Extending UNIX System Logging with SHARP

Matt Bing & Carl Erickson – Grand Valley State University

ABSTRACT

System messages in a UNIX system are handled by syslog. The responsibilities of syslog are to filter and disperse program generated messages based on a priority code contained in each message. Filtering with priority codes is not sufficient to generate enough usable information for the system administrator. Utilities which do regular expression parsing of syslog messages typically do not run continuously and thus are limited by a lack of state in detecting potentially important patterns in syslog messages.

SHARP (Syslog Heuristic Analysis and Response Program) improves the monitoring of systems by extending the existing syslog infrastructure with programmable modules. These modules use a library with a simple API to perform near real time analysis based on the messages they register to receive. System administrators can use SHARP to improve the services provided by their systems without the need for constant manual evaluation of message logs. The SHARP system and several modules were tested in a higher education production environment during the spring of 2000. Experience with SHARP indicates that it is stable, reliable, and improves the overall operation of a laboratory while not significantly increasing the workload on the system administrator.

Syslog

The "system logger", or *syslog*, gives programs a standard interface to report interesting events to the administrator. These messages are read by a background daemon and routed accordingly. The data which a program passes to syslog is called a *message*. A message consists of two parts: priority and textual data [9]. The *priority* of a message also contains two parts: an encoded *facility* and *level*. The facility of a message is a general category into which the message fits. The level of a message is a way for the program to rate the severity of the message, typically ranging from emerg to debug. The textual data of a syslog message is a string provided by the program that describes the event being logged.

Programs use library calls to send a syslog message. The library allows the program to choose a facility and level pair, as well as the text describing the message. The library will typically prepend information such as a timestamp, hostname, program name, and PID [7]. The library then delivers the message to the syslog daemon.

The syslog daemon, syslogd, acts as the router for system messages. When it receives a message from a program, it in turn must decide what to do with the it. Most commonly this action involves writing the message to disk, but other potential actions include printing it to the system console, notifying online users, or forwarding the message to another system. syslogd makes these decisions based on a configuration file written by the system administrator. The rules in this configuration file are based entirely on the priority of the message.

Shortcomings of syslog

The standard syslog daemon[1] lacks many important features. These features impact the the reliability of message delivery and the integrity of messages after delivery.

There is no standard structure for writing syslog messages. A cleverly written program could bypass the syslog library calls and write directly to the listening syslogd socket. When syslogd reads this message, it will prepend default priority information and route the message according to these defaults [8]. While the lack of message structure is not critical for system operation, it does not encourage good programming form.

When syslog messages are forwarded over a network the problem of time synchronization arises. There are techniques for network time synchronization, but even short skews of seconds are a problem. When syslogd receives a message from a remote host, it writes the message to disk as it was received and does not provide an additional timestamp. When analyzing messages, this lack of consistent timestamping makes strict ordering impossible.

Some versions of syslog have the ability to route messages based upon regular expression filtering. This allows greater discrimination and handling of messages than is possible with priority filtering. Sophisticated classification and processing of messages is still difficult with regular expression filtering. Extending syslog in this manner violates the UNIX design

[1]This refers specifically to the 4.4 BSD implementation [8].

philosophy of simple tools doing one thing well. Extra processing must be performed with each message, which decreases message handling capacity.

When the syslog system is compiled, the different facilities and levels are hard coded into the system. Most systems have eight different local facilities to give the programmer a set of alternative facilities. Some programs assume that a single local facility is available for exclusive use when this is not necessarily the case. With as few as eight local facilities available, there may be conflicts between programs. There is no way for the programmer to extend this limit without significantly altering the architecture.

When syslogd writes a message to a file, all priority information is lost. After the message has been written, syslogd drops the message, forever losing potentially important forensic data. The priority of the message could be useful in that it shows the state of the program that generated the message. While the text of a message is the most immediately helpful portion, it should not be treated as the only valuable piece.

After the message has been written to a file, any person with write access to the file, authorized or not, may alter the contents of the file. While there is no way to stop any intruder with total access to a system, there are cryptographic protocols designed to detect alterations in log files [3]. These protocols write a sister log file containing cryptographic information to verify the integrity of the messages.

There is currently an IETF working group [2] that is attempting to create a new standard to solve many of these problems.

Extensions to syslog

Shortcomings in syslog and the need for improved message analysis have spawned several utility programs. These programs read syslog messages, perform analysis, and direct the results. These tools are used by administrators to automate analysis of messages.

The simplest and most common type of message analysis is the grep-style filter. This type of program regularly parses syslog messages written to disk for predefined messages. In its simplest form, this program is a shell script spawned from cron that uses regular expressions to grab messages from a log file [6]. These types of utilities have value, but only as post mortem tools. By the time the administrator has seen the output, a significant amount of time may have occurred where a potential problem may no longer be fixable. Imagine an administrator runs a regular expression filter nightly, only to find out the next morning that the company database has been filled for hours, rendering the system unusable.

Another type of tool does the same style of parsing, but in real-time. These utilities, swatch [1] being

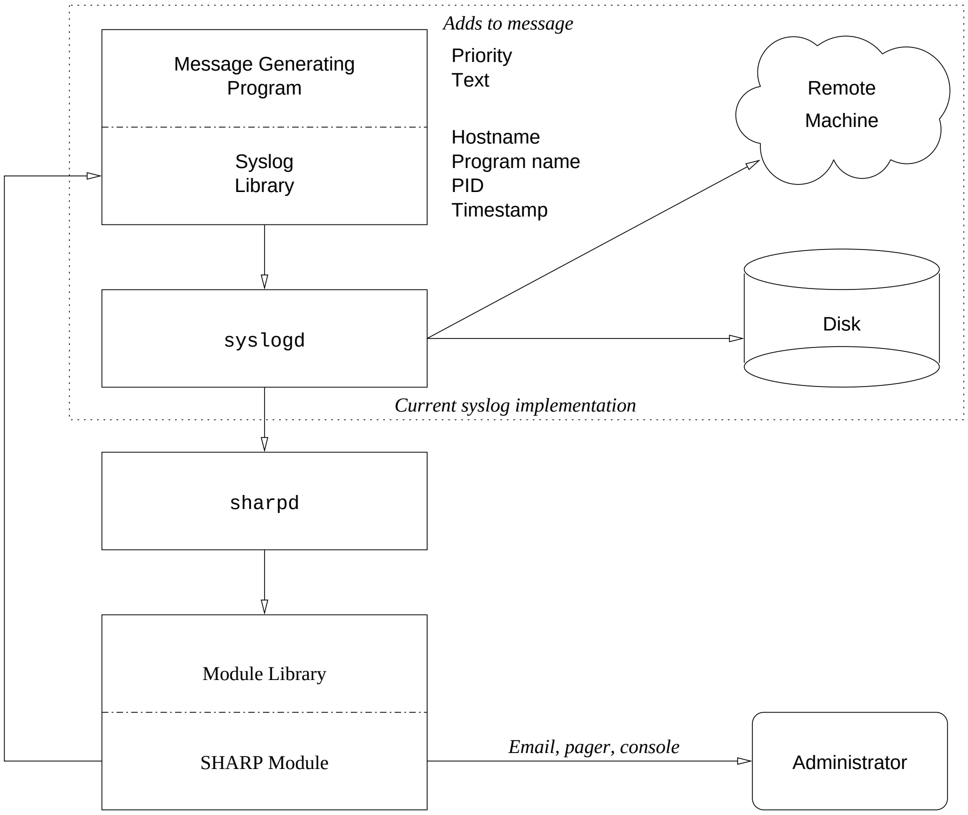

Figure 1: syslog and SHARP

the most prominent, read syslog messages in real-time, perform string based parsing, and direct the output. The limitation of this type of system is the ability to form complex routines based on messages. While you can configure this device to pass messages to an outside program, state is not saved across invocations. Assume an administrator has a swatch filter that dials a pager each time the company database is filled. When a database fills, it usually complains repeatedly, not just with just a single message. With a swatch style filter, the administrator will be paged as many times as the database error message occurs.

SHARP

All of these shortcomings with syslog point to the need for another tool that is able to perform real-time analysis and execute complex heuristics while retaining compatibility with the current style of system logging. The SHARP (Syslog Heuristic Analysis and Response Program) system attempts to address many of these architectural shortcomings. SHARP offers an interface for heuristic analysis programs to wisely utilize system logs. System administrators can decrease the burden of manually reviewing logs and improve the overall fluidity of the system by using SHARP.

Architecture

SHARP was designed for both simplicity and reliability. Simplicity is necessary to not obfuscate an already complex architecture of UNIX messaging. Reliability is necessary to guarantee the proper handling of messages and the overall stability of the system. A single mishandled message could lose information crucial to the administrator.

The SHARP architecture consists of three parts. SHARP modules are registered processes that perform some sort of heuristic analysis. The modules use a simple library interface to connect to sharpd, the SHARP daemon. sharpd communicates directly with the syslog system and receives a copy of every system message from syslogd. The library does the work of connecting to sharpd and passing the received messages back to the module.

Using the library interface, resident SHARP modules can specify the priority of messages they would like to receive. When syslogd receives a message, it passes it to sharpd, which in turn passes the message to only the interested modules. The modules perform their heuristics and can take various actions such as logging, emailing, or notifying the administrator.

When sharpd passes a message to a module, the message has been parsed and placed in a predefined structure to standardize analysis. This structure contains the time the message was received, the priority, the host that generated the message, and the text of the message. Communication with modules occurs over UNIX domain sockets. Using UNIX domain sockets provides further extensibility for replacement with internet sockets. A SHARP system could forward messages to modules residing on separate systems for analysis.

To report its findings, a SHARP module may choose to send a syslog message. If a module reaches a conclusion based upon previous messages, this new syslog message could report its findings at a higher level than the previous messages. This higher level message might be noticed by the administrator, while the previous messages of a lower level may have slipped past. Modules could use syslog messages as a rudimentary version of interprocess communication. Care must be taken by the module programmer not to generate messages that would travel in an infinite loop between the SHARP and the syslog architectures.

Example Usage

For example, a module could track user patterns and report anomalous behavior. There are multiple user entry points into a UNIX system: ssh, telnet, ftp, rlogin, and so forth. Since each point of access is equally valid, it would be insufficient to monitor only one. SHARP is a perfect system to correlate this login information. The SHARP module would indicate to sharpd that it only wishes to receive messages about logins. Over time, the module would build up a database of user patterns such as the time of login and the remote host from which the user is accessing the system. When the module receives information about a new login, it would report anomalous behavior based on a configurable degree of confidence. Suppose a user normally only logs in to the system from a machine local to the office and only during business hours. When the SHARP module sees the same user logging in from a system across the Internet at 3am, it would report the strange behavior.

A SHARP module could be written to handle the problem of an administrator being inundated with pages due to a flood of messages. This SHARP module would maintain a queue of incoming messages to be paged, and throttle similar messages. With this module, the administrator could fine tune the ability of programs to flood the notification path of urgent messages.

A Complete System: nsyslogd and SHARP

As previously shown, the current syslog system is inadequate for a system such as SHARP to operate at full potential. A replacement for syslogd that does not have as many problems can be used. nsyslogd solves many of these problems by extending the functionality of the standard syslog system [5].

nsyslogd uses TCP instead of UDP for network message delivery. TCP, a connection oriented, reliable protocol, is much safer to use on long network paths. TCP is more difficult to spoof than UDP, but not impossible. nsyslogd can also use SSL over TCP, which gives network connections host authentication and data encryption.

nsyslogd uses a hash based algorithm [3] to guarantee message integrity. For each file destination a sister log is created containing chained hashes of the written logs. An external program is executed to check the integrity of the logs against the hash file.

Other syslog replacements have similar features, but nsyslogd has one prevailing feature that is useful to the operation of SHARP: priority preservation. When nsyslogd writes a message to a destination, it includes the priority information. SHARP uses this feature to filter messages for delivery to interested modules only. Without this feature SHARP is forced to flood each message to every module, which in turn is expected to do its own parsing.

sharpd communicates with nsyslogd by means of a UNIX domain socket. nsyslogd can be configured to write all system messages to a socket where sharpd receives them and passes them to interested modules.

nsyslogd was written by Darren Reed and is freely available on the Internet. It has been ported to multiple UNIX platforms. It has proven to be a reliable alternative to the standard syslog that, when paired with SHARP , becomes an even more powerful system.

Modules

Modules utilize the framework provided by SHARP. A module is defined as any program that utilizes the SHARP module interface and receives messages from sharpd. Developers of modules include a single header file and link their module against the SHARP library. The interface is presented along with a simple example that illustrates proper usage.

Module interface

The interface to the module library consists of only three core functions and is designed to be as simple and flexible as possible for the programmer.

void sharp_filter (const char *fac, const char *lev)
: Register interest in messages of facility fac and level lev. A wildcard facility named all is available that matches every facility. This function may be called any number of times, but must be called before sharp_init().

int sharp_init (const char *name);
: Connect to sharpd and register with the corresponding name. 0 is returned on success, -1 on error. This function must be called before sharp_run().

void sharp_run (void (*callback)());
: Begin receiving messages from sharpd. callback() is the function that handles a received message and must be of a particular prototype. sharp_run() is distinct from sharp_init() to allow for future developments in SHARP that perform run-time checks on modules when they connect to sharpd.

void callback (sharp_msg m); ---begin:quotation--->
: The function defined by the programmer that performs analysis on the message passed as a parameter. When callback() returns, the sharp_run() function again waits for another message from sharpd.

The data structure containing the message from sharpd is of the following format:

```
typedef struct sharp_msg {
  time_t time; /* sharpd timestamp */
  char  pri[]; /* priority */
  char host[]; /* originating host */
  char text[]; /* message text */
} sharp_msg;
```

Each string field is of a fixed length defined in the header file for security and robustness. Since this structure is just a copy of a message, the module is allowed to alter the contents of sharp_msg.

The function atexit() is used to force execution of a SHARP library function that deregisters itself with sharpd. The programmer does not explicitly have to deregister the module. Figure 2 shows the state of a module as it connects to sharpd, receives messages, processes them, and finally exits.

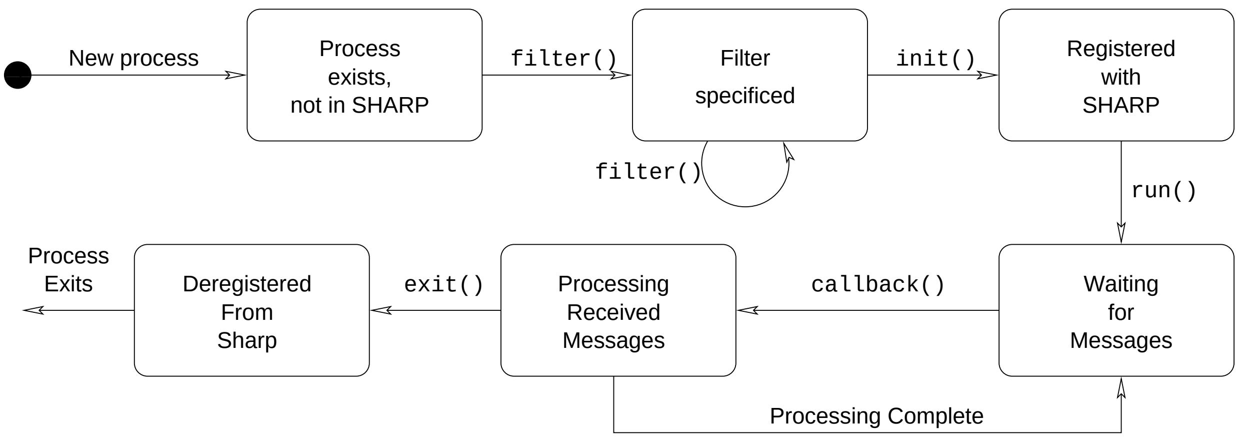

Figure 2: SHARP module state diagram.

Example Module: mark

To fully illustrate the simplicity and flexibility involved with authoring a SHARP module, an example is presented here to introduce the interface to potential programmers. This module is called mark because it reads syslog mark messages and reacts when a problem occurs.

syslogd can be configured to generate a message of facility mark at a predefined interval. This message is a "heartbeat" to show that syslogd is still running. The mark module reads these messages and reacts if it does not receive one at a predefined interval. Listed below is the source code, edited for both clarity and brevity. See Listing 1.

Note that this example is a skeleton and does require extra coding to be fully functional. The example only shows the code necessary to monitor syslogd on a single host. Adding support to the module for multiple hosts is straight forward.

Also included with the SHARP system are utility functions that modules would often have to use, such as sharp_email(), shown in the above example.

Current and Planned Modules

The authors have implemented and tested several SHARP modules. It is hoped that the SHARP web site comes to be a central point for module sharing in the spirit of the Open Source movement. The modules which we have implemented or anticipate implementing include:

- mark The mark module outlined above receives heartbeat messages from any number of syslog daemons on remote machines. If a message is not received in a configurable amount of time, the mark module assumes the remote host or network is down and informs the administrator.
- userpattern A module that gathers statistical data of the system use of users and reports anomalies. For instance, a particular user is known to only use the system during office hours and connects only from a particular host. When the module notices that the user is online at 3am from an off-site host, the module would report this anomaly. This shows the power of a heuristic, stateful approach in monitoring.
- useralert A module that alerts the administrator when a particular user connects to the system. This module can also be used to centralize login records for multiple machines.
- dailyfilter This module emulates the functionality of syslog utility programs. It filters out all syslog messages based upon regular expressions and email the output.

```
#include "sharp.h"
#define TIMEOUT 60              /* threshold seconds */
static time_t last;
void main(int argc, char *argv[])
{
    sharp_filter("mark", "info");
      sharp_init("mark");
    signal(SIGALRM, alarm);
    alarm(TIMEOUT);
    sharp_run( (void*) mark_callback);
}
void mark_callback(sharp_msg m) {
    if(m.time-last > TIMEOUT)
        react();
    else
        last=now;
}
void alarm() {
    /* Did not receive the mark message */
    react();
}
void react() {
    /*
     * Possible reactionary methods include sending an email,
     * dialing a pager, or sending a syslog message.
     */
    sharp_email("admin@localhost", "syslogd died");
}
```

Listing 1: mark module.

problemalert Critical errors repeated more than a threshold number of times over a set period escalate notification (ie paging the sysadmin). Works well with the "all" wildcard.

firewall This module receives firewall logs and analyzes trends and anomalies.

Experience using SHARP

The Computer Science and Information Systems department at Grand Valley State University maintains a network of machines known as the Experimental Operating Systems Laboratory, or EOS lab. The EOS lab consists of 28 Linux workstations and one central file and authentication server. Because of its classical network architecture, this environment was optimal for testing a working implementation of SHARP. Due to the enormous amount of system messages, they are generally ignored by the EOS administrators and only used as an autopsy instrument.

Results

It was determined each workstation generates an average of 250 system messages per day. The main server for the lab generates an average of 36,000 syslog messages per day. The total number of system messages generated in the EOS lab averages just above 42,000 per day. If this rate were normally distributed over the course of 24 hours, a message would appear every 2 seconds, way beyond the sensory capacities of any human to manually monitor. These statistics are highly site and service dependent, but should offer a general idea as to message rates.

The SHARP system has worked smoothly on this network. The concept of a centralized logging system seems to be the best operating environment for SHARP. Individual host installations of SHARP do not scale well when dealing with large networks. It is much easier to configure a new system to forward all syslog messages to a central logging system than to install SHARP and the various corresponding modules.

The EOS lab is a very small network in comparison to larger installations. SHARP is expected to scale well to these much larger systems. Due to its nature, the architecture is well suited for scalability. The centralized logging host could forward messages to instances of SHARP on different systems. Each system could have a unique set of modules. The socket architecture built into SHARP allows seamless changes between a local ==>[ignored: sc]<== UNIX-domain socket and a network socket. SHARP would thus easily support applications such as distributed anomaly detection.

Future Work

With the experiences gained from using SHARP, the authors have noted a few features that would both aid module developers and improve overall performance of the system.

- Patches for other implementations of syslogd to pass priority information with messages. SHARP users will not be forced to use nsyslogd.
- A global configuration file, such as /etc/sharp.conf, would define variables that modules could use. For instance, a module would call sharp_get_var(ADMIN) to get the email address of the administrator defined in the configuration file. This way potentially dynamic information would not have to be hardcoded into modules.
- Support for other languages besides C that have better text processing capabilities. Ideally a Perl module will be the first to be implemented.
- Support threadSHARP library calls.

Conclusion

The traditional UNIX message facility is inadequate for the active monitoring and response required of system administrators of even moderately large networks. Working with the existing syslog infrastructure, SHARP can improve the quality of monitoring, while at the same time reducing the burden on the system administrator.

Our goals for the design of SHARP have been met. SHARP is compatible with syslog, and hence requires no changes to future or existing message generating programs. SHARP 's architecture is simple and clean, promoting extensibility and sharing of modules. The relatively small (approximately 2000 lines of code) implementation of SHARP is indicative of its simple and clean design.

SHARP's modules allow for continuous monitoring and the maintenance of state, which in turn support the implementation of monitoring heuristics not possible with other extensions to syslog. The modules we have implemented work as expected and have proven their utility in a production environment.

We have tested SHARP in a production environment of a network of Linux workstations and a single file/authentication server. Our testing has shown SHARP to improve the monitoring of a network of hosts by supporting a pro-active form of monitoring by the system administrator.

Availability

SHARP and a few packaged modules are available in source code form from the official web site: http://www.csis.gvsu.edu/sharp. This code is available under the BSD license [4]. Both the SHARP daemon and the modules have been compiled and tested on multiple platforms.

Author Information

Matt Bing will receive his M.S. in Computer Science from Grand Valley State University the day after the conference. He currently resides in Ann Arbor, Michigan working on intrusion detection systems for Anzen Computing. His email address is matt@csis.gvsu.edu .

Carl Erickson is an associate professor in the CSIS department at Grand Valley State. Last year he succumbed to start-up fever and is currently on leave working as a software architect with XiphNet, Inc. Reach him via email at erickson@csis.gvsu.edu .

References

[1] Stephen E. Hansen, E. Todd Atkins, "Centralized System Monitoring With Swatchp" *LISA, 1993*.

[2] "Security In Network Event Logging", August, 2000 IETF Draft available at http://www.mail-archive.com/syslog-sec@employees.org/msg00466.html .

[3] Bruce Schneier, "Secure Audit Logs to Support Computer Forensics," *ACM Transaction on Information and System Security*, v.1, n.3, 1999.

[4] BSD License http://www.freebsd.org/copyright/license.html .

[5] Darren Reed, nsyslog, http://cheops.anu.edu.au/avalon/nsyslog.html .

[6] FreeBSD /etc/security, http://www.freebsd.org/cgi cvsweb.cgi/src/etc/security .

[7] BSD source lib/libc/gen/syslog.c .

[8] BSD source usr.sbin/syslogd/syslogd.c .

[9] BSD source sys/sys/syslog.h .