PIKT: Problem Informant/Killer Tool

Robert Osterlund – University of Chicago

ABSTRACT

When faced with the many problems that arise in a complex of heterogeneous networked workstations, systems administrators often resort to coding scripts to monitor and problem-solve, scripts that they then schedule via cron. PIKT is a new and innovative approach to monitor scripting and managing system configurations. PIKT consists of an embedded scripting language with unique labor-saving features, a sophisticated script and system configuration file preprocessor, a scheduler, an installer, and other useful tools. More than just a systems monitor, PIKT is also a cross-categorical toolkit for configuring systems, organizing system security, formatting documents, assisting command-line work, and performing other common systems administration tasks.

The Problem, A Solution

Sysadmins have long wrestled with the task of writing generalized scripts to monitor systems and deal with recurring problem situations. As conventionally practiced, this approach has numerous disadvantages: it is hard to account for diversity across machines and operating systems; operations are fragile and error-prone; scripts for handling simple tasks are difficult to code, or are hardly worth the effort to maintain; scheduling and managing scripts are time-consuming and repetitive; setup is inflexible; activity and error logging is rudimentary or nonexistent; and the whole mass of scripts and configuration files is nearly impossible to keep track of or even comprehend.

PIKT attempts to solve some of the problems observed in more traditional methods of monitor scripting and managing system configurations. PIKT is an embedded scripting language and accompanying script interpreter. PIKT is also a sophisticated script and system configuration file preprocessor for use with the Pikt scripting language or any other scripting language of your choice. Finally, PIKT is a cross-platform, centrally run script scheduler (like cron), customizing installer (like rdist), command shell enhancement, and total script and configuration file management facility. PIKT's primary purpose is to monitor systems, report problems, and fix those problems whenever possible, but its flexibility lends itself to quite a few other uses as well.

Overview

In the usual PIKT configuration, you manage the monitored client ("slave") machines from a central ("master") control machine. On the central control machine, there are eight controlling config files: systems.cfg, defines.cfg, macros.cfg, alerts.cfg, alarms.cfg, objects.cfg, programs.cfg, files.cfg. They define your entire setup. You invoke the overseeing management utility, piktc (for "pikt control") to preprocess those files, to install client target files, and to perform other management functions, such as stopping and restarting daemons.

Two daemons run on each client, piktc_svc and piktd. piktc_svc listens for and responds to piktc requests. piktd launches Pikt scripts at specified intervals.

On all clients, piktd wakes up every minute to check if one or more groups of scripts, also known as "alarms", are due to run that minute. Alarm scripts are grouped together as "alerts". Alerts run at specific intervals, e.g., hourly, once daily, once weekly, etc. At the appropriate time, piktd summons the pikt interpreter to run the Pikt scripts for that alarm group. You can also run Pikt scripts manually at the command line, but usually they are invoked by piktd.

pikt is the Pikt scripting language interpreter. Individual Pikt scripts usually monitor just one aspect of a system. You can monitor a single object, or collections of things listed in the object files, for example: system processes, disks, devices, lists, etc.

Each Pikt alarm script gets its input from processes, files, or log files. For log files, only new log entries since the last alarm run are considered.

A typical alarm consists of a sequence of logical tests. If the current input line satisfies one or more conditions, actions may or may not be triggered. Conditions might also refer to data in the previous input line, to data for this line remembered from the prior alarm run, even to data coming from outside the current alarm and pikt process.

Triggered actions might include generating a line of e-mail. At the end of the current pikt run, queued lines are e-mailed in a single problem report to one or more sysadmins. Queued lines might be printed or logged, whether to syslog, to this alert's log file, or to some other special log file. Or commands might be executed, for example to restart a detected dead system process, to chown a file, or perhaps to page the sysadmins.

For generating alarm script input, for taking action, also for serving as subroutines, you can employ auxiliary programs and scripts written in other, non-Pikt languages.

All external commands are logged for debugging and auditing purposes. If your alarm script makes reference to data from the prior alarm run, current data is stored in history files for looking up next time. And, very importantly, all errors are logged (including errors generated by invoked scripts written in other languages), giving you a complete audit trail when things go wrong.

You may also employ PIKT to manage system configuration files, such as inetd.conf, syslog.conf, sudoers, etc. It becomes much easier, for example, to enforce consistent access rights across your many systems.

The PIKT binaries are written using a combination of C, lex (flex), and yacc (bison). Most of the sample scripts are written in the Pikt script language, although several auxiliary Perl, expect, and shell scripts are also provided.

Configuration

Config Files

Every config file is a sequence of stanzas. A stanza consists of a stanza identifier, in the first column, then the stanza body, either on the same line or in multiple lines following. The stanza body must be indented, using spaces, tabs, or the #indent preprocessor directive.

Almost without exception, and aside from the above simple rules, PIKT is indifferent to script and config file layout. In other words, spacing and line breaks really don't matter, and you may lay out your config files in any style that pleases you.

PIKT comments are like those in C++, that is, // and /* */.

systems.cfg is where you specify host systems, host aliases, and host groups. Here is an example:

```
///////////////////////////////////
//
// PIKT systems configuration file
//
///////////////////////////////////

solaris
    hosts       moscow athens2
                berlin milan london
                paris paris4 paris5

linux
    hosts       murmansk firenze

...

milan
    aliases     bonn rome

...

mailserver
    members     moscow paris

...
```

defines.cfg specifies a set of "defines" – logical switches for including or excluding sections of the config files. Here are some example defines:

```
debug           FALSE

verbose         FALSE

paranoid        TRUE
```

macros.cfg specifies a collection of text substitutions. Macro definitions, but not macro names, may include embedded macros. Some examples:

```
behead(N)       =sed '1,(N)d'

offhours        ( #hour() >= 18 || \
                  #hour() < 6 )

sysadmins       brahms|albeniz|liszt
```

In config files, a macro reference is preceded by an equal sign, for example: =behead(1), =offhours, =sysadmins.

Macros may also include macro arguments. As with simple text-substitution macros, macros-with-arguments may reference other macros-with-arguments, so all manner of macro nesting is allowed.

In alerts.cfg, you schedule alarm scripts. Here is an example alerts.cfg stanza:

```
Urgent
    timing      0-45/15 * * * *
    drift       5
    priority    10
    mailcmd     "=mailx -s 'PIKT \
                Alert on =host: \
                Urgent' =pikturgent"
    lpcmd       "=lp =piktprinter"
    alarms
                SysRebootUrgent
                FsMountsUrgent
                SwapChkUrgent
                ...
```

The timing parameters follow the usual cron conventions and then some. One not so usual timing spec is random timings (for example, 'timing 20% * * * *', which says to run the alert on average every five minutes). The random timing spec is especially useful in security situations, where you want some unpredictability in your monitoring schedules. Still another novel timing spec is "drift" – how many minutes an alert launch may randomly occur before or after a specified time – useful when you don't want alerts to "bunch up."

As alarm scripts are run, their output is queued. At the end of the alert run (an alert is a set of alarms), the queued output may be sent as a single e-mail message to one or more sysadmins, or printed out, using the commands specified.

The alarms.cfg file is a series of Pikt scripts or alarm definitions. For a more detailed discussion of Pikt scripts, see the Scripting Language section below.

In objects.cfg, you specify system objects to be monitored. Object listings can also include data parameters. For example:

```
UserDirs
#if kiev2
    /pub/mus_disk_5
    /pub/mus_disk_6
#elif kiev0
    ...
#endif

...

SysProcs
    ...
    cron : /etc/init.d/cron start
    ...

...
```

The file programs.cfg contains support scripts written in other scripting languages, with each program in its own stanza.

In files.cfg, you can centrally manage system configuration files (such as inetd.conf, motd, and so on), and indeed any text file. files.cfg is much like programs.cfg, except that it can and should contain non-program files and/or programs external to the PIKT setup.

Partial Configurations

In a complete PIKT setup, you have all eight basic configuration files. You might, in addition, have #include file spinoffs from those basic eight.

It is possible to deploy PIKT in a partial configuration, with subsets of the eight basic config file types. systems.cfg is always required, but all the rest are optional.

Here are the most common PIKT setups:

- piktc as rsh/ssh replacement (no macros or defines): systems.cfg only
- piktc as rsh/ssh replacement (with macros and possibly defines): systems.cfg, macros.cfg; and optionally defines.cfg
- piktc as rdist replacement: systems.cfg, files.cfg; and optionally programs.cfg, macros.cfg, defines.cfg
- a centrally managed cron replacement: systems.cfg, alerts.cfg, alarms.cfg; and optionally macros.cfg, defines.cfg
- system/network monitor (but without system files management): all config files except files.cfg
- system/network monitor; rsh/ssh, rdist, cron replacements: all config files

So, you may utilize all that PIKT has to offer, or just pick and choose among its many functionalities.

Preprocessing

piktc & piktc_svc

PIKT is managed through the combined action of the interactive control program, piktc (on the central master machine only), and the piktc_svc service daemon (on all slave machines).

The piktc command options are shown in Appendix 1.

When specifying items, you include items with "+" and exclude with "-". For example, "+A all" includes all alerts. "+A all -A EMERGENCY Info" includes all alerts except EMERGENCY and Info. Another way to achieve the same effect is with just "-A EMERGENCY Info" (leaving out the "+A all", which is implicit).

This sample command checksums (using MD5) all files on all user systems except the Linux machines and any down systems:

```
# piktc -m5v ALL -H nonusersys
                     linux downsys
```

Preprocessing

You use piktc to preprocess source configuration (*.cfg) files on the master machine, and send the post-processing alert (.alt), object (.obj), program, and other files over the network to receiving piktc_svc daemons for installation on the slave systems. Preprocessing entails:

- stripping out meta-comments (comments of the form // or /* */)
- #include'ing auxiliary files (e.g., a list of Unix command macros)
- using #if <os|host|hostgroup> #endif preprocessor directives, filtering through lines pertaining only to the current client (e.g., #if solaris)
- using #ifdef <define> #endifdef preprocessor directives, for including/excluding portions of the text (e.g., #ifdef debug)
- making macro substitutions (e.g., substituting a Unix command path, with command options, appropriate to the current client)
- performing an across-the-board syntax check

Note that, in addition to the Pikt script and config files, it is possible to use meta-comments, #include's, #if's, #ifdef's, and macros in managed system configuration files and scripts written in other languages (e.g., Perl [11], Python [6], AWK [2]). Note, too, that scripts may rewrite config #include files, raising interesting possibilities for maintaining dynamic system configuration files.

Preprocessor Directives

You can customize config files by means of the #if, #elif, #else, and #endif preprocessing directives. The format is

```
#if <machine class>
    <lines>
#elif <machine class>
```

```
     <lines>
#else
     <lines>
#endif
```

where <machine class> can be a series of host names, host aliases, or host groups, separated by the |, &, or ! set operators. | indicates set union, & set conjunction, and ! set negation. You can also use parentheses, (and), in the class specifications.

Akin to #if, a second class of preprocessor directives consists of: #ifdef, #ifndef, #elifdef, #elifndef, #elsedef, #endifdef, #define, and #undefine. The format is

```
#ifdef <define>
     <lines>
#elifdef <define>
     <lines>
#elsedef
     <lines>
#endifdef
```

where <define> is an identifier representing a type of logical switch that is either defined (true) or undefined (false).

Logical defines are set (to TRUE) or unset (to FALSE) in any of three ways: (a) in the file defines.cfg; (b) in any config file, except systems.cfg or defines.cfg, by means of the #define and #undefine directives; or (c) at the command line, by means of either the +D or -D switches.

Observe that you can set and unset defines on a per-machine basis in the defines.cfg file, for example

```
#if dbserver
paranoid     TRUE
#else
paranoid     FALSE
#endif
```

as well as nest #ifdef's within #if's, and vice-versa, throughout the config files.

A config file can incorporate one or more other files by means of the #include directive. Included files may themselves include other files, but only of the same basic configuration type (macro files include macro files, for example). Here is an example #include directive:

```
#include <security_alarms.cfg>
```

Includes are especially useful for compartmentalizing across different systems administrators (where each has his/her own sub-config file), and across functions (e.g., security alarms in one file, network alarms in another), and for including files contributed by outsiders. Includes are also good for quarantining information particular to different operating systems.

There are other preprocessing directives, but the ones described above are the most common.

Scripting Language

Script Outline

The general outline of a Pikt script is:

```
<script name>
    init
        status         active|
                       inactive
        level          emergency|
                       urgent|
                       critical|
                       ...
        task           "<text>"
        input proc|    "<process>"
              file|    "<file>"
              logfile  "<logfile>"
        filter         "<process>"
        seps           "<char(s)>"
        dat            <var> <spec>
                        ...
        keys           <var> [...]
    begin
        <statement>
          ...
    rule
        <statement>
          ...
    ...
    end
        <statement>
          ...
```

Init Section

In the init section, you lay the basis for subsequent script actions.

The alarm is given one of eight severity levels, analogous to syslog's severity levels.

The primary alarm input is the output of a process, the full contents of a text file, or logfile updates (new info since the previous alarm run). If a process, it can be any system process (including multiple processes tied together by pipes) yielding text output. Pikt does not deal with binary input. Input may also be passed through an optional filter.

One or more dat statements map input data to variables. The dat statement takes one of three forms:

```
dat <var> x       [ordinal]
dat <var> x,y     [columnar]
dat "<regexp>"
```

For ordinal input, ‘‘seps’’ specifies a field separator (or separators) other than the default (whitespace).

Concluding the init section, the optional keys line lists variables used as database lookup keys when referring to history values (values stored from previous script runs).

Begin, End, and Rule Sections

Next come action statements, grouped into begin, end, and rule sections.

The heart of a Pikt script is the main processing loop: A line of input (from a proc, file, or logfile) is read in, then acted upon, the next line is read in and acted upon, and so on until the input is exhausted. Before input processing, you might have a begin section, to initialize some variables or take some other preliminary actions. You might also have an end section for input processing followup. (You can achieve additional data processing loops within a Pikt script using a combination of #fopen(), #popen(), #read(), and #fclose(), and/or #pclose().) In other words:

```
    begin               [optional]
        <statement>
        <statement>
         ...
 [while there's input]  [optional]
    rule
        <statement>
        <statement>
         ...
    ...
 [endwhile]
    end                 [optional]
        <statement>
        <statement>
         ...
```

The input processing loop consists of one or more rule sections. A rule section usually groups together program statements pertaining to a single attribute of the current input line. Strictly speaking, there is never a need to break up the set of input processing statements into separate rule sections, but doing so helps clarify program logic.

Data Types

Pikt has three basic data types: strings, numbers, and file handles (or proc handles).

Pikt supports both ‘‘associative’’ (string-indexed) and numeric (numerically-indexed) arrays. Array indices are computable (e.g., a concatenation, or the sum of two functions). Numerical arrays are (for now) limited to at most three dimensions. Note that, unlike with many other languages, Pikt array indices start at 1, not 0. (In Pikt, generally speaking, all indexing, in whatever context, begins with 1.)

Variables come in three different time forms. “$” and “#” as variable prefixes refer to current values (strings and numbers respectively). “@” (e.g., @uid) signifies the value for this variable for the preceding input line. “%” (e.g., %usage) signifies the value for this variable during the previous script run.

So, in Pikt, there is no need to save input data values from one line to the next. Values from the previous input loop are stored automatically for you. The same is true with so-called ‘‘history variables.’’ Pikt stores values in a data file for recall the next time the alarm script runs. If a value is tied to a particular input data variable (specified in a dat statement) and a particular line of input, Pikt does a keyword lookup (specified in a keys statement) to find the appropriate data value.

Other Language Features

In general, every Pikt object serves a semantic purpose. Hence, and for example, parentheses are not required around an if condition or the arguments to a for statement. Nor are semicolons or end-of-lines required to signal the end of a program statement.

Pikt provides the usual operators, and a few not so usual. They mostly follow the Perl and AWK models.

Pikt offers a wide variety of built-in functions. An unusual feature of Pikt functions is that they are data-typed: their return value type is signified by either the “$” prefix (for string; e.g., $trim()) or “#” prefix (for number; e.g., #median()). Pikt does not currently support user-definable functions, although you can write pseudo functions using macros-with-arguments to achieve much the same effect.

Pikt comes with a panoply of flow control structures, most usual, and a few not so usual (e.g., ‘again’, for repeating the current rule; ’leave’, for leaving the current rule). Every Pikt statement begins with a keyword (e.g., ‘set’, ‘if’, etc.). Statement blocks are indicated by a keyword-keyword combination, for example, if-endif, for-endfor.

Pikt uses AWK and GNU RX-style regular expressions.

Several Monitoring Examples

It should be emphasized that the examples following are not an intrinsic part of PIKT. They are solutions that you might implement, not that you are forced to adopt.

Case Study 1: IdleUserSession

IdleUserSession is a short Pikt script to kill abandoned user sessions. Listing 1 is the source version on the master control machine as it would appear in the alarms.cfg file.

We have decided that this needs to be run every other hour or so, so we group it with other ‘‘critical’’ alerts in the alerts.cfg file:

```
Critical

    timing       30 0-22/2 * * *
    drift        5
#if moscow | munich
    priority     10
#else
    priority     0
#endif
    mailcmd      "=mailx -s 'PIKT \
                  Alert on \
                  =pikthostname: \
                  Critical' \
                  =piktcritical"
    alarms
                 ...
                 IdleUserSession
                 ...
```

We would install this alarm, along with the other alarms in the Critical alerts group, with the command

```
# piktc -iv +A Critical -H downsys

processing madrid2...
installing file(s)...
Critical.alt installed

...
```

We have defined macro command paths in macros.cfg like so:

```
#if solaris
...
kill            /usr/bin/kill
...
nawk            /usr/bin/nawk
...
#endif
```

If the current client were defined as a solaris system in the PIKT systems.cfg file, the piktc preprocessor installs this script on the client (in the Critical.alt file) with the macros resolving to the appropriate solaris command paths, as in Listing 2, for example.

Note how macro substitutions have inserted the appropriate paths for the w, nawk, ps, and kill commands. If this were for one of the other supported operating systems, different paths would be inserted.

You no longer have to concern yourself with specifying the correct path for this or that command in your scripts, either by maintaining separate script versions or by inserting per-OS case statements into your code. Simply define the path once and for all in the macros.cfg file, then use the =nawk macro (for example) ever after in all of your scripts (including scripts written in other languages, such as Perl, AWK, etc.). PIKT will automatically substitute the correct version for you.

Input data results from the command "=w", i.e., "/usr/bin/w". Here is a sample input line:

```
bach pts/4 29Jun98 3days 3:25 2 zsh
```

We pass this input along to nawk with the instructions: match lines showing idle time in days; transform, for example, "pts/4" into "pts\/4"; output just the first and second fields.

Pikt maps the nawk output "bach pts\/4", setting $user to the first field and $tty to the second.

This alarm has but one rule: We exec a kill command to terminate the idle session in question. (The exec is automatically logged for auditing and debugging purposes.)

You could, if you want, add rules to kill root sessions only, or to kill after midnight and on weekends, or if certain other conditions are met. Instead of killing, you could send e-mail alerts to the system administrators, who could then decide if manual session kills are required.

Case Study 2: FileStatChk

One thing you would certainly want to monitor is the state of essential system files: Have they

```
IdleUserSession
    init
        status active
        level critical
        task "Terminate idle user sessions."
        input proc "=w | =nawk '/[1-9]day/ {gsub("\\/","\\\\/"); \
                    print $1 " " $2}'"
        dat $user 1
        dat $tty  2
    rule
        =execwait "=kill `=ps -ef | =nawk '/$user.+$tty/ {print \$2}'`"
```

Listing 1: IdleUserSession (source version).

```
IdleUserSession
    init
        status active
        level critical
        task "Terminate idle user sessions."
        input proc "/usr/bin/w | /usr/bin/nawk '/[1-9]day/ \
                    {gsub("\\/","\\\\/"); print $1 " " $2}'"
        dat $user 1
        dat $tty 2
    rule
        exec wait "/usr/bin/kill `/usr/bin/ps -ef | \
                  /usr/bin/nawk '/$user.+$tty/ {print \$2}'`"
```

Listing 2: IdleUserSession (target version).

disappeared? Do they have the right ownerships and permissions?

We start by listing those files, together with their desired attributes, in objects.cfg (see Listing 3).

If we had adjusted the files list for the moscow system only, we would refresh the SysFiles objects set on that system with the command:

```
# piktc -iv +O SysFiles +H moscow
processing moscow...
installing file(s)...
SysFiles.obj installed
```

We could refresh all objects files on all active systems with the command

```
# piktc -iv +O all -H downsys
```

```
SysFiles

#if linux

    /etc/group              -rw-r--r--      644     root    root
    /etc/passwd             -rw-r--r--      644     root    root
    ...

#endif  // linux

...

// local stuff

#if moscow
    /etc/mail/classalias    -rw-r--r--      644     root    other
    ...
#endif

...
```

Listing 3: SysFiles.

```
FileStatChk

    init
        status active
        level critical
        task "Detect critical file access deviations on system files."
        input file "=sysfiles_obj"
        dat $fil 1
        dat $prm 2
        dat $mod 3
        dat $own 4
        dat $grp 5
        keys $fil

    rule
        if ! -e $fil
            output mail "$fil not found!"
            next
        endif

    rule
        do #split($list, $command("=lld $fil"), " ")

    rule
        if $list[1] ne $prm
            =execwait "=chmod $mod $fil"
            =outputmail "$fil permissions $list[1] are wrong" . \
                $if(#defined(%list[1])," (were %list[1]),",",") . \
                " changed to $prm"
        endif

    [similar rules follow]
```

Listing 4: FileStatChk

It should be clear by now that the file /etc/mail/classalias would appear in moscow's SysFiles.obj file and in no other system's.

Listing 4 is a script to enforce those file attributes.

For the first input line, "/etc/group" would be assigned to $fil, "-rw-r--r--" to $prm, "644" to $mod, and so on.

In the first rule, if the file fails the existence test, that gets reported, and we move on to the next input line.

In the next rule, we take the output of the 'ls -l' command and #split() and assign the component parts to the $list[] array.

In the third rule, if the actual file permissions, $list[1], do not equal the desired permissions, $prm, we fix and possibly report this.

The doexec define lets us control whether actions are exec'ed else a report of intent is e-mailed only. If this is a new PIKT installation, we might want to see what PIKT would do before committing PIKT to actually doing it. We could handle the conditionality this way:

```
#ifdef doexec
    exec wait "=chmod $mod $fil"
#elsedef
    output mail "=chmod $mod $fil"
#endifdef
```

But defining the following macro

```
execwait
#ifdef doexec
    exec wait
#elsedef
    output mail
#endifdef
```

in macros.cfg is more elegant, because now we can more succinctly write

```
    =execwait "=chmod $mod $fil"
```

and either "exec wait" or "output mail" will be preprocessed in depending on how we defined doexec earlier.

In most circumstances, we simply want the file permissions fixed and don't need to be told about it. Sometimes, however, we want a full report of all that PIKT is doing. We control this by setting, in defines.cfg, the define verbose to be TRUE or FALSE. By defining the outputmail macro in macros.cfg as

```
outputmail
#ifdef verbose
    output mail
#elsedef
    output log "/dev/null"
#endifdef
```

```
rule
    output log "=swapchk_log" $inline()

end // only report if use is very high and increased by at
    // least 5% since last time (hence don't report when
    // swap use is high but declining)
    set #use = (#blksum-#fresum)/#blksum
    if    ( #use >= 80% )
       && (    ( ! #defined(%use) )
            || ( %use < 80% )
            || ( #use - %use >= 5% )
          )
        output mail "swap utilization is $text(100*#use,0)%:=newline"
        output mail "swapfile             dev  swaplo blocks   free"
        for #i=1 #i<=#innum() #i+=1
            output mail $line[#i]
        endfor
        output mail =newline
        output mail $command("=dfk /tmp | =behead(1)")
        =dutop(10, /tmp)
        output mail "contents of /tmp:=newline"
        do #popen(LL, "=ll /tmp", "r")
        while #read(LL) > 0
            output mail $rdlin
        endwhile
        do #pclose(LL)
        output mail =newline
        =toptop(20)
    endif
```

Listing 5: SwapChk (fragment)

we can concisely write

```
=outputmail "$fil permissions
            [...]"
```

If verbose is set to FALSE, the message is logged to /dev/null, that is, just thrown away.

Note the $if(#defined(%list[1])," (were %list[1]),",","). If we have run this script before, we have a record of the actual file permissions the last go-around in %list[1]. PIKT remembers this for us automatically. So if #defined(%list[1]) is true, we report what they were, and in any case report what they have been changed to – but only if we have set verbose to TRUE.

Case Study 3: SwapChk

Another thing we monitor is if systems run out of swap space. For that purpose, we use the SwapChk script, a portion of which is shown in Listing 5.

The input for this script comes from input proc "=swap -l | =behead(1)". The last rule above logs all input. This might come in handy some day if we need data to justify purchase of additional RAM.

At the end of all input, we compute #use as a percentage. If #use is equal or greater than 80%, or if %use is not defined (because this is the first alarm run, say), or if %use was less than 80% previously, or #use has gone up by at least 5% over the previous %use, we format a report and send it off as alert mail. Listing 6 is a sample report.

PIKT has assembled for us automatically all the diagnostic information we need to assess the situation. Moreover, after we have identified user freil as the memory hog, we can simply add some extra comments to the top of this alert e-mail and forward it along to freil – demonstrating one advantage of using e-mail as PIKT's primary notification mechanism.

We could also, at least under certain circumstances or on certain systems, augment swap space on the fly by adding the appropriate Pikt exec statements.

Case Study 4: ProcCountsChk

Recently, we have faced a crisis where a bug in the current version of our Web-based e-mail client has

```
                         PIKT ALERT
                  Thu Aug 17 21:20:14 2000
                          paris6

URGENT:
    SwapChk
        Report when swap use is high

        swap utilization is 98%:

        swapfile             dev  swaplo blocks   free
        /dev/dsk/c0t0d0s1   32,1      16 1003184  24384
        /pub/perf_disk_20/swap  -       16 524272      0

        swap                  803568  757800   45768    95%    /tmp

        758376  /tmp/SAS_worka0000420D
        8       /tmp/screens
        240     /tmp/ups_data
        ...

        contents of /tmp:

        total 544
        drwx------   2 freil  perf       629 Aug 17 21:18 SAS_work
        drwxr-xr-x   2 root   other       69 Aug 16 06:15 screens
        -rw-rw-r--   1 root   sys     239160 Aug 16 11:12 ups_data
        ...

        last pid: 17014;  load averages:  0.20,  0.23,  0.23  21:20:21
        54 processes:  46 sleeping, 3 zombie, 4 stopped, 1 on cpu

        Memory: 128M real, 1576K free, 738M swap in use, 7984K swap free


          PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU CMD
        16845 freil     1  35     0   12M 3336K sleep   4:27  9.28% r3
        16909 freil     3  35     0 6432K 1464K sleep   1:37  5.21% sas
        16969 root      1  33     0 4872K 2792K sleep   0:00  2.80% pikt
        ...
```

Listing 6: SwapChk (sample report).

```
ProcCountsChk
    init
        status active
        level emergency
        task "Report unusually high counts of per-user procs."
        // note: a defunct process might show an empty comm field
        // below, so we pipe the ps output through the awk filter
        input proc "=ps -eo user,comm | =behead(1) | =awk 'NF==2' | \
                    =sort | =uniq -c"
        dat #count 1
        dat $user  2
        dat $proc  3
    begin   // read in process and threshold data from objects file
        if #fopen(PROCCOUNTS, "=proccounts_obj", "r") != #err()
            while #read(PROCCOUNTS) > 0
                if #split($rdlin) == 5
                   set #lgcnt[$1] = #val($2)    // log   thresholds
                   set #alcnt[$1] = #val($3)    // alert thresholds
                   set #pgcnt[$1] = #val($4)    // page  thresholds
                   set #klcnt[$1] = #val($5)    // kill  thresholds
             // else send an error message?
                fi
            endwhile
            do #fclose(PROCCOUNTS)
        else
            output mail "Can't open =proccounts_obj for reading!"
            quit
        fi
    rule
        foreach #keys($pr, #lgcnt)
            if $proc =~~ "$pr$"   // '=~~', not 'eq', so that '\\*'
                                  // works as a default
                if #lgcnt[$pr] && #count >= #lgcnt[$pr]
                    // for gathering diagnostic stats
                    output log "=proccounts_log" $inline
                fi
                if #alcnt[$pr] && #count >= #alcnt[$pr]
                    output mail $inline
                    if $proc eq "imapd"  // special case
                        =archive_mail_file($user, #true())
                    fi
                fi
                if #pgcnt[$pr] && #count >= #pgcnt[$pr]
                    exec wait "echo '=pikthostname: $inlin' | \
                               =mailx -s '=pikthostname: $inlin' \
                               =pagesysadmins"
                    pause 5
                fi
                if #klcnt[$pr] && #count >= #klcnt[$pr]
                    =kill_user_proc($proc, $user, #true())
                fi
                next    // next input line
            fi
        endforeach
```

Listing 7: ProcCountsChk

the imapd, under occasional and mysterious circumstances, spawning instances of itself every second or so. For a handful of users, we are seeing occasional "imapd storms" with per-user imapd counts reaching into the dozens, hundreds, and sometimes even thousands! At about the same time, but for different reasons, we began seeing "ypserv storms". Not only do these storms risk losing user mail files, they also imperil the entire system. Listing 7 is a Pikt script we have put into operation to deal with these sorts of problems.

The input proc statement yields input like

```
 34  root /usr/lib/sendmail
404  chico imapd
  1  zeppo imapd
```

In the begin section, we read data in from the ProcCounts.obj file (see Listing 8).

In the script's only rule, we check to see if the actual per-user process count exceeds the thresholds we set in the begin section, also if the threshold is non-zero.

Instead of foreach #keys($pr, #lgcnt), we could have used for $pr in #keys(#lgcnt). These accomplish the same purpose but with somewhat different syntax. Variety of expression and keyword synonyms are typical of Pikt. Did you notice the use of if ... endif in Case Studies 2 and 3 as opposed to if ... fi in the current case

```
ProcCounts

// 0 signifies take no action; 1 signifies always take action

//      proc            log         alert       page        kill

#    if moscow
        imapd           10          100         1000        100
#    endif
#    if mailserver
        sendmail        50          100         200         200
#    else
        sendmail        5           10          20          40
#    endif
#    if nisserver
        ypserv          2           3           3           0
#    endif
        crack           1           1           1           1
        sniffit         1           1           1           1
//      ...
// wild card should be last in ProcCounts list
        \\*             10          20          40          0
```

Listing 8: ProcCounts (objects.cfg fragment).

```
kill_user_proc(P, U, M)
    // kill off all instances of a given process for a given user
    // (P) is the process name (e.g., $proc, or "imapd")
    // (U) is the user (e.g., $user, or "root")
    // (M) is whether or not to output mail (e.g., #true())
    set #killcount = 1    // initialize
    while #killcount > 0
        set #killcount = 0
        do #popen(KILL, "=ps -eo pid,user,comm", "r")
        while #read(KILL) > 0
            if #split($readline) != 3
                cont
            fi
            if    $2 eq (U)
               && $3 eq (P)
#ifdef debug
                output log "=proccounts_log" "$1, $2, $3"
                output log "=proccounts_log" "(P), (U), $text((M))"
#endifdef
                exec wait "=kill -9 $1"
                set #killcount += 1
            fi
        endwhile
        do #pclose(KILL)
    endwhile
    if (M)
        output mail "killed all (U) (P) processes"
    fi
```

Listing 9: kill_user_proc()

study? Another example: elif, elsif, elseif are synonymous, and all achieve identical effect.

If the #lgcnt[] threshold is non-zero and if the process count exceeds the #lgcnt[] threshold, we log some diagnostic statistics for post-mortem analysis. If the process count exceeds #alcnt[], we send alert mail reporting that fact. In the case of imapd only, we also backup the user's mail file by means of the =archive_mail_file() macro (not shown).

If #count exceeds #pgcnt[], we send a short alert message to =pagesysadmins, a macro that resolves to the sysadmins' pager numbers.

Finally, if #count exceeds #klcnt[], we kill off the user processes by means of the =kill_user_proc() macro (see Listing 9).

```
UserActivity

    init
        status active
        level critical
        task "Report and/or log suspicious after-hours activity."
        input proc "=w -hs"
        =wdata

    begin
        exec wait "=touch =useractivity_log"  // forced update
#ifdef worried // or paranoid
        if #false()             // never quit this alarm if we're
                                // worried (or paranoid); monitor
                                // activity at all hours
#elsedef
        if #hour() >= 8         // 8 AM to midnight only
#endifdef
            quit                // don't monitor, move on to next alarm
        endif

    rule
#ifdef worried // or paranoid
        if    #true()          // all users
#elsedef
#  if nonusersys
        if    #true()          // all users, on admin systems
#  else
        if    $user eq "root"  // root only, on user systems
#  endif
#endifdef
           && (    #length($idle) == 0
                || $idle =~ "^[0-9]+$"  // idle time in minutes,
                                        // not hours or days
              )
            // escalate notification at higher levels of security
            output log "=useractivity_log" $inline
#ifdef cautious // or worried or paranoid
            output syslog $inline
            output mail $inline
#endifdef
#ifdef worried // or paranoid
            output print $inline
#endifdef
#ifdef paranoid
            exec wait "echo '=pikthostname: $inlin' | =mailx -s \
                       '=pikthostname: $inlin' =pagesysadmins"
#endifdef
        endif
```

Listing 10: UserActivity.

Here is a sample alert message:

```
             PIKT ALERT
       Tue Apr 25 00:17:02 2000
               moscow
URGENT:
  ProcCountsChk
    Report unusually high counts \
      of per-user procs.
    404  chico imapd
    saved user mail file as
      /var/mail/arc/chico.956639822
    killed all chico imapd \
      processes
```

(We still don't have an understanding of these problems, much less fixes, but at least we are not losing any more user e-mail, and our mail server is coping.)

Before leaving ProcCountsChk, note that by defining all count thresholds to 1 across the board, we can guard against users running "dangerous" or "forbidden" programs such as Crack or Sniffit.

Case Study 5: UserActivity

You can use PIKT's define feature to achieve precision control over your security setup. Consider these security settings in defines.cfg:

```
attentive   TRUE    // lowest level
                    // of security
cautious
#if misscritsys | cssys
            TRUE
#else
            FALSE
#endif
worried
#if misscritsys
            TRUE
#else
            FALSE
#endif
paranoid    FALSE   // highest level
                    // of security
```

Listing 10 shows how you might use them in an alarm to monitor suspicious, after-hours user activity (some per-OS customizations were omitted for brevity).

We can also apply these defines to the UserActivity.log file produced by the UserActivity alarm. Here is a sample log entry:

```
Aug 3 01:36:01 CRIT: root p0 1 -csh
```

Listing 11 shows the log monitoring script.

As security conditions change, we can generate more or fewer log entries by changing our security

```
UserActivityLogChk
    init
        status active
        level critical
        task "Report all new security incidents in UserActivity log."
        input logfile "=useractivity_log"
#ifndef cautious // or worried or paranoid
    begin
        quit
#endifdef
#ifdef cautious // or worried or paranoid
    rule
        output syslog $inline
#endifdef
#ifdef worried // or paranoid
    rule
        output mail $inline
#elsedef // only cautious
    rule
        if $inline =~ "root"
            output mail $inline
        fi
#endifdef
#ifdef paranoid
    rule
        output print $inline
        // page also?
#endifdef
```

Listing 11: UserActivityLogChk.

defines (from TRUE to FALSE, or vice-versa) for different systems, then using piktc to reinstall the modified scripts on those systems. This gives us pinpoint control over our security setup.

Other Uses

Those are just a very few of the things you can use PIKT to monitor. We use it for all kinds of systems administration tasks, including: clearing out /tmp files; reporting system crashes; monitoring changes in critical system files, directories, and devices; detecting passwd and shadow file anomalies; running a mail quota system; reporting "orphaned" accounts and home directories; detecting bad e-mail list addresses; clearing out user Web browser caches; removing core files; rotating and retiring system log files; reporting full file systems; reporting runaway processes; reviving vital system processes; reviewing security log files – the list goes on and on.

Working with Other Scripting Languages

If you prefer to use a different scripting language, that is no problem. Here is a short Pikt wrapper script around a much longer, and very complicated Perl script, =mailchk (/pikt/lib/programs/mailchk.pl):

```
MailChk

    init
        status active
        level warning
        task "Check for mail \
              errors, such as \
              forwarding loops."
        input proc "=mailchk 2>&1"

    rule
        output mail $inline
```

mailchk.pl yields output which the Pikt MailChk script captures in a PIKT e-mail alert. If you wish, you could let your Perl script handle all the reporting, but still have PIKT deal with scheduling and logging, using the minimalist Pikt script:

```
MailChk

    begin
        exec "=mailchk 2>&1"
```

We have a suite of over two dozen account management programs, almost all of them written in Perl, that we maintain within our programs.cfg file. We don't use Pikt or piktd at all to run these. Rather, we use PIKT to manage the per-OS and per-machine differences, to install, and to monitor script integrity.

Config Files Installation and Management

Listing 12 is a portion of our files.cfg, the section configuring inetd.conf.

Turning services on and off is as easy as editing the central files.cfg, then reinstalling inetd.conf with the appropriate piktc command.

Recently, a CERT advisory was broadcast advising against running the rpc.ttdbserverd service with root privileges. For the rpc.ttdbserverd line, we substituted "daemon" for "root" (the line was already commented out anyway), then updated inetd.conf and reconfigured inetd on all Solaris systems with:

```
# piktc -iv +F inetd.conf \
          +H solaris -H downsys

# piktc -xv +S SigHupInetd \
          +H solaris -H downsys
```

where SigHupInetd is a Pikt script written expressly for that purpose.

Another problem we have faced is keeping up-to-date our sudoers file – especially the list of part-time Computer Assistants. We do it in files.cfg by means of an include file:

```
        User_Alias      PARTTIMERS=\
#include <sudo_parttimers_files.cfg>
```

where the sudo_parttimers_files.cfg file might be:

```
        larry,moe,curly,sporty,\
        ginger,baby,posh,scary,\
        john,paul,george,ringo
```

We have a separate script that rewrites the sudo_parttimers_files.cfg file nightly based on an authoritative and up-to-date GNU Mailman list. The result: a dynamic sudoers config file!

Remote Command Execution

You can use piktc for remote program execution as an alternative to rsh or ssh. The command

```
# piktc -Xv +C "<command(s)>" \
          +H <systems>
```

executes the given command(s) on the specified systems.

You can insert PIKT macros within +C command strings. See Listing 13, for example.

```
#if usersys
        ftp     stream  tcp     nowait  root    /usr/tcpd/tcpd  in.ftpd
        telnet  stream  tcp     nowait  root    /usr/tcpd/tcpd  in.telnetd
        ...
#else
        #ftp    stream  tcp     nowait  root    /usr/tcpd/tcpd  in.ftpd
        #telnet stream  tcp     nowait  root    /usr/tcpd/tcpd  in.telnetd
        ...
#endif
```

Listing 12: files.cfg (fragment).

Note that kiev0 is a SunOS system. We want ‘df -k’ to run on the Solaris systems and just plain ‘df’ to run on the SunOS systems. The macro =dfk resolves to the desired path and command option.

System Lists

With perhaps the simplest but still useful PIKT setup imaginable – the piktc binary and a systems.cfg file – you can maintain custom system lists, whether for referencing within other programs, as in this Perl statement

```
@hpsys = ‘piktc -L +H hpux \
                   -H downsys‘;
```

or for command-line work, as in a command loop we use to upgrade our Solaris PIKT binaries (see Listing 14).

The uses of PIKT really are limited only by your imagination!

Security

The current PIKT security model is fairly trusting. There are things you can do now to tighten security, while other things – Kerberos-style client-server authentication and data encryption, for example – are under study and planned for implementation in the near future.

In PIKT.conf, which functions roughly like a .rhosts or .shosts file, you set various access parameters (e.g., uid, gid, master, domain, master_address, etc.) and service rights (e.g., all_services, kill_service, install_service, etc.). By careful and judicious use of these settings, and in combination with other measures (firewalling, running piktc_svc only when necessary, etc.), you can achieve a level of security sufficient in many situations.

When you issue a piktc command, the slave (remote) host and requested service are registered with the master (local) piktc_svc. Upon receiving the service request, the slave piktc_svc checks its local access authorizations in PIKT.conf. If the service request is authorized, the slave piktc_svc then does a ‘‘callback’’ – a second, independent TCP connection – to the master piktc_svc, seeking to verify the request. If it verifies – i.e., both slave host and service request match – only then does the slave piktc_svc perform the requested service. It then returns the outcome of the service request to the requesting piktc. Finally, the slave host and requested service are deregistered on the master piktc_svc. Please see Figure 1.

piktc and piktc_svc do complete service request logging, and piktc_svc marks denied requests as ‘‘ERROR’’. It is not difficult to set up log monitoring scripts (and scripts to monitor the monitoring scripts, etc.) to spot attempted security break-ins. Remember that you can log to syslog and special logs, and dump to a printer, besides sending out e-mail alerts. Scripts can also call pagers in security emergencies. You can even have a monitoring script kill the local piktc_svc under suspicious circumstances.

You can use PIKT for security in at least the following six ways:

1. As just a centrally managed scheduler. Have piktd invoke your preferred security tools according to the schedules in piktd.conf.
2. The above, plus have PIKT manage other security tools’ config files (the inevitable per-machine and per-OS customizations).
3. All of the above, plus use PIKT #ifdef’s to activate and deactivate different security tools as changing conditions warrant.
4. All of the above, plus use PIKT to handle your security log file analysis and incident reporting.
5. All of the above, plus employ PIKT alarms and data objects as supplements to the standard tools. (For example, have PIKT do things that COPS or Tiger don’t do.)
6. Use PIKT exclusively.

```
# piktc -x +C "hostname; =dfk /tmp" +H mus
kiev0
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/sd0e             993006      17  893689     0%    /tmp
kiev
Filesystem            kbytes    used   avail capacity  Mounted on
swap                  769096     296  768800     1%    /tmp
...
```

Listing 13: piktc Command Example

```
# for sys in ‘piktc -L +H solaris -H piktdevsys no_usr_local downsys‘
> do
> echo $sys
> ssh $sys "/pikt/lib/programs/svcstart.pl -k; \
  cp /pikt/bin/pikt* /pikt/bin/bak; \
  cp /usr/local/pikt/bin/solaris/pikt* /pikt/bin; \
  /pikt/lib/programs/svcstart.pl -r"
> done
```

Listing 14: System Lists

PIKT is especially adept at points one, two, and three above.

As for point four, there are good solutions out there already for analyzing your security log files, but PIKT might be superior due to its more powerful and flexible built-in scripting language.

As for point five, extending one tool requires you to learn Perl, another to get intimate with the Bourne shell, while five others require you to learn five different cryptic and proprietary command languages. Learn those languages and modify those tools on their own terms where that makes the most sense, but when sensible resort to PIKT.

As for using PIKT exclusively, in spite of its great virtue of involving just one system and command language to learn and use, I don't propose that PIKT do everything! For many purposes, there are some really excellent alternatives available. I therefore suggest that the optimum solution lies somewhere around points three, four or five.

Security should be systematic, flexible, and easy to manage. These are areas where PIKT excels. At the very least, PIKT is a general framework on which to build your security efforts.

Future Plans

PIKT could use a GUI, not so much to overlay piktc management as to handle incoming alert messages. This could take form as, for example, a Tk/Tcl or Java front-end acting on syslog messages sent to the console machine.

Although there are methods to program against endlessly repeating nuisance messages (so-called "nagmail"), PIKT would benefit from more automated ways to do this. Message routing could also be improved.

PIKT has a steep learning curve, and setup can be daunting. A project is underway to put together a PIKT "standard library" of ready-to-run defines, macros, alerts, scripts, programs, and objects sets. We need to improve the user's "out-of-the-box" experience.

The weakest link in PIKT's chain is the script interpreter, pikt. Although it gets the job done, it is not fast or feature-complete. Pikt is not a stand-alone, all-purpose scripting language, and you cannot effectively call Pikt scripts from programs written in other scripting languages. Pikt was designed to aid in systems administration and to run "fast enough" in a small memory space within the broader PIKT system. Plans are to rewrite Pikt's underlying engine, perhaps using GNU Guile or embedded Perl.

PIKT requires a thorough security audit, and its client-server communications need to be made iron-clad secure. Adding encryption and Kerberos authentication is under consideration. We also plan to implement a comprehensive security package of PIKT defines, macros, scripts, and objects sets to work alone or in concert with other security products.

PIKT has an Introduction and comprehensive Reference Manual but lacks a Getting Started guide, also an Operations Manual.

Other Solutions

PIKT is often compared to Cfengine [3]. In the words of its author, Mark Burgess,

> Cfengine ... is a very high level language for building expert systems which administrate and configure large computer networks. Cfengine uses the idea of classes and a primitive form of intelligence to define and automate the configuration of large systems in the most economical way possible.

In Cfengine, you create a configuration file (or files) describing the ideal setup for all of your hosts. When run, the cfengine program will check the actual machine configurations against the ideal and, if desired, fix any deviations.

Cfengine and PIKT address generally the same problem but in significantly different ways. Cfengine is a high-level, declarative or descriptive language (a single statement might set permissions on hundreds of files, for example), while Pikt is a low-level, procedural language. Cfengine tends to provide specific solutions to specific problems, while PIKT tends to be more general. Cfengine's specificity (it has built-in support for configuring network interfaces, for example) would be out of place in base PIKT. (With PIKT,

```
 MASTER                             SLAVE

piktc<-------------------------------+
  |         (2) send request         |
  |         (7) return result        |
  |                                  |
  |  (1) register slave & request    |
  |  (8) deregister slave & request  |
  V                                  V
piktc_svc<---------------------->piktc_svc
            (4) callback          (3) check authorizations
            (5) reply             (6) perform service
```

Figure 1: callback.

you would write a script to configure the network interface calling the usual UNIX networking commands.)

While Cfengine achieves per-OS and per-machine customization by means of classes, for example,

```
    FTPserver.Sunday.Hr00::
        /local/iu/xferlog rotate=3
```

which means to run xferlog at midnight Sunday if this system is a FTPserver, PIKT would achieve a similar effect as follows (in the alerts.cfg file):

```
Notice
    timing      0 0 * * 0
    ...
        alarms
                ...
#if ftpserver
                LogFileChkNotice
#endif
                ...
```

Cfengine has its own unique keywords, syntax, macro and variable forms, etc. Although Pikt has some unique elements, much of it should be familiar to any Perl or C programmer, especially the idea of preprocessing. If PIKT has a steep learning curve, Cfengine's is equally steep, if not steeper.

Cfengine tries to anticipate many of your needs, but when you veer off the beaten path, Cfengine is not quite so helpful. In many situations, you will still need to write your own scripts. With PIKT, you script everything. This makes PIKT inherently more flexible and applicable to a broader class of applications – not just fixing broken system configurations and executing routine tasks, but also reacting to errant dynamic processes.

Cfengine is quite good at what it is designed to do. It would be especially useful (and superior to PIKT) for configuring a new system or restoring a system after a crash or cracker break-in. One really nice Cfengine feature is that ordinary users can invoke it, attempting to fix a broken configuration if the system administrator is unavailable. (PIKT is typically just for root use.)

In work first presented at the LISA 1999 Conference [4], Alva Couch and Michael Gilfix have

> ... created a system administration library that allows one to perform system administration tasks in Prolog. This is much more powerful and flexible than using other current tools, and has the advantage that the resulting Prolog programs are much closer to describing actual policies than CFEngine configuration files or PIKT scripts.

Perhaps because their comments were based on earlier, less mature versions of PIKT, I feel that they underestimate the power, flexibility, and expressiveness of Pikt scripts, especially the fully documented, macro-enhanced versions found in the central configuration files (as opposed to the preprocessed, uncommented versions installed on the slave systems).

Their Prolog-based approach to systems administration is intriguing and potentially far-reaching, but it suffers from one significant problem: Unless one attains proficiency with Prolog (not a widely used language, to say the least), their system is a ‘‘black box,’’ closed to the do-it-yourselfer who demands complete control over, or at least complete understanding of, the system. In any case, at this time, source code is not yet available for public distribution, so it is hard to evaluate their approach effectively.

There are other systems monitoring packages out there, including: Big Brother [7], and its clone Big Sister [1]; Mon [9]; NetSaint [5]; and still others. These tend to focus more on performance statistics and problem reporting, less on systems configuration and problem solving. To their credit, they rely on standard scripting languages, but they don't deal specifically or as extensively with the problem of per-machine and per-OS customization like PIKT, Cfengine, and the Prolog-based library do.

I have no experience using any of the high-octane, very expensive commercial packages (like Tivoli [8] or CA Unicenter TNG [10]) and can't venture any comparisons or opinions about them.

Parting Thoughts

The heart and soul of PIKT is its preprocessor, piktc, and all the special scripting and file management facilities it provides: per-machine and per-OS #if directives; the #ifdef family of logical switches; #include files; macros; pinpointed file installation; central scheduling; and so on. PIKT moves scripting toward the kind of full-featured development environment that users of ‘‘more serious’’ languages have long enjoyed.

The Pikt scripting language offers some unique features, or features better tailored to the job of day-to-day systems administration: automatic previous-line (@foo) and prior-run (%foo) value references; a clean, uncluttered syntax; free-form, flexible layout; keyword synonyms; automatic logging of everything of consequence; a cautious approach to script execution (no assumed variable defaults; serious errors trigger automatic script shutdown); a built-in input loop (much like AWK's); many standard input and output options.

On the other hand, people have a right to question whether the world needs Yet Another Scripting Language. Also, scripting language preference is often a highly personal, even emotional matter. Pikt, the scripting language, is just the first among equals in PIKT, the sysadmin toolkit. Use of other languages within the PIKT system is encouraged, and embedding other scripting languages within PIKT is actively being considered.

But the point bears repeating: the piktc preprocessor and control program is at the center of PIKT. It is what sets PIKT apart from other scripting languages and other system monitors. PIKT is more than just another systems monitor and Yet Another Scripting Language. When confronted with its multi-functionality, one Web administrator didn't quite know where to list it, saying that he might have to invent a whole new category for PIKT. If ever a tool were more than the sum of its parts, PIKT is that tool. The PIKT combination is a very powerful, wide-ranging, and ambitious toolkit indeed.

Availability

PIKT's homepage is: http://pikt.uchicago.edu/pikt, where you will find not only the distribution package but also complete on-line documentation, sample configuration files, a comprehensive test suite (with over 600 validation tests), and other useful items. PIKT is also available for download at a number of ftp sites. pikt-users and pikt-workers mailing lists have been formed. Operating systems now supported include GNU/Linux, Solaris, SunOS, FreeBSD, OpenBSD, AIX, HP-UX, and IRIX.

Acknowledgments

I need to thank the following persons for their helpful criticisms, suggestions, bug reports, and in some cases code: Bardur Arantsson, Michel Blanc, Jim Botts, Leon Breedt, Chris Halverson, Magdalena Hewryk, Rich Hoffer, Kelsang Wangden, James Low, David Masterson, Miguel Armas del Rio, Roland Roberts, Raul Alexis Betancort Santana, Mike Scheidler, Joe Siegrist, and especially Harlan Stenn, who implemented the PIKT autoconf/automake and who has helped out in other innumerable ways. I owe a huge debt of gratitude to the authors and maintainers of gcc, gdb, and make, as well as GNU flex (lex) and bison (yacc), upon which PIKT relies quite heavily. I am also very grateful to Will Partain and Kelsang Wangden for providing thoughtful and incisive suggestions for improving this paper.

Author Information

In a former life, Robert Osterlund earned a couple of economics degrees from the University of Chicago and worked as an economist and college teacher while serving as a U.S. Peace Corps Volunteer in the Philippines. After making a mid-life career switch to computing, he took computing courses for a while at the University of Illinois at Chicago, then returned to the Philippines to organize and head the computer department at a small college there. He was employed for five years as Senior Programmer Analyst at the University of Chicago's Social Sciences and Public Policy Computing Center, and has worked as Unix Systems Manager at the University's Graduate School of Business since 1995. You can mail him at: Robert Osterlund, Graduate School of Business, University of Chicago, 1101 E. 58th Street, Walker 309, Chicago, Illinois 60637, USA. Or send e-mail to: robert.osterlund@gsb.uchicago.edu.

References

[1] Thomas Aeby, Big Sister, http://bigsister.graeff.com/ .

[2] Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, The AWK Programming Language, Addison-Wesley, 1988.

[3] Mark Burgess, GNU Cfengine, http://www.iu.hioslo.no/cfengine .

[4] Alva Couch and Michael Gilfix, http://www.eecs.tufts.edu/˜couch/prolog/ .

[5] Ethan Galstad, NetSaint, http://www.netsaint.org .

[6] Mark Lutz, Programming Python, O'Reilly & Associates, 1996.

[7] Sean MacGuire, Big Brother, http://bb4.com .

[8] Tivoli, http://www.tivoli.com/ .

[9] Jim Trocki, Mon (Service Monitoring Daemon), http://www.kernel.org/software/mon .

[10] Unicenter TNG, http://www.ca.com/ .

[11] Larry Wall, Tom Christiansen, Randal L. Schwartz, Programming Perl, O'Reilly & Associates, Inc., 1996.

Appendix 1: piktc Command Options

```
Usage:  piktc <-cCdefGhikKlLm#rRstTvxX>
              [+|-D          <define(s)>]
              [+C            <command(s)>]
              [+|-A|S[f] all|<alert(s)/script(s)>]
              [+|-P[f]   all|<program(s)>]
              [+|-F[f]   all|<file(s)>]
              [+|-O[f]   all|<object(s)>]
               +|-H      all|<host(s)>
              [ALL]

        -c                           syntax check all config files
        -C                           syntax doublecheck all config files
        -d                           disable alert(s)
        -e                           enable alert(s)
        -f                           diff alert/script/program/file/
                                       object file(s)
        -G                           run in debug mode
        -h                           show program help
        -i                           install alert/script/program/file/
                                       object file(s)
        -k                           kill alert daemon (piktd)
        -K                           kill service daemon (piktc_svc)
        -l                           list alert/script/program/file/
                                       object file(s)
        -L                           list alert/script/program/file/
                                       object or host/os/group/alias
                                       command-line item(s)
        -m#                          checksum alert/script/program/file/
                                       object file(s), where # is
                                       checksum level 1-5
        -r                           (re)start alert daemon (piktd)
        -R                           (re)start service daemon (piktc_svc)
        -s                           show alert(s) status
        -t                           delete alert/script/program/file/
                                       object/history/log file(s)
        -T                           run in test mode
        -v                           run in verbose mode
        -x                           execute alert(s)/script(s)
        -X                           execute alert(s)/script(s)
                                       with no wait
        +|-D          <define(s)>    define/undefine define(s)
        +C            <command(s)>   include command string(s)
        +|-A|S[f] all|<alert(s)>     include/exclude (fix) alert(s)
        +|-P[f]   all|<program(s)>   include/exclude (fix) program
                                       file(s)
        +|-F[f]   all|<file(s)>      include/exclude (fix) other
                                       file(s)
        +|-O[f]   all|<object(s)>    include/exclude (fix) object
                                       file(s)
        +|-H      all|<host(s)>      include/exclude host(s)/os(es)/
                                       group(s)/alias(es)
        ALL                          +A all +S all +P all +F all
                                       +O all +H all
```