RI 11013, 10 August 2011 **Computer Science**

IBM Research Report

Correlating user activity with system data for fast detection and diagnosis of system outages

Vijay Mann
IBM Research - India
vijamann@in.ibm.com

Anilkumar Vishnoi
IBM Research - India
avishnoi@in.ibm.com

IBM Research Division Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Abstract

Human error has been identified one of the major factors behind system outages and network downtime in a number of previous research papers and surveys. Gartner statistics show that almost 40% of unplanned application downtime is caused due to operator errors such as unintentional changes to network configuration resulting in a network outage, patch installations, service restart,etc. Yet, system admin activities on production IT systems are rarely properly logged and monitored. Existing tools to track user activities either produce too much information without any hints of a potential outage scenario or too little information to be useful in a meaningful way.

In this paper, we describe the design and implementation of iTrack - a framework for monitoring user activities and correlating them with system data for fast detection and diagnosis of service outages. iTrack makes use of commonly available native monitoring and diagnostic utilities on operating systems to monitor systems events as well as system admin activity, correlates these two sets of information and categorizes the activity as potentially abnormal or harmful based on their impact on the system in terms of file system, network and memory activities. Our results confirm that iTrack overhead in terms of CPU time, activity completion time and data generated is within the tolerance range of most production systems. In cases, where the overhead was found to be unacceptable, we detect the underlying cause and provide solutions which improve performance by up to 20% to 90%, in terms of managed sever and iTrack server CPU utilization, respectively and by up to 2 times in terms of completion time of certain system admin activities on the managed server.

I. Introduction

Outages of production IT services result in huge revenue or business loss for enterprises. Human error has been identified one of the major factors behind system outages and network downtime in a number of previous research papers [1]–[4]. Furthermore, repairing such mistakes has been found to be time consuming [4]. The characteristics of the mistakes are usually environment specific, but the most common mistakes include software misconfiguration and improper deployment of new or upgraded software [5]. Gartner statistics [6] show that almost 40% of unplanned application downtime is caused due to operator errors. Other surveys [7]–[9] have found human configuration errors to be behind 50-80% of data center outages and network downtime.

Even though the role of human error behind system outages has been emphasized as above, system admin activities are rarely properly logged and monitored. This further aggravates the problem as the root cause is usually not detected and both the duration and frequency of the outage increases as a result. There are tools that either track the problem symptoms (e.g. a network port going down or the death of a critical process or excessive resource usage) or track all system admin activities (shell history, etc). However, none of these tools correlate these two sets of information to find out who caused what. Tools to track user activities such as shell history file, audit trails [10], terminal typescripting using the "script" command [11], or commercial tools such as ObserveIT [12] either produce too much information without any hints of a potential outage scenario (terminal typescripting, audit trails, ObserveIT) or too little information (shell history file) to be useful in a meaningful way.

On the other hand, products that offer change management solutions, track configuration changes in the system by creating a baseline after scanning the entire file system. Configuration changes can be figured out after comparing with the next system scan report. TripWire [13] is one such product suite. The drawbacks of such solutions is that it is is not clear how many times a snapshot should be taken and compared with a baseline. Typically many configuration changes might have been applied between two snapshots. Hence, it is difficult to pinpoint the exact configuration change that might have led to an outage. Since these tools just report the configuration changes, but do not report on the user action or process that made the change, a full diagnosis may not be possible.

There is a need to track all system admin actions on a system and the associated side effects of such actions and correlate this data with potential outage symptoms to be able to quickly detect any potential system outage and identify its root cause. This will help reduce both length of the outage as restorative action can be taken quickly after an alert and frequency of the outage since the undesirable side effects of a particular change activity will be known and recorded, and repeated occurrences will be prevented.

In this paper, we describe the design and implementation of iTrack - a framework for monitoring user activities and correlating them with system data for fast detection and diagnosis of service outages. We revisit the various standard diagnostic utilities provided by Unix like commodity operating systems. We find that most of these utilities provide valuable information about specific aspects of an operating system but they continue to be used as point solutions by system admins without a unifying framework that aids the overall diagnosis of an outage. iTrack makes use of such commonly available monitoring and diagnostic utilities on operating systems. It monitors systems events as well as system admin activity, correlates these two sets of information and categorizes the commands as potentially abnormal or harmful based on their impact on the system in terms of file system, network and memory activities or

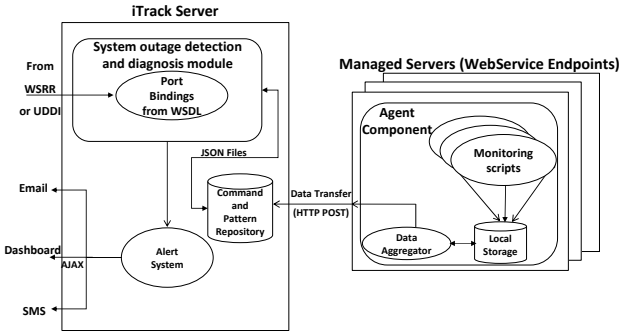

Fig. 1. System Architecture

based on their parameter values. It compares the commands and their associated system impact against rules such as those specified in a firewall policy or a Web Service Definition Language (WSDL) document in order to detect a possible service outage. We make the following contributions in this paper:

1) We formulate key requirements for a monitoring and diagnosis engine based on our interactions with various data center system administrators.
2) We present the design and implementation of the iTrack monitoring and diagnosis framework. iTrack makes use of commonly available monitoring and diagnostic utilities available on most operating systems and does not require changes to existing infrastructure. This was one of our key requirements and this significantly reduces the deployment and installation effort required. We describe the algorithms used by our system that make use of process hierarchy information for identifying processes to be traced and for parsing the trace data.
3) We evaluate the monitoring and diagnosis overhead of iTrack through various test scenarios. Our results confirm that iTrack overhead in terms of CPU time, activity completion time and data generated is within the tolerance range of most production systems. In cases, where the overhead was found to be unacceptable, we detect the underlying cause and provide solutions which improve performance by up to 20% (in case of managed sever CPU utilization) and by up to 90% (in case of iTrack server CPU utilization) or by upto 2 times (in terms of compeltion time of certain system admin activities on the managed server).
4) We describe various techniques that we are currently working on to further reduce the performance overheads of iTrack.

The rest of this paper is organized as follows. Section II presents the requirements for iTrack and describes our methodology. We present the design and implementation of iTrack in Section III. Section IV presents an evaluation of iTrack. Section V presents an overview of related research. We summarize our findings and give an overview of our current and future work in Section VI.

II. Requirements and Methodology

Based on our interactions with various data center system administrators and business unit heads of a large telecom provider in India, we first came up with some key requirements of a monitoring and diagnosis engine:

- **R1.** It should have minimal overhead (not more than 10-12%) during regular day-to-day operations. Administrators were willing to tolerate a slight increase in the overhead if it happens only during the change management window.
- **R2.** It should not require lengthy and complex installation and configuration, especially on the managed servers since there are thousands of them in any data center.
- **R3.** It should work across major Unix* like operating systems without any major changes as the data centers tend to have various flavors of commodity operating systems (e.g. various Linux distributions such as Red Hat vs Suse, AIX 5.1 vs AIX 5.3, etc) running on various managed systems.
- **R4.** It should be able to export a rich visualization of its monitoring and diagnosis in real time which highlights the potential outage and its associated root cause.
- **R5.** Entire detection and diagnostic process should be largely automated with little manual intervention required for creating rules or policies, since the overhead of rule and policy creation makes their use less widespread.

R1 requires that the monitoring and diagnosis engine be designed very carefully so as to keep the CPU, memory and I/O overheads to minimal. R2 requires that the monitoring and diagnosis engine be inherently plug and play which can be automatically downloaded and started on the managed servers. R3 is the most challenging requirement

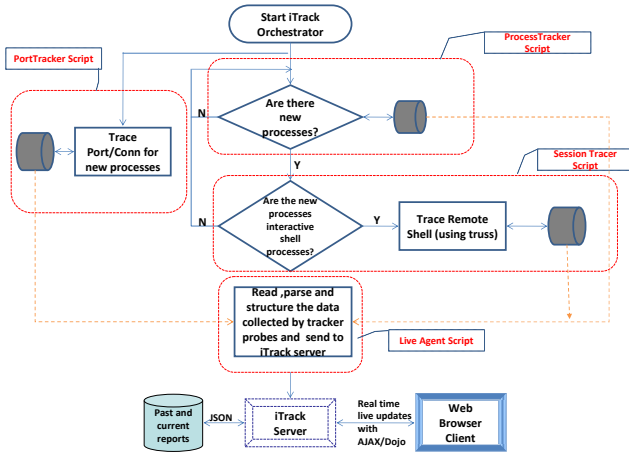

Fig. 2. iTrack Monitoring Control Flow

since it mandates that the monitoring and diagnosis engine can utilize only those utilities that are found on most common Unix* like operating systems. Platform independent options such as Java are not much of a help as they also require a different Java runtime to be installed for each operating system variant and this violates R2. Furthermore, most Java based solutions tend to have a much higher overhead and thus violate R1. R4 necessitates real time diagnosis and reporting of the collected data to a set of servers that can be used to create the necessary visualization.

The above requirements shape our overall methodology and design. R4 mandates that there are a set of centralized iTrack servers for visualization and reporting. R2 and R3 together motivate our design decision to use a scripting framework which makes use of common native monitoring and diagnostic utilities available on all Unix* like operating systems such as "ps","lsof","strace" or "truss". These scripts are pushed to all the managed servers from the iTrack server and can start without requiring any compilation or installation. R1 requires that we keep polling as infrequent as possible, use heavy overhead monitoring only during change management windows and use event notification mechanisms whenever available. R5 mandates that we use information available from available documents such as firewall policy or a Web Service Definition Language (WSDL) document to infer rules.

III. Design and Implementation

Our overall design follows the standard design paradigm of monitoring systems wherein lightweight monitoring agents reside on each managed server and periodically send their aggregated data to a set of centralized iTrack servers which use this data to create real time visualization, reports and alerts for the system administrators (refer Figure 1). Each iTrack server is responsible for a set of managed servers. Our framework assumes that all servers in the data center have ssh capability and it configures password-less ssh for the super-user or system admin user names. This is a one time setup that many system admins implement anyway.

A. iTrack Monitoring Agent

Figure 2 shows the overall control flow of iTrack monitoring. iTrack monitoring agents comprise of perl scripts that are pushed from the iTrack server(s) to the managed servers by secure copy (scp), either at installation time or every time the agent code gets updated. In case of an update, the existing agents are first stopped using the iTrack server. The agents collect their data using commonly found native monitoring utilities such as "ps", "lsof" and "truss". Scripts using "ps" and "lsof" (the **Process Tracker** and the **Port Tracker**, respectively) periodically take a snapshot of current processes and ports and compare this with the last snapshot to find out the new and dead processes as well as opened and closed network ports and socket connections in a particular polling window. Polling interval is configurable and a default value of 5 seconds is used. This means that any transient processes that get forked and killed within a 5 second interval will not be detected by our scripts. This holds true for network ports and connections as well that open and close within the 5 second window.

Every time a new interactive shell process is detected (by the **Session Tracer** script), it is assumed that a potential change window has started and a change can take place on the system. The newly forked interactive shell is traced using native tracing utility such as "strace" (on Linux) or "truss" (on AIX, Solaris). This tracing helps us detect any

Algorithm 1 Trace Decision Algorithm

```
1:  new_processes ← get_new_process_list()
2:  i ← new_processes.length
3:  while (i > 0) do
4:     pid ← getpid(new_processes[i])
5:     pname ← getprocessname(pid)
6:     ppid ← getparentpid(pid)
7:     ppname ← getprocessname(ppid)
8:     pforktime ← getprocessforktime(pid)
9:     if ( (pname eq "ksh") or (pname eq "csh)" or (pname eq "sh") or (pname eq "bash") ) then
10:       if (file_exists("*ppid*.trace") then
11:          pptracetime ← get_creation_time("*ppid*.trace");
12:          if (pptracetime < pforktime) then
13:             continue
14:          end if
15:       end if
16:       take_process_snapshot();
17:       take_port_snapshot();
18:       fork_daemon_and_start_truss_tracing(pid)
19:    end if
20:    i ← i − 1
21: end while
```

child processes that get forked from this shell or files that get modified or deleted. To keep the tracing overhead low, only a few system calls such as fork and execve (to detect new child processes); ulink (to detect file deletions); and open and write (to detect file modifications) are traced. Changes related to networking ports and socket connections are detected using "lsof" output which is correlated with "truss" output. iTrack also allows a user to disable tracing of system calls related to file modifications (write and open) if a lot of file activity is anticipated.

The algorithm used for checking if a newly forked process should be traced is given in Algorithm 1. The if statements in line 10 and 12 ensure that a newly forked process is not traced if its parent is already being traced and the tracing started before the child was forked, since the child will get traced by the truss process tracing its parent. In addition to starting the "truss" tracing of a newly forked interactive shell (a new remote session), this script also takes a "ps" and "lsof" snapshot and sends it to a iTrack server (lines 16 and 17). These two snapshots represent the system state at the start of change window. These are later compared at the server side with the snapshots taken at the end of the change window (denoted by death of an interactive shell process which was being traced). The difference in the process and port state during the change window is compared with the bindings section of a WSDL (or any other file that contains a list of important processes and ports to be monitored) to detect a web service outage.

A daemon process is forked off which in turn forks different child processes to trace each of the selected interactive shells. This is done so that the script can exit after its current iteration, while the truss tracing can continue in the background. The output of each truss command (for each of the selected shells) is written into files that are named according to the following convention:

```
<SessionID>_<TerminalName>_<User>_<PID>_<PPID>_<ClientIP>.trace
```

The **Live Agent** script parses the truss output file(s) generated by the SessionTracer script child process(es). The parsing algorithm is given in algorithm 2 (system calls mentioned here are for AIX, it varies slightly for Linux). The trace data is correlated with "ps" and "lsof" data based on process hierarchy (PID and PPID) and a tree list view of commands executed on a given shell along with processes forked or killed, ports opened or closed, and files deleted or modified is created (refer Figure 3). This data is sent to a iTrack server using HTTP post. Using HTTP ensures that in case the iTrack servers are outside the firewall, the communication can still work.

In the current implementation we have chosen file based input output for simplicity of design and implementation. Even though our experiments demonstrate that the current file based I/O has insignificant performance overhead, we are currently working on replacing the file based input output with named pipes so that inter process communication (between the "truss" output and the LiveAgent script) can be even faster. Figure 2 also shows an overview of the tasks performed by the various monitoring scripts to create the real time dashboard.

Algorithm 2 Trace Parsing Algorithm

Require: Output of "truss" (AIX) or "strace" (Linux) in a file - tracefile
Ensure: a temporary file with a list of the new processes invoked and the command that invoked them as well as file(s) modified/deleted along with the process responsible for modification/deletion

while $((line \leftarrow getline(tracefile)) \neq EOF)$ **do**
  if $(contains(line, \text{“}kfork\text{”}))$ **then**
    $(pid, ppid) \leftarrow extract\_pid\_ppid(line)$
    add_to_process_map(pid,ppid)
  end if
  if $(contains(line, \text{“}exit\text{”}))$ **then**
    $(pid, ppid) \leftarrow extract\_pid\_ppid(line)$
    remove_from_process_map(pid,ppid)
  end if
  if $(contains(line, \text{“}execv\text{”}))$ **then**
    $(pid, ppid) \leftarrow extract\_pid\_ppid(line)$
    $(pname) \leftarrow extract\_process\_name(line)$
    $line \leftarrow get\_next\_line(tracefile)$ {truss puts the command string that triggered the process on the next line}
    $(command\_string) \leftarrow extract\_command\_string(line)$
    write_to_temp_file("COMMAND",pid,ppid,command_name)
  end if
  if $(contains(line, \text{“}kwrite\text{”}))and(error\_code(line)) == 0)$ **then**
    $(pid, ppid) \leftarrow extract\_pid\_ppid(line)$
    $(filename) \leftarrow extract\_file\_name(line)$
    write_to_temp_file("FILEMODIFIED",pid,ppid,file_name)
  end if
  if $(contains(line, \text{“}unlink\text{”}))and(error\_code(line)) == 0)$ **then**
    $(pid, ppid) \leftarrow extract\_pid\_ppid(line)$
    $(filename) \leftarrow extract\_file\_name(line)$
    write_to_temp_file("FILEDELETED",pid,ppid,file_name)
  end if
end while

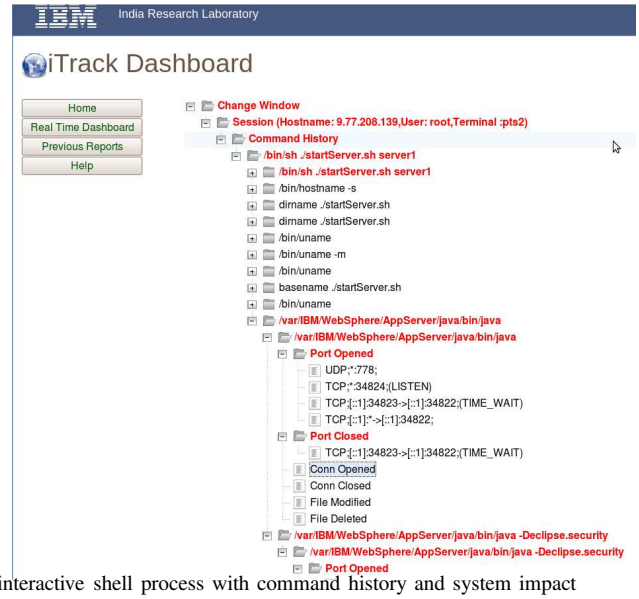

Fig. 3. Real time display of an interactive shell process with command history and system impact

B. iTrack Server

Each iTrack server comprises of a real time dashboard which provides a real time web 2.0 (implemented using DOJO [14]) based visualization of the entire detection and diagnosis process. The real time dashboard gives a window into the managed system and displays all current interactive sessions in real time. It also keeps record of past sessions for up to a week. Each session report, whether current or past, displays each command that was typed on the interactive shell. It also displays the tree view, mentioned above, highlighting the effects of a command on the system - processes that got forked as a result of a particular command, files that got modified or deleted by any of these processes, as well as any ports that were opened or closed. Typically commands will have several subcommands (child processes). However, very few of them have side effects (network ports or sockets opened or

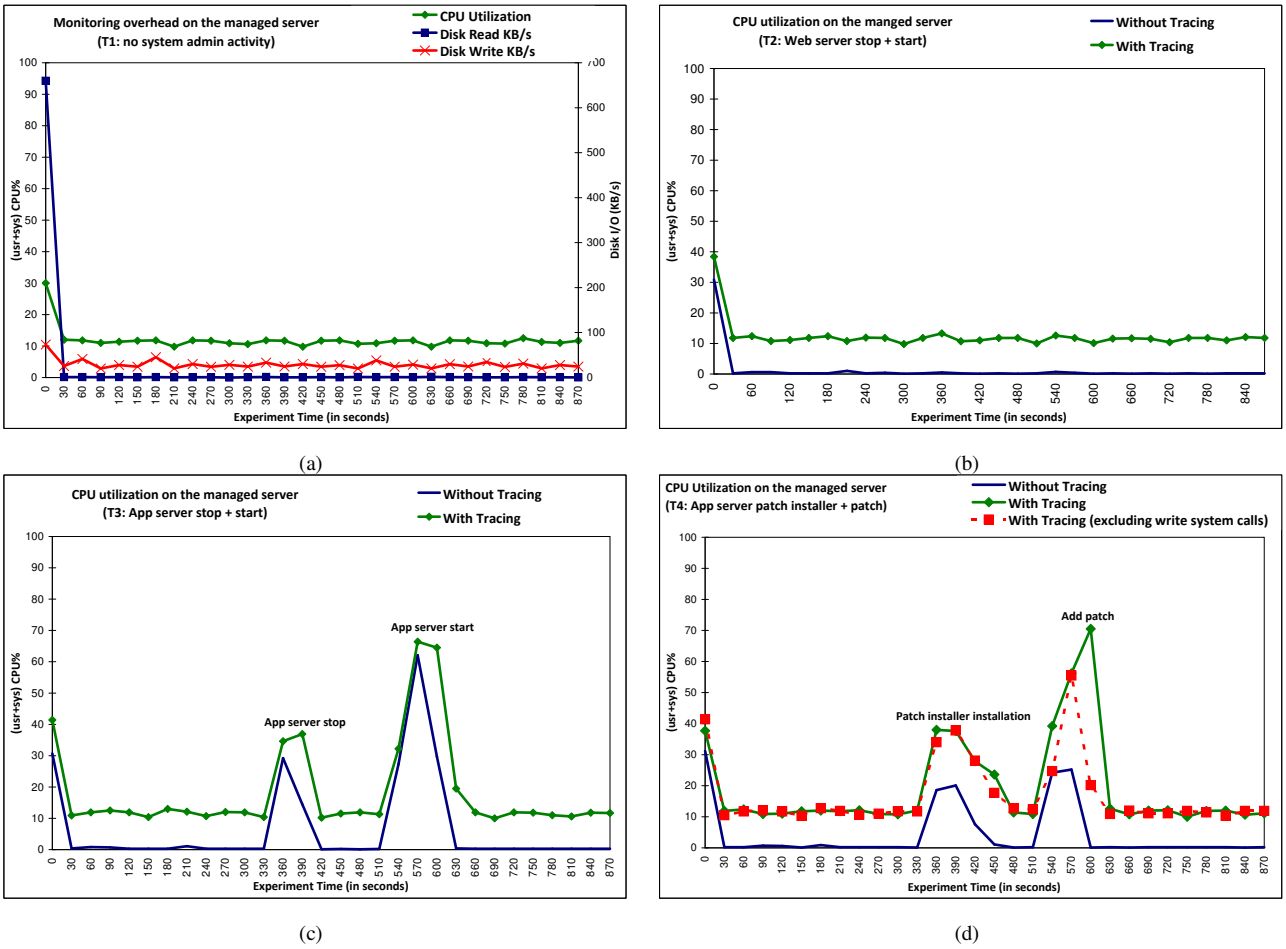

Fig. 4. CPU Utilization at the managed server for various test scenarios

closed, files deleted or modified). Most commands (such as “ls”, “grep”,“cd”, etc) will not have any side effects. Visualization tree highlights all subcommands with side effects in red for easy readability (refer Figure 3). The user interface also allows the user to filter out specific commands (commands with no side effects such as “ls”, “grep”,“cd”, etc) from the reports and the visualization tree.

iTrack server code comprises of a set of Java servlets and JSPs that generate alerts which are displayed in the live session visualization as well sent via Email. These servlets and JSPs together implement the following functions:

- They receive the HTTP POST request from the LiveAgent script and send the relevant data to a user on the browser client if the user is currently observing the real time dashboard. They also simultaneously store this data in a JSON file as a past report for the monitoring session.
- If no user is currently observing the real time dashboard, the server directly writes the POST request data to the JSON file. Other cases such as a user closing his dashboard in the middle of a session or it navigating away from that page, or a browser crash are also handled appropriately.
- They also receive the process and port snapshots from the Session Tracer script at the start and end of a change window (denoted by the fork and death of new remote shell process). These two snapshots are compared and the difference in the process and port state during the change window is compared with the bindings section of a WSDL (or any other file that contains a list of important processes and ports to be monitored) to detect a web service outage. Currently we assume that the WSDLs are locally available. We plan to integrate this with a WSRR or UDDI registry so that WSDLs can be retrieved from them. Once the servlet finds that a particular port mentioned in the WSDL document matches with a port that has closed during the change window, it sends out a potential outage email event notification to the user.

Test Scenario	File data (in KB)
T1 - no system admin activity	19
T2 - web server stop + start	52.4
T3 - app server stop + start	1042
T4 - app server patch installer + patch	23527
T4 - app server patch installer + patch (excluding write system calls)	250.6

TABLE I
File data generated by iTrack monitoring agent

IV. Experimental Evaluation

We now present an experimental evaluation of iTrack. One of the key requirements from section II was to keep the monitoring overhead low. We evaluated the monitoring overhead of iTrack through a series of test scenarios that represent system admin activities commonly performed on production IT systems. In all these scenarios, the polling frequency of iTrack monitoring agent was set to 5 seconds and the test length was 15 minutes. All tests that involved system admin activities, were repeated twice - once with the iTrack monitoring agent running and once without it.

1) **T1: No system admin activity** - In this test, we did not perform any system admin activity and measured the monitoring overhead on the managed server in terms of CPU and disk activity for a period of 15 minutes.
2) **T2: Web server stop/start** - In this test, we simulated a scenario where a system admin would remotely log in to a managed server and restart a running web server. This is a common scenario in production IT systems to resolve slow response time problem either due to memory leaks or other issues.
3) **T3: App server stop/start** - In this test, we simulated a scenario where a system admin would remotely log in to a managed server and restart a running app server. Like T2, this is a common scenario in production IT systems to resolve slow response time problems.
4) **T4: App server install patch installer/add patch** - In this test, we simulated a scenario where a system admin would remotely log in to a managed server and patch an App server. Patching of the App server first requires installation of a patch installer utility followed by the addition of the patch itself.

The CPU utilization graphs for the above tests are given in Figure 4 and the file data generated is given in Table I. In the absence of any system admin activity (T1 above), iTrack monitoring agent periodically invokes "ps" and "lsof" utilities to find new (and dead) processes and ports, respectively. This represents the steady state overhead of iTrack monitoring agent on the managed server. Average CPU utilization (averaged over a 30 second window) in this test remained roughly around 11% (refer Figure 4(a). Most of this overhead (around 8% of it) comes from invocation of "perl" interpreter, and "ps" and "lsof" utilities. The remaining overhead is the correlation and aggregation overhead of iTrack monitoring agent. There was very little disk read activity and the disk write activity remained around 30 KB/s. Total amount of file data that was finally generated at the end of the 15 minute run was 19KB (refer Table I. This means that in absence of any system admin activity, iTrack agent would generate around 1.8 MB data in a day (24 hours) or 54 MB data in a month. Clearly, the file data generated is minimal and does not require much intervention from the system admin to prevent the disk from getting full. Furthermore, all this data is not required at the managed server as the relevant data is sent to the iTrack server periodically and can be deleted thereafter.

Test scenarios T2,T3 and T4 involved remote logins and interactive shells through which system admin activities were performed. These interactive shells were traced by the iTrack monitoring agent using "truss". The output of "truss" is written to a file, which is parsed by the iTrack monitoring agent and relevant data is sent to iTrack server. For test scenarios T2, T3 and T4, in addition to CPU utilization, we also measured the completion time for all system admin activities. These are shown in Figure 5.

In case of T2, the web server stop and start activities are fairly light weight and happen quite quickly. The average CPU utilization (refer Figure 4(b))remained around 11% (which is the overhead due to iTrack monitoring in absence of any system admin activities). The file data generated in this test was 52.4 KB. For T3, CPU utilization graph in Figure 4(c) shows two peaks (of 30% and 60% utilization) that correspond to App server stop and start respectively. The run with tracing enabled follows these two peaks closely and does not introduce much additional overhead during these tasks. The completion time graph shows that the completion time for the run with tracing enabled, increased by around 24% (for App server stop) and 34% (for App server start). The file data generated

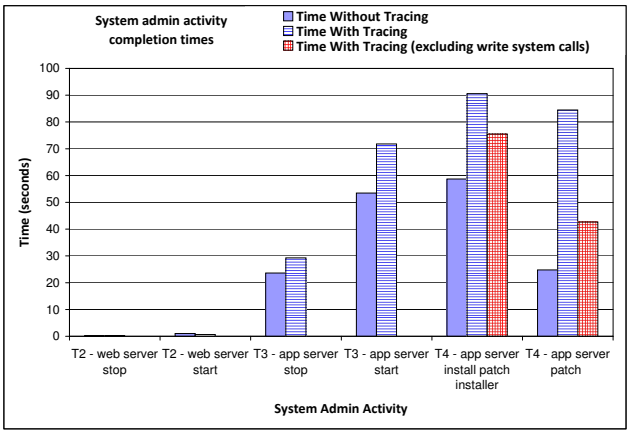

Fig. 5. Comparison of completion times of various system admin activities with and without iTrack tracing

in this test was 1042 KB (nearly 1MB). This is significantly higher than what was generated in T1 and T2 since these activities do a lot of more work and they were being traced by "truss".

In case of T4, when tracing is enabled, the CPU utilization shoots up significantly higher (20% and 45% higher than in absence of any tracing) during the patch installer installation and patch addition activities (refer Figure 4(d)). The completion times for these two activities are also significantly higher (by 54% and 241%, respectively) as compared to completion times without any tracing. Even the file data generated is huge (23527 KB - refer Table I).

A quick look at the tracing data revealed that the two system admin activities in this test scenario (patch installer installation and patch addition) result in a much larger number of write system calls ("kwrite" in AIX). This is expected since any installation and patching activity involves writing new files to the disk. In this particular test, these two activities generated 188,648 write system calls. To put this number in perspective, the total number of write system calls during the App server stop and start activities (T3) was just 1412. Also, the number of fork system calls in this test were twice the number of fork system calls in T3. Clearly, the large number of write and fork system calls that are traced resulted in a much higher CPU overhead. Tracing fork system calls is required for iTrack monitoring in order to track all sub-commands or new processes that may be forked off from a shell. However, tracing of write system calls is only required if the system needs to track the files that are modified from a given shell (or any of its child processes). Since, thousands of files may be modified during an installation or patching activity, this data is not of much use in any case.

We disabled tracing of write system calls and repeated T4. This resulted in significant reduction in completion times (refer Figure 5). Patch installer installation time reduced from 90 seconds to 75 seconds and patch addition time reduced almost by half from 84 seconds to 42 seconds (2 times improvement). The file data generated also got reduced significantly - by almost 94 times (from 23527 KB to 250 KB). However, the CPU overhead remained almost the same during the first activity (installation of patch installer) and reduced significantly (by 15-20%) only during the second activity (patch addition). The users of iTrack can choose to disable tracing of write system calls if they expect heavy installation/patching activity or if they do not really care to track every file system modification done by a system admin to bring the monitoring overhead down.

Performance evaluation of iTrack server:

We measured the resource utilization of iTrack server for the case where a single web user viewed the real time dashboard updates from a single managed server. This gives us a sense of how iTrack server would scale in presence of multiple web users and managed servers.

Figure 6 shows the CPU utilization of the machine hosting iTrack server for the test scenario T4 above (App server patch installer installation and patch addition). Average CPU utilization on the server reaches close to 100% during these system admin activities. The reason for this high CPU utilization at the iTrack server lies in the way it handles incoming data from the monitoring agent. iTrack server gets data from the monitoring agent periodically and writes it to the disk as a single file in JSON format. However, these disk writes are not sequential. This is because the iTrack server assumes that it can get a system event for any command for any session that was started in the past and one which is still executing. With this design, every time the iTrack server gets any HTTP post data from the monitoring agent, it has to read in the corresponding JSON file and search that file to find out the appropriate place (JSON node) in the file where it can append this data (based on session id and process id hierarchy). This is clearly very inefficient and results in the high CPU spikes. We worked on an alternate design

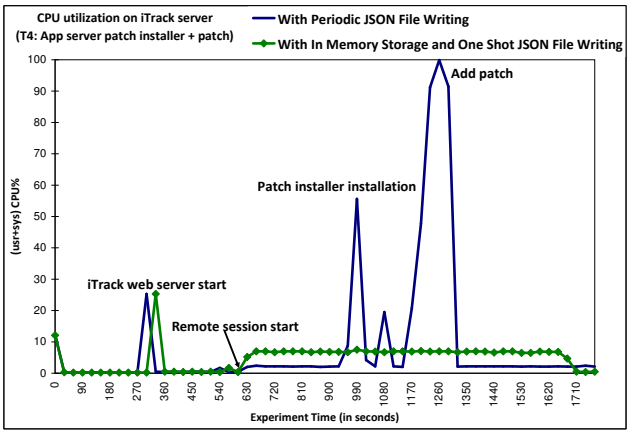

Fig. 6. iTrack server CPU utilization for test scenario T4

where the entire JSON structure for a session is kept in memory and written to the disk only when the session ends. With this design, the high CPU spikes go away and average CPU utilization at the server stays around 7% (refer Figure 6). The memory utilization at the server was around 20MB for keeping this JSON data in-memory. With this CPU and memory utilization, the iTrack server can easily manage 15-20 managed servers with simultaneous system admin activity. In practice, manual system admin activity rarely happens simultaneously on more than 20% of the managed servers. Hence, a single iTrack server can be easily used for tracking system admin activities on 100 managed servers.

The in memory design described above has one drawback - the server hosting iTrack server can run out of memory if the remote session is too long or there are several concurrent remote sessions (on different managed servers) of medium length. To prevent such a scenario, we are currently working on a design in which several small files are created for each subset of data (for example, for a single command). These small sets of data are first kept in memory for some time before they are written to disk. This would reduce the time taken to read in a file related to a system event since the file sizes will be kept small. When a remote session ends, all the files related to that session (corresponding to the different commands that were executed in that session) are merged into a single JSON file, which can be retrieved as a single JSON report by the web user.

V. Related Research

There has been quite a lot of work on preventing human errors in systems management. It can be broadly categorized as follows:

- management systems that automate certain tasks [15], [16]
- management systems that guide operator tasks that need to be done manually [17], [18]
- management systems that audit changes to the managed system's persistent state [19]
- management systems that isolate and validate operator actions before they become visible to the users or to the rest of the managed system [2], [20]

The most recent work from Oliveira et. al [5] proposes a "mistake-aware" management system called "Barricade" which continuously monitors operator commands and system state to decide whether it should react to what the operator is doing. The overall system provides a capability to take the appropriate nodes off-line, recognize the task being performed, and block that task from being carried out on all replicas if the expected cost of mistakes is high. The operator actions are allowed to disseminate to the other nodes after they have been tested and verified to be correct by the management system. The iTrack framework presented in this paper compliments the human error prevention systems such as barricade in cases when there is not enough test data to predict an erroneous action.

Another related area of work is around recording of user sessions for server [11], [12] and personal desktop systems [21]. While these tools give a very accurate record of what happened on the system, they are not very helpful in detecting system outages for two reasons. First, they do not record system level events that occur as a result of user activities. These system level events provide vital clues to outages and in most cases they also represent early symptoms of an outage. Second, these systems tend to produce too much data per session and this makes it difficult to use them in a data center environment with thousands of managed servers. iTrack correlates user activities with monitored system data to provide an end-to-end diagnosis engine.

Performance problem determination through statistical analysis of performance or monitoring data from middleware systems [22], [23] or configuration data from network elements [24], [25] (typically referred to as "anomaly detection") is an orthogonal area of related research with rich history. Typically these systems rely on statistical rule mining and changes in configuration or monitoring data to detect any behavior changes in network [24], [25] or server [22], [23] performance. We view this area of work as one possible area of future research for us. Outage detection in iTrack is currently largely based on explicit rules such as those found in firewall policies or WSDL documents. However, this might not work that well for outages that occur as a result of configuration file changes. Statistical rule mining techniques can be applied to the data generated by iTrack monitoring agents to learn associations between user activities and their associated side effects and deviations away from the "normal" associations can be flagged as a potential outage scenario.

VI. Conclusion and Future Work

In this paper, we described the design and implementation of iTrack - a framework for monitoring user activities and correlating them with system data for fast detection and diagnosis of service outages. We formulated key requirements for a monitoring and diagnosis engine based on our interactions with various data center system administrators. We leveraged the various standard diagnostic utilities provided by Unix like commodity operating systems and created a unifying framework that meets our requirements and aids the overall diagnosis of an outage. iTrack monitors systems events as well as system admin activity, correlates these two sets of information and categorizes the commands as potentially abnormal or harmful based on their impact on the system in terms of file system, network and memory activities or based on their parameter values. Our results confirmed that iTrack overhead in terms of CPU time, activity completion time and data generated is within the tolerance range of most production systems. In cases, where the overhead was found to be unacceptable, we detect the underlying cause and provide solutions which improve performance by up to 20% (in case of managed sever CPU utilization) and by up to 90% (in case of iTrack server CPU utilization) or by up to 2 times (in terms of completion time of certain system admin activities on the managed server).

We are currently working on a number of enhancements that aim to further reduce the performance overheads of iTrack as well as make it more automatic and capable of detecting a wider variety of outages. We are working on replacing the file based input output at the agents with named pipes so that inter process communication (between the "truss" output and the LiveAgent script) can be even faster. On the server side, we are currently working on integrating the server with a WSRR or UDDI registry so that WSDLs can be automatically retrieved from them and compared with the iTrack agent data. We are also working on replacing the current in-memory storage of a system admin remote session at the iTrack server with a hybrid model where each session is split into smaller sub-sessions and data for each sub-session is first stored in memory and periodically synced to disk. Finally, we are also exploring if statistical rule mining can be employed to learn rules about various commands and their associated impact on the system using the data generated by iTrack agents. Once rules have been learnt, alerts can be generated when the observed behavior deviates significantly from these rules. Such deviations may be early signs of impending outages.

References

[1] J. Gray, "Why do Computers Stop and What Can Be Done About It?" in *IEEE SRDS*, January 1986.

[2] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and Validating Database System Administration," in *USENIX Annual Technical Conference*, 2006.

[3] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems," in *NSDI*, 2006.

[4] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do Internet Services Fail, and What Can Be Done About It," in *USENIX Symposium on Internet Technologies and Systems*, 2003.

[5] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Barricade: Defending Systems Against Operator Mistakes," in *Eurosys*, 2010.

[6] D. Scott, "Operation zero downtime," *Gartner Security Conference presentation*, May 2002.

[7] "Juniper, What is behind network downtime?" 2008.

[8] Z. Kerravala, "As the value of enterprise networks escalates, so does the need for configuration management," *The Yankee Group*, Jan 2004.

[9] "2007 Aperture Research Institute survey," http://royal.pingdom.com/2007/10/30/human-errors-most-common-reason-for-data-center-outages/.

[10] "Linux Audit," http://doc.opensuse.org/products/draft/SLES/SLES-security_draft/cha.audit.comp.html.

[11] "Script command linux man page," http://linux.die.net/man/1/script.

[12] "ObserveIT," http://www.observeit-sys.com/.

[13] "Tripwire," http://www.tripwire.com/.

[14] "DOJO," http://dojotoolkit.org/.
[15] Y. Y. Su, M. Attariyan, and J. Flinn, "AutoBash: Improving Configuration Management with Operating System Causality Analysis," in *SOSP*, 2007.
[16] W. Zheng, R. Bianchini, and T. Nguyen, "Automatic Configuration of Internet Services," in *Eurosys*, 2007.
[17] P. Bodik, A. Fox, M. I. Jordan, D. Patterson, A. Banerjee, R. Jagannathan, T. Su, S. Tenginakai, B. Turner, and J. Ingalls, "Advanced Tools for Operators at Amazon.com," in *1st Workshopon Hot Topics in Autonomic Computing*, 2006.
[18] V. R. Madduri, M. Gupta, P. De, and V. Anand, "Towards Mitigating Human Errors in IT Change Management Process," in *ICSOC*, 2010.
[19] C. Verbowski, J. Lee, X. Liu, R. Roussev, and Y. M. Wang, "LiveOps: Systems Management as a Service," in *20th Large Installation System Administration Conference*, 2006.
[20] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. Nguyen, "Understanding and Dealing with Operator Mistakes in Internet Services," in *OSDI*, 2004.
[21] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh, "DejaView: A Personal Virtual Computer Recorder," in *SOSP*, 2007.
[22] V. Mann, M. K. Agarwal, M. Gupta, and N. Sachindran, "Problem Determination in Enterprise Middleware Systems Using Change Point Correlation of Time Series Data," in *IEEE/IFIP NOMS*, 2006.
[23] M. K. Agarwal, N. Sachindran, M. Gupta, and V. Mann, "Fast Extraction of Adaptive Change Point Based Patterns for Problem Resolution in Enterprise Systems," in *IFIP/IEEE DSOM*, 2006.
[24] K. El-Arini and K. S. Killourhy, "Bayesian detection of router configuration anomalies," in *ACM SIGCOMM Workshop on Mining Network Data (MineNet)*, 2005.
[25] A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons, "Detecting the Performance Impact of Upgrades in Large Operational Networks," in *ACM SIGCOMM*, 2010.