local .Xdefaults file.
  privmap=$(xrdb -query | grep 'Maker.forcePrivateMap:')
  if [[ $progname = maker || $progname = reader ]]; then
    if [[ -n $privmap ]]; then
      if [[ $privmap = *[Ff][Aa][Ll][Ss][Ee] ]]; then
        if ! enough_colors 18
          then
            spkapplog F "Not enough colors available."
            exit 1
        fi
      fi
    fi
  fi

  #  See if the user can see and execute the EXECUTABLE (ie. it exists and
  #  has executable permissions for the user.
  if ! app_can_execute $EXECUTABLE "$APP_OWNER"
    then exit 1
  fi
fi

############################################################################
#  INVOKE THE APPLICATION

# Report successfull completion of environment setup for specified version.
echo "$APP_HANDLE environment setup completed. (version: $version)"

case $run_mode in
  run)  case $progname in
               maker|\
          fmclient|\
              reader) # These are the X clients, so we need to run these via
                    # XLaunch for the sake of trapping and reporting errors.
                        eval XLaunch \
                       -m \"$APP_MAILCMD\" \
                       -u \"$UNTIL_REGEX\" \
                       -w \"$WEEDOUT_REGEX\" \
                           -e \"$EXECUTABLE $parameters\" &
                        ;;
              fm_fls) # Start the license daemon.  This relieves the init.d
                    # script from needing to specify all the options (ie. it
                    # simply calls the command "fm_fls")
                    $FMHOME/bin/fm_fls \
                       -log /opt/adobe/frame_common/log/fm_fls.log \
                       /opt/adobe/frame_common/license/licenses \
                       >/dev/null 2>&1
                    ;;
                   *) # Any other command is one of the utilities provided by
                    # framemaker.  These are not X clients, so we don't need
                    # to monitor them as closely.
                    eval $EXECUTABLE $parameters
                        ;;
          esac
        # Call the spkapplog logging utility so that we capture useful run
        # information for use in generating statistics and reports.
          spkapplog S
          ;;
  env)  # Don't do anything if we were called with "+env".  At this point
        # all required variables and paths have been set.  All that is needed
        # now is to simply exit.  Users can use this form:
        #     . maker +env
```

```
        # in order to source the framemaker environment into their current
        # shell (usually for troubleshooting purposes).
        :
          ;;
 vars)  # Show each variable we are setting (in an evaluatable format)
        # Each statement MUST end with a semicolon for proper interpretation
        echo "export rem=\"######### ${APP_HANDLE} ENVIRONMENT #########\";"
        echo "export FMHOME=$FMHOME"
        echo "export FM_FLS_HOST=$FM_FLS_HOST"
        echo "export PATH=$PATH"
        ;;
esac
```