Pp. 145-154 of the *Proceedings of LISA ’02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

Process Monitor: Detecting Events That Didn’t Happen

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

The successful operation of a large scale enterprise information system relies, in part, on the regular and successful completion of many different tasks. Some of these tasks may be fully automated, while others are done manually. One of the challenges we face is detecting when one of these tasks fails (often silently) or is forgotten. While you will eventually learn of these omissions, it is much better to have the system detect them rather than your users! This paper discusses how we implemented a system that watches what we do and reminds us when we (or our computers) forgot to do something.

Introduction

Inspector Gregory: “Is there any other point to which you would wish to draw my attention?”
Holmes: “To the curious incident of the dog in the night-time.”
“The dog did nothing in the night-time.”
“That was the curious incident,” remarked Sherlock Holmes.
From *The Adventure of Silver Blaze* by Arthur Conan Doyle.

At Rensselaer, we manage many of our system and site administration tasks[1] with an Oracle database. For example, we take a data feed from Human Resources to automatically create and expire Unix/email and Windows 2000/Exchange accounts. One aspect of this is that we have many tasks, some run via cron and other scheduling mechanisms, and others run by hand on a regular basis. These tasks generate configuration files [7], [3], web pages (phone directory) [5], process accounting and billing records, update the Active Directory server, and many other things.

One of the problems that we face is knowing when something that is supposed to happen did not. This may be due to a transient file server failure, configuration problems, the failure of a daemon, or simply someone forgetting to do some periodic, yet infrequent task. There are a number of monitoring and logging tools available, from those built into systems such as syslog and programs to help process logs such as *Swatch* [10]. There are also tools that monitor network traffic and system activity such as *Peep* [9] and others. In general, however, all of these systems are looking for things that *are* happening but, like Sherlock Holmes, we are interested in those things that did *not* happen. An earlier project to monitor workstation usage patterns [4] briefly discussed detecting failed workstations by a lack of usage data, but was not pursued. Some other projects to measure system performance via statistical analysis [11, 2] don’t really apply to very low frequency events.

[1]Originally, I was calling tasks “processes,” but this was causing some confusion on the part of some readers with Unix (or other system) processes. There are still many references to “process” in table and function names, but the intention is to refer to a “task.”

One of the things that I wanted to avoid was writing and maintaining lots of configuration files. Instead, I wanted tasks to report in to a central server when they completed successfully, and then have a nice interface to identify new tasks and quickly set the frequency at which they should reoccur. After that, I don’t want to have to think about that particular task again. Given our heavy use of Oracle in maintaining our system, and that many of the things I was interested in monitoring were already accessing Oracle, an obvious approach for me was to use the database for all of the heavy lifting.

Task Monitor

With that, the Task Monitor project was born. When I use the term “process,” I am not referring to a Unix process, but rather a specific task such as “load printer accounting records,” “update online directory files,” “propagate password changes to Windows,” [8], etc. These may actually be an Oracle job, or part of a job, or a script run out of cron, or even something running on a Windows server.

The information on a task is stored in an Oracle table with the name Process_Monitor. The description of this table is broken up into several parts and is included in the appropriate section of the paper. In Figure 1, we have the rough architecture of the Task Monitor system. At the center of things is the Task_Monitor package, which acts as an interface between the different tasks and the Process_Monitor database table (labeled Task_Monitor in the diagram). Tasks communicate via a number of different methods. Some, such as the Student Upd package, are running on the oracle server and communicate directly.

Others, such as the Generate_File based modules, connect via SQL*NET. We may also add other interfaces such as syslog or SNMP.

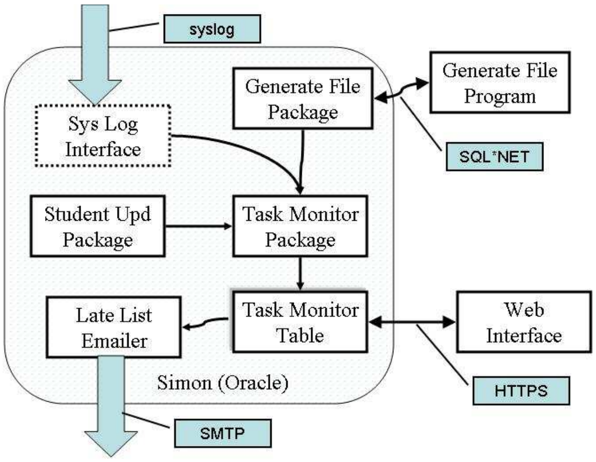

Figure 1: Task monitor architecture.

We also have different ways of getting information out of the Task Monitor system. A program on the database machine generates email notifications and sends them to interested parties. We also connect via a secure web server for administrative purposes.

Administrative Interface

One of the key parts of this system is the administrative interface, which allows us to set the options for each task. This is implemented via a secure web server.

In Figure two, we have a screen capture of the main web page for the Task Monitor system. This allows you to display different sets of tasks. You select the things you are looking for, and press the "LIST" button. All of the attributes are combined, so the more you select, the more restricted the selection. The first option is to find late or "not late" tasks. Next, you can select the family from the pull down list. There is a also a special "NONE" entry that will limit the results to those tasks that are not in a family. You can also select based on those tasks with or without a run delta or schedule, and those tasks that are marked as inactive. Finally, you can restrict the tasks to just those owned by you. (This is run as an administrator; less

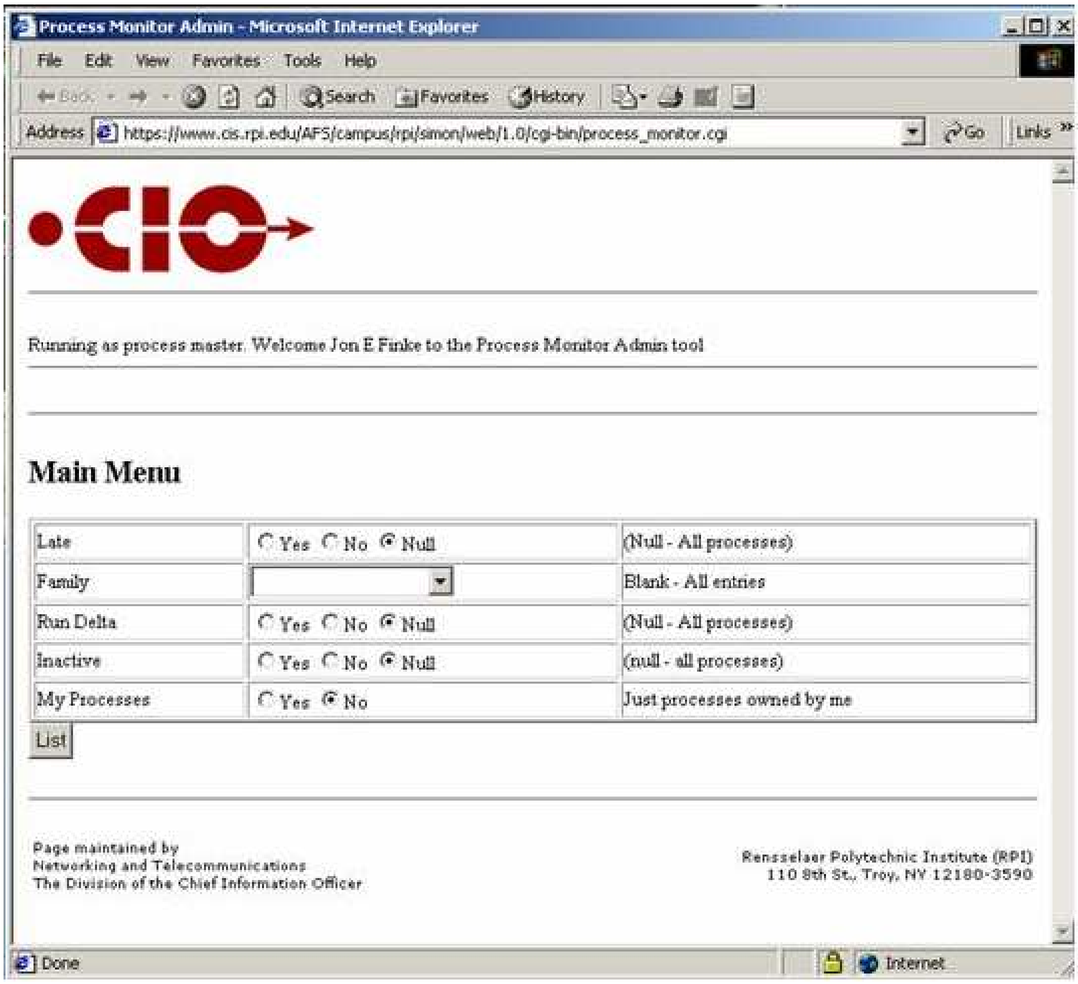

Figure 2: Main web page.

privileged users will just get their own tasks.) A sample of this list can be seen in Figure seven.

In Figure three, we have a sample web page of the ‘‘Logins-Oracle IDs’’ task. The objective of this task is to create Oracle accounts based on changes in the Logins table. This task is considered part of the ‘‘daily run,’’ a set of activities performed by our User Services staff. From this page, we could move the task to another family using the pull down list, or create a new family by entering the name in the ‘‘New Family’’ box. (This box does not appear if you are not an administrator; you are limited to existing families.) In this case, we don’t care what system this task runs on, only that it is run; so we leave the ‘‘System’’ box empty. The person who normally does this task is Judy Shea, so we have listed her as the owner. If we wanted to let other folks know if this task was late, we could provide a list of email addresses in the ‘‘Contact List’’ box. The next thing we can specify is the ‘‘Run Delta,’’ which is specified as DAYS HOURS:MINS:SEC. In this case, we want this to be run every 28 hours (one day and four hours). This gives Judy a little bit of flexibility in when she does the actual run. The ‘‘Notify Delta’’ is like specified like the ‘‘Run Delta’’ and controls how frequently we report a missed task. Lastly, we can mark as task as inactive, which turns off all notification.

The next part of the page reports on information collected at the last run. The ‘‘Next Run’’ is the time and date when we next expect this task to be run. If that time was passed, this would be in bold face and be marked as late. (This is only set if there is a ‘‘Run Delta’’ set.) The ‘‘Last Run,’’ which is always available lists the time and date of the most recent run. Currently, the only way to get a task into the system is to run it, so there is always a ‘‘Last Run’’ entry. Next up is when we last notified someone about a late process, and when we expect to send the next notification (assuming a run has not been completed.) There is also space for a free format comment on the task.

When a run is recorded, the system attempts to capture the host OS username and the hostname. There is also an option when recording a task to include a comment; this is task specific. We also record the Oracle user and what Oracle package made the call.

Identifying a Task

A lot of the tasks we are interested in monitoring are site wide. For example, the process that regenerates the directory web pages only needs to run on a single system and write a file into our central file server. There is no need to run it on each of our production web servers, as they all use the same central file server. In other cases, however, we want to be sure that each server is reporting in. An example of this would be the process that collects the printer accounting logs. We want to ensure that each print server is

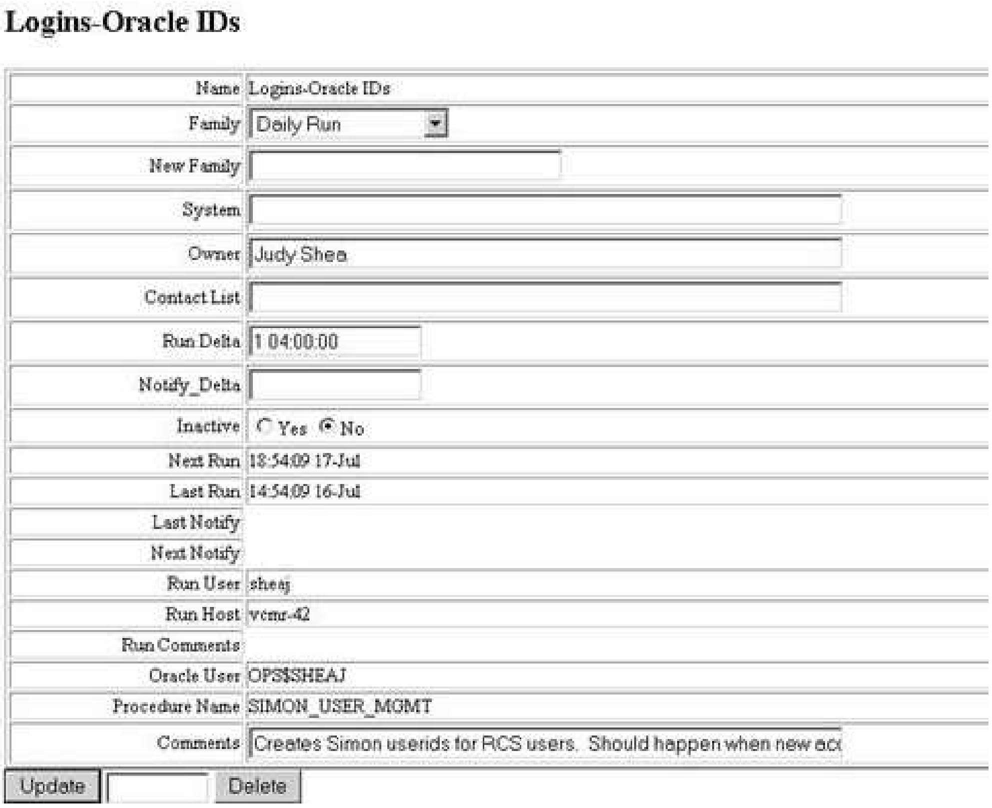

Figure 3: Sample task web page.

reporting its activity on a regular basis. It may also be useful to identify the user that is running the process.

For the purposes of this project we identify a process by a process name and the system on which it runs. In this way, if the same process runs on two different machines, it will be considered two different tasks for the purposes of monitoring. This has not been a problem with the site wide tasks, as they are generally run via cron or some other trigger on a single designated machine and by the same user (daemon or equivalent). Since most of the testing and development takes place on a different machine, this isolates the development and test runs from the production runs. We also record the name of the person who ran the task, but this is currently not used to distinguish tasks.

Parenting a Process

In order to help with sorting and grouping, each process can be assigned to a "Family." These are general categories such as "Accounting," "Daily Run," "File Gen," etc. When a new process is entered, it will not have a family assigned to it. This works well, as the administrative web tool can display all tasks in a family, or those without a family. This provides a quick and easy way to identify new tasks.

When we encounter a new process, we assign it an owner and a family. We can also link it to a service in our ServiceTrak [6] so a the page displaying service information can also include details on some of these tasks. Once a process has an owner, that owner can use the same web tool to finish the setup by assigning a run delta or schedule, or just marking it as inactive.

Reporting a Task

The first challenge of this project was to find ways for tasks to report that they ran. When a process reports in via one of the methods described below, we first see if we have an existing process record. If not, we create a new one; otherwise, we update some of the information and, if there is a run schedule or delta, we then calculate the next run time and save the record. If there were previous error conditions, we clear them as well.

Direct PL/SQL Procedure Call

A number of the tasks that we want to watch are written entirely in PL/SQL and are run on the database

Name	Type	Size	Description
Entry_Id	Number		A unique key to identify this record.
Name	varchar2	32	The name or external identifier for this process.
System_Id	Number		The unique identifier of this system in the hostmaster and service database.
Run_Host	varchar2	128	The hostname where this last ran. Useful when the System_Id can not be determined.
Run_User	varchar2	32	The name of the host system user who ran the process (if available).

Table 1: Process_Monitor table – identification.

Name	Type	Size	Description
Family	varchar2	32	An identifier used to group tasks for display and reporting.
Owner	Number		The internal identifier of the person who "owns" this process.
Service_Id	Number		The internal identifier of the service (see ServiceTrak) that this process supports.
Run_Delta	Number		The maximum allowable time in seconds between the last run and the next run of this process.
Run_Schedule	varchar2	128	A crontab format schedule.
Inactive	varchar2	1	A flag indicating that the current entry is inactive and should be ignored.

Table 2: Process_Monitor table – parenting.

Name	Type	Size	Description
Oracle_User	varchar2	32	The name of the oracle user. This is always available.
Proc_Name	variable	65	The name of the oracle procedure that logged this run.
Last_Run_Time	Date		The time and date when this process last ran.
Next_To_Last_Run_Time	Date		The previous value. Useful in calculating the spacing between runs.
Next_Run_Time	Date		The date and time when we next expect to see this run. This is the key trigger for notification.
Run_Comment	varchar2	255	An optional comment set by the caller that will be displayed in status messages. Unlike the Error_Flag, this does not trigger notifications.

Table 3: Process_Monitor table – reporting.

machine. For example, we have a routine that we run daily to compare the Simon Banner_Students table with the student base table (SGBSTDN) on our administrative machine. This is typical of many similar routines, and it has two optional parameters, a Target_PIDM which allows us to update a record for a specific person, and a StopCount which stops the update after a set number of records (this is handy for debugging). When neither parameter is set, we want to record the fact that the routine ran to completion.

In Figure four, we have a code segment of the routine that checks the student base table on Banner (our student record system) and updates the Simon student table. The two cursors are written so that if they are opened with a value for the PIDM, they will return just the single record for that person; otherwise, they will return a full set of records. There is also an option to stop after a set number of rows. Once the loop is complete, and if we did *not* exit due to the stop count, and we were *not* doing the check on behalf of a specific individual, we will record this run using the Process_Monitor_Record.Mark_Proc procedure. This procedure will obtain the user, hostname, and other information from the database environment. The only thing we need to give it is the task name (Target_Name) and the name of the current package. In this way, whenever we do the general update, it will record the fact that it ran; it doesn't matter how we did it.

Generate_File Definition

A number of the tasks that we want to watch are run via our Generate_File system (described in the LISA 2000 Proceedings). These tasks are generally reading or writing files, using stored procedures in the database. When we develop a new file target, we store the PL/SQL source code in a file, and read that into the database using SQL*PLUS.[2] At the end of this file, we include a block of PL/SQL to register the new targets with the system. This is done with a procedure called Add_Target_Simple or Add_Target_Complex.[3]

In Figure five, we have a fragment of the file used to generate some web pages documenting our network routers and subnets. In the package, we define some entry points that will be called by the Generate_File system. At the end of the sample, we register three things: a simple target (web_routers) that will call the Get_Router_Html routine to generate a list our our routers into the primary_routers.html file, a complex target that will generate a set of files based on Get_Network_List routine, and, finally, a special target, Add_Process_Record, that will record the fact that the first two entries have been executed and have completed. This last routine makes it trivial to record the completion of any Generate_File run by simply adding the Generate_File.Add_Process_Record to the end of the registration statements.[4] This has the added advantage of not having to modify the source code that doing the direct PL/SQL call would require.

The Add_Process_Record routine actually calls the Add_Target_Simple routine to register a special

[2]SQL*PLUS is a command line interface to Oracle. One of the options is to read from a file and pass the information to Oracle for processing.

[3]Since the original paper, we have added several other kinds of targets in addition to these.

[4]The Generate_File registration routines will default to the target name specified in earlier calls if not provided on later calls.

```
procedure Sgbstdn_Full(Target_Pidm in number,
               stop_count in number)
is
    Banner      Sgbstdn_Scan_Curs%RowType;
    Simon       Simon_Scan_Curs%RowType;
    Act_Cnt     number := 0;
is
    Open Sgbstdn_Scan_Curs(Target_Pidm);    -- Full scan if NULL
    Open Simon_Scan_Curs(Target_Pidm);  -- Ditto
    loop
. . . (Details of processing omitted)
        exit when Act_Cnt > Stop_Count;
    end loop;
    close Sgbstdn_Scan_Curs;
    close By_Pidm_Curs;
    if Act_Cnt > Stop_Count
    then
        dbms_output.put_line('Stopped due to stop_count');
    elsif Target_Pidm is null
    then
        Record.Mark_Proc(Target_Name => 'Student-Sgbstdn',
                 Procedure_Name => 'BStudent_Maint');
    end if;
end Sgbstdn_Full;
```

Figure 4: Recording run from a PL/SQL procedure.

target that just records the fact it was called, and exits after writing a few lines to stdout. Since it is called from within the Generate_File environment, it can get all of the information it needs for recording from that, and we don't need to pass in any parameters. It also prepends GENERATE_FILE- to the target name to come up with the name to record.

Special Generate_File Target

We still have tasks that we are interested in watching that are not written in PL/SQL or using Generate_File. These might be older file generation programs, or just shell scripts run out of cron. The Generate_File program has the ability to pass a parameter to the processing routine. We combined this, with a variant of the previous routine to have a new Generate_File target that will record anything, with a prefix of MANUAL-.

In Figure six, we have a simple shell script that is run from cron to generate the /etc/printcap file for our system. Assuming that the program exists, and runs successfully, we then call Generate_File with the target Record_Process to record the completion of the task MANUAL-Printcap.

Notifications

Although it is all well and good for the database to know when a process is overdue, we really need some way of letting the appropriate people know about this. It is important, however, that the mechanism used is appropriate for the type of failure and the urgency of the process. For example, when the process feeding password changes into our Active Directory server fails, we want to get the service restored within minutes. But if the billing run for our backup service is a day late, it isn't a major problem; we normally run this two or three times a year.

Most of the tasks we are monitoring run once or twice a day. As a result, we are currently only

```
define name=GENERATE_NETWORK_LIST
prompt Create Package &NAME
Create or Replace Package &NAME as
--
-- Generate web pages documenting our network.
-- Define the standard interface
procedure Get_Network_List(Fname out varchar2, Dbmsout out varchar2);
Procedure Get_Router_Html(result out varchar2, p1 in varchar2, p2 in varchar2);
Procedure Get_Subnet_Html(result out varchar2, p1 in varchar2, p2 in varchar2);
... Package definition omitted
end &NAME;
/
begin
Generate_File.Add_Target_Simple(
    target => 'web_routers',
    filename => 'primary_routers.html',
    get_data_rtn => '&name..Get_Router_Html');
Generate_File.Add_Target_Complex(
    get_attr_rtn =>'&NAME..Get_Network_List',
    get_data_rtn =>'&NAME..Get_Subnet_Html');
Generate_File.Add_Process_Record;
end;
/
```

Figure 5: Recording Run from a Generate_File target.

```
#!/bin/sh
#
# Script to Generate /etc/printcap
#
FILE_GEN=/campus/rpi/simon/directory/2.0/@sys/bin/Generate_File
PCAP_GEN=/campus/rpi/simon/printmaster/1.0/@sys/bin/etcprintcap
#
if [ -x $PCAP_GEN ]; then
    $PCAP_GEN
    if [ $? -ne 0 ]
    then
        echo "Error in printcap generation!!!!!"
        exit 1
    fi
    $FILE_GEN -target Record_Process -par2 Printcap
fi
```

Figure 6: Recording run from a generic Generate_File target.

checking for "late" tasks a few times a day, generating a report, and mailing it to interested parties. At present, we don't have anything in place to escalate a problem that has not been repaired in a timely fashion. So far, the notifications are unique and infrequent enough that they are not ignored.

In Figure seven, we have a notification email from the system. This is actually sent as an HTML page, instead of a plain text message. While this might be annoying to some, I already had the code to generate the late list as a web page in the administrative tool, and I just had to wrap it in a call for Generate_File. This has the added advantage that the buttons visible on the email message are fully functional, in that I can press one and see the detailed information for that task. In this case, I have three tasks that are late, printing and disk billing (that was desired actually, as we were deferring some revenue to the new fiscal year) and the Generate_File run that produces our Building Directory web pages. In this case, the normal run had failed due to the administrative database being down for a backup.

Conclusions

The Task Monitor tool has proven to be very useful in detecting things that should be happening and failed for some reason. This is especially useful for the infrequent jobs that are easy to forget. Because it is so easy to add monitoring to existing tasks, the number of things that we watch has grown quickly.

Existing Limitations

Not all aspects of the original design have been implemented. At present, we are only checking for "late" processes twice a day. Before we can make this a more frequent occurrence, we need to implement the notification limits. While it may be useful to check for late processes every two minutes, I don't want to get an email every two minutes when a monthly task is a day late.

Another part we have not implemented is dealing with non regular, but recurring schedules. A number of our business applications do not run on the weekends. The intention to handle these would be to support cron style schedules, and figure the next run time based on the

Name	Type	Size	Description
Contact_List	varchar2	128	A list of email addresses to contact when problems are detected.
Error_Flag	varchar2	128	An optional error message set by the process. Results in immediate notification.
Notify_Delta	Number		The minimum time in seconds between notifications when an error has been detected.
Last_Notify_Time	Date		The time and date when notification was last attempted.
Next_Notify_Time	Date		The time and date when notification will next be attempted if a successful run has not occurred.

Table 4: Process_Monitor table – notifications.

Figure 7: Sample "late" message.

cron format schedule plus the run delta. This would also make it easier to handle manual operations that we expect to be done each business day. This still does not handle holidays; this needs some more thought.

Future Directions

All of the tasks we are currently monitoring with this system are either directly accessing the database, or have the Generate_File program available. However, in order to help track activity on other (Unix) systems, a syslog or SNMP interface to allow other things to occasionally report in might be very useful.

As the number of system specific tasks, such as the last run of CFEngine [1] on a machine, grows, some automatic classification and run delta assignment would remove the bottleneck of having to assign an owner, family, and schedule information for each new service. The ability to designate a particular task entry as a "prototype" for new tasks of the same name and different system identifier would make this very easy.

Other notification methods need to be explored. The ability to generate syslog or SNMP messages and direct them to other monitoring tools could be very useful. This could in turn generate pages, or be directly incorporated into this system.

Another extension to this project is as basic reminder system. This would just require a tool to manually enter a new process, and directly set the Next_Run_Time value. This might be used to remind people to reset annual allocations or renew licenses and service contracts.

We also have a number of tasks that are started on response to some user request, such as a quota change request. One approach would be to have the request also set the "next run" time for the quota change task. However, this approach might run into problems if people keep making new requests before the timeout is detected. A different approach is to be able to make periodic "empty" requests that will require that the task finish all queued work. Both options need some consideration.

Several of our file generation scripts are run out of cron. These are basically shell scripts that run the Generate_File program with different targets. Sometimes one or more of these will fail, possibly due to a server being down, or PL/SQL packages that need to be recompiled. One possible approach would be to add a "rerun" flag to the shell script that would be passed to the Generate_File program. If set, another special target could be added that would have Generate_File skip the run if the target was not late. With this, if there were some problems, a person could just run the shell script with the "rerun" flag, and only those targets that were late would get regenerated.

References and Availability

All source code for the Simon system is available on the web. Please refer to the following URL for details: http://www.rpi.edu/campus/rpi/simon/README.simon .

In addition, all of the Oracle table definitions as well as PL/SQL package source are available at http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html .

Although this is implemented in Oracle as part of the Simon system, there is very little that requires Simon or even Oracle. Just about any relational database would be able to handle the moderate processing and database needs for this system. Given our starting point, most of our examples are deeply tied to Simon, but with alternate interfaces such as syslog and snmp, there is no reason why this could not be deployed without Simon or Oracle.

Acknowledgments

I would like to thank Marcus Ranum for is shepherding of this paper, as well as Deb Wentorf for her proofreading and editing. I also want to thank Rob Kolstad for his excellent (as usual) job of typesetting this paper.

Author Biography

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming, with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past 11 years. He is currently a Senior Systems Programmer in the Networking and Telecommunications department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. When not playing with computers, you can often find him building or renovating houses for Habitat for Humanity, as well as his own home.

Reach him via U. S. Mail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via http://www.rpi.edu/~finkej.

References

[1] Burgess, M. "A Site Configuration Engine," *Computing Systems*, 8(1):309, MIT Press, Winter 1995.

[2] Burgess, M., "Theoretical System Administration," *The Fourteenth Systems Administration Conference (LISA 2000)*, p. 1, USENIX, December 2000.

[3] Finke, Jon, 'Automating Printing Configuration," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pp. 175-184, USENIX, September 1994.

[4] Finke, Jon, ''Monitoring Usage of Workstations With a Relational Database,'' *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pp. 149-158, USENIX, September 1994.

[5] Finke, Jon, ''Institute White Pages as a System Administration Problem,'' *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pp. 233-240, USENIX, October 1996.

[6] Finke, Jon, ''Automation of Site Configuration Management,'' *The Eleventh Systems Administration Conference (LISA 97) Proceedings*, pp. 155-168, USENIX, October 1997.

[7] Finke, Jon, ''An Improved Approach to Generating Configuration Files from a Database,'' *The Fourteenth Systems Administration Conference (LISA 2000)*, pp. 29-38, USENIX, December 2000.

[8] Finke, Jon, ''Embracing and Extending Windows 2000,'' *The Sixteenth Systems Administration Conference (LISA 2002)*, USENIX, November 2002.

[9] Gilfax, M. and Alva Couch, ''Peep (The Network Auralizer): Monitoring Your Network with Sound,'' *The Fourteenth Systems Administration Conference (LISA 2000)*, p. 109, USENIX, December 2000.

[10] Hansen, Stephen E. and E. Todd Atkins, ''Automated System Monitoring and Notification with Swatch,'' *USENIX Systems Administration (LISA VII) Conference Proceedings*, pp. 145-156, USENIX, November 1993.

[11] Hoogenboom, P. and J. Lepreau. ''Computer System Performance Problem Detection Using Time Series Models,'' *USENIX Systems Administration (LISA VII) Conference Proceedings*, p. 15. USENIX, November 1993.