Red Hat Enterprise Linux 5.3

SystemTap Beginners Guide

For use with Red Hat Enterprise Linux 5

Don Domingo

Red Hat Enterprise Linux 5.3 SystemTap Beginners Guide
For use with Red Hat Enterprise Linux 5
Edition 1.0

Author Don Domingo ddomingo@redhat.com

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072USAPhone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588Research Triangle Park, NC 27709USA

This guide provides basic instructions on how to use SystemTap to monitor different subsystems of Red_Hat_Enterprise_Linux 5 in finer detail. The SystemTap Beginners Guide is recommended for users who have taken RHCT or have a similar level of expertise in Red_Hat_Enterprise_Linux 5.

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.
>
> A useful shortcut for the above command (and many others) is **`Tab`** completion. Type **`cat my_`** and then press the **`Tab`** key. Assuming there are no other files in the current directory which begin with 'my_', the rest of the file name will be entered on the command line for you.
>
> (If other file names begin with 'my_', pressing the **`Tab`** key expands the file name to the point the names differ. Press **`Tab`** again to see all the files that match. Type enough of the file name you want to include on the command line to distinguish the file you want from the others and press **`Tab`** again.)

The above includes a file name, a shell command and two key caps, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**-**`Alt`**-**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**-**`Alt`**-**`F7`** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`Mono-spaced Bold`**. For example:

[1] https://fedorahosted.org/liberation-fonts/

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }

}
```

1.3. Notes and Warnings

Finally, we use three distinct visual styles to highlight certain information nuggets.

Note

A note is useful bit of information: a tip or shortcut or an alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

The Important information box highlights details that are easily missed: such as configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring important information won't cause data loss but may cause irritation and frustration.

Warning

A Warning highlights vital information that must not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/bugzilla/* against the product **Red_Hat_Enterprise_Linux 5.**

When submitting a bug report, be sure to mention the manual's identifier: *SystemTap_Beginners_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction

SystemTap is a tracing and probing tool that provides users to study and monitor the activities of the operating system (particularly, the kernel) in fine detail. It provides information similar to the output of tools like **`netstat`**, **`ps`**, **`top`**, and **`iostat`**; however, SystemTap is designed to provide information that is more "granular" in nature.

For system administrators, SystemTap can be used as a performance monitoring tool for . It is most useful when other similar tools cannot precisely pinpoint a bottleneck in the system, requiring a deep analysis of kernel activity. In the same manner, application developers can also use SystemTap to monitor, in finer detail, how their application behaves.

1.1. Goals

SystemTap provides the infrastructure to monitor the running Linux kernel for detailed analysis. This can assist in identifying the underlying cause of a performance or functional problem.

Without SystemTap, monitoring the activity of a running kernel would require a tedious instrument, recompile, install, and reboot sequence. SystemTap is designed to eliminate this, allowing users to gather the same information by simply running its suite of tools against specific *tapsets* or SystemTap scripts.

However, SystemTap was initially designed for users with intermediate to advanced knowledge of the kernel. As such, much of the existing documentation for SystemTap is primarily for advanced users. This could present a steep learning curve for administrators or developers whose knowledge of the Linux kernel is little to none.

In line with that, the main goals of the *SystemTap Beginner's Guide* are as follows:

- To introduce users to SystemTap, familiarize them with its architecture, and provide setup instructions for all kernel types.
- To provide pre-written SystemTap scripts for monitoring and forensic tasks, along with instructions on how to analyze their output.

1.2. SystemTap Versus Other Monitoring Tools

Advantages

TBD

Limitations

TBD

Understanding How SystemTap Works

SystemTap allows users to write and reuse simple scripts to deeply examine the activities of a running Linux system. These scripts can be designed to extract data, filter it, and summarize it quickly (and safely), enabling the diagnosis of complex performance (or even functional) problems.

The essential idea behind a SystemTap script is to name *events*, and to give them *handlers*. When SystemTap runs the script, SystemTap monitors for the event; once the event occurs, the Linux kernel then runs the handler as a quick sub-routine, then resumes.

There are several kind of events; entering/exiting a function, timer expiration, session termination, etc. A handler is a series of script language statements that specify the work to be done whenever the event occurs. This work normally includes extracting data from the event context, storing them into internal variables, or printing results.

2.1. Architecture

A SystemTap session begins when you run a SystemTap script. This session occurs in the following fashion:

Procedure 2.1. SystemTap Session

1. SystemTap first translates the script to C, running the system C compiler to create a kernel module from it.
2. SystemTap loads the module, then enables all the probed events by "hooking" those events into the kernel.
3. As the events occur, their corresponding handlers are executed.
4. Once the SystemTap session is terminated, the hooked events are disconnected from the kernel; afterwards, the kernel module is unloaded.

This sequence is driven from a single command-line program: **`stap`**. This program is SystemTap's main front-end tool. For more information about **`stap`**, refer to **`man stap`** (once SystemTap is set up on your machine).

2.2. SystemTap Scripts

For the most part, SystemTap scripts are the foundation of each SystemTap session. SystemTap scripts instruct SystemTap on what type of information to trap, and what to do once that information is trapped.

As stated in *Chapter 2, Understanding How SystemTap Works*, SystemTap scripts are made up of two components: *events* and *handlers*. Once a SystemTap session is underway, SystemTap monitors the operating system for the specified events and executes the handlers as they occur.

Note

An event and its corresponding handler is collectively called a *probe*. A SystemTap script can have multiple probes.

A probe's handler is also commonly referred to as a *probe body*.

In terms of application development, using events and handlers is similar to inserting **`print`** statements in a program's sequence of commands. These **`print`** statements allow you to view a history of commands executed once the program is run.

SystemTap scripts go one step further by allowing you more flexibility with regard to handlers. Events serve as the triggers for handlers to run; handlers can be specified to trap specified data and print it in a certain manner.

Format

SystemTap scripts use the file extension **`.stp`**, and are written in the following format:

```
probe [event],[another event]{
 [handler] exit()
}
```

The *`exit()`* condition is optional, but it is recommended since it safely terminates the session once the script successfully traps the required information.

Important

Section 2.2, "SystemTap Scripts" is designed to introduce readers to the basics of SystemTap scripts. To understand SystemTap scripts better, it is advisable that you refer to *Chapter 4, Useful SystemTap Scripts*; each section therein provides a detailed explanation of the script, its events, handlers, and expected output.

2.2.1. Events

SystemTap events can be broadly classified into two types: *synchronous* and *asynchronous*.

Synchronous Events

A *synchronous* event occurs when any processor executes an instruction matched by the specification. This gives other events a reference point (or instruction address) from which more contextual data may be available.

Examples of synchronous events include:

kernel.function("*`[function]`*")
: The entry to the kernel function *`function`*. For example, **`kernel.function("sys_open")`** refers to the "event" that the kernel function **`sys_open`** is used. To specify the *return* of the kernel function **`sys_open`**, append the **`return`** string to the event statement; i.e. **`kernel.function("sys_open").return`**.

  When defining functions, you can use asterisk (**`*`**) for wildcards. You can also trace the entry/exit of a function in a kernel source file. Consider the following example:

```
probe kernel.function("*@net/socket.c") { }
probe kernel.function("*@net/socket.c").return { }
```

Example 2.1. Wildcards and Kernel Source Files in an Event

In the previous example, the first probe's event specifies the entry of ALL functions in the kernel source file **net/socket.c**. The second probe specifies the exit of all those functions. Note that in this example, no handler was specified; as such, no information will be displayed.

syscall.*[system_call]*
: The entry to the system call *[system_call]*. Similar to **kernel.function**, appending a **return** to the statement specifies the exit of the system call. For example, to specify the entry of the system call **close**, use **syscall.close.return**.

To identify what system calls are made by a specific program/command, use **strace *command***.

module("*[module]*").function("*[function]*")
: Allows you to probe functions within modules. For example:

```
probe module("ext3").function("*") { }
probe module("ext3").function("*").return { }
```

Example 2.2. Module Probe

The first probe in *Example 2.2, "Module Probe"* points to the entry of *all* functions for the **ext3** module. The second probe points to the exits of all entries for that same module; the use of the **.return** suffix is similar to **kernel.function()**. Note that the probes in *Example 2.2, "Module Probe"* also do not contain probe bodies, and as such will not print any useful data (as in *Example 2.1, "Wildcards and Kernel Source Files in an Event"*).

A system's loaded modules are typically located in **/lib/modules/*[kernel version]***, where *kernel version* refers to the currently loaded kernel. Modules use the filename extension **.ko**.

Asynchronous Events

Asynchronous events, on the other hand, do not point to any reference point. This family of probe points consists mainly of counters, timers, and similar constructs.

Examples of asynchronous events include:

begin
: The startup of a SystemTap session; i.e. as soon as the SystemTap script is run.

end
: The end of a SystemTap session.

timer.ms()
: An event that specifies a handler to be executed "after X number of milliseconds". For example:

```
probe timer.ms(4000)
{
 exit()
}
```

Example 2.3. Using timer.ms

Example 2.3, "Using timer.ms" is an example of a probe that allows you to terminate the script after 4000 milliseconds (or 4 seconds). When used in conjunction with another probe that traps a large quantity of data, a probe using **`timer.ms()`** allows you to limit the information your script is collecting (and printing out).

Important

SystemTap supports the use of a large collection of probe events. For more information about supported events, refer to **`man stapprobes`**. The *SEE ALSO* section of **`man stapprobes`** also contains links to other **`man`** pages that discuss supported events for specific subsystems and components.

SystemTap supports multiple events per probe; as shown in *Format*, multiple events are delimited by a comma (**`,`**). If multiple events are specified in a single probe, SystemTap will execute the handler when any of the specified events occur.

2.2.2. Handlers/Probe Body

Consider the following sample script:

```
probe begin
{
  printf ("hello world\n")
  exit ()
}
```

Example 2.4. Hello World

In *Example 2.4, "Hello World"*, the event **`begin`** (i.e. the start of the session) triggers the handler enclosed in **`{ }`**, which simply prints **`hello world`**, then exits.

printf ( ) Statements

The **`printf ()`** statement is one of the simplest handler tools for printing data. **`printf ()`** can also be used to trap data using a wide variety of SystemTap handler functions using the following format:

```
printf ("[format string]\n", [argument])
```

The *`[format string]`* region specifies how *`[argument]`* should be displayed. The format string of *Example 2.4, "Hello World"* simply instructs SystemTap to print **`hello world`**, and contains no arguments.

You can use the variables **%s** (for strings) and **%d** (for numbers) in format strings, depending on your list of arguments. Format strings can have multiple variables, each matching a corresponding argument; multiple arguments are delimited by a comma (**,**) and space.

To illustrate this, consider the following probe example:

```
# This probe will need to be manually terminated with Ctrl-C
probe syscall.open
{
  printf ("%s(%d) open\n", execname(), pid())
}
```

Example 2.5. Using Variables In printf ( ) Statements

Example 2.5, "Using Variables In printf ( ) Statements" instructs SystemTap to probe all entries to the system call **open**; for each event, it prints the current **execname()** (which is a string) and **pid()** (which is a number), followed by the word **open**. A snippet of this probe's output would look like:

```
vmware-guestd(2206) open
hald(2360) open
hald(2360) open
hald(2360) open
df(3433) open
df(3433) open
df(3433) open
hald(2360) open
```

Handler Functions

SystemTap supports a wide variety of handler functions that can be used as **printf ()** arguments. *Example 2.5, "Using Variables In printf ( ) Statements"* uses the handler functions **execname()** (current process name) and **pid()** (current process ID).

The following is a list of commonly-used handler functions:

tid()
: The ID of the current thread.

uid()
: The ID of the current user.

cpu()
: The current CPU number.

gettimeofday_s()
: The number of seconds since UNIX epoch (January 1, 1970).

get_cycles()
: A snapshot of the hardware cycle counter.

pp()
: A string describing the probe point currently being handled.

probefunc()
: If known, the name of the function in which the probe was placed.

thread_indent()
: This particular handler function is quite useful, providing you with a way to better organize your print results. When used with an indentation parameter (for example, **-1**), it allows the probe to internally store an "indentation counter" for each thread (identified by ID, as in **tid**). It then returns a string with some generic trace data along with an appropriate number of indentation spaces.

  The generic data included in the returned string includes a timestamp (number of microseconds since the most recent initial indentation), a process name, and the thread ID. This allows you to identify what functions were called, who called them, and the duration of each function call.

  Consider the following example on the use of **thread_indent()**:

```
probe kernel.function("*@net/socket.c")
{
 printf ("%s -> %s\n", thread_indent(1), probefunc())
}
probe kernel.function("*@net/socket.c").return
{
 printf ("%s <- %s\n", thread_indent(-1), probefunc())
}
```

Example 2.6. Using thread_indent()

Example 2.6, "Using thread_indent()" prints out the **thread_indent()** and probe functions at each event in the following format:

```
     0 ftp(7223): -> sys_socketcall
  1159 ftp(7223): -> sys_socket
  2173 ftp(7223):   -> __sock_create
  2286 ftp(7223):    -> sock_alloc_inode
  2737 ftp(7223):    <- sock_alloc_inode
  3349 ftp(7223):    -> sock_alloc
  3389 ftp(7223):    <- sock_alloc
  3417 ftp(7223):   <- __sock_create
  4117 ftp(7223):   -> sock_create
  4160 ftp(7223):   <- sock_create
  4301 ftp(7223):   -> sock_map_fd
  4644 ftp(7223):    -> sock_map_file
  4699 ftp(7223):    <- sock_map_file
  4715 ftp(7223):   <- sock_map_fd
  4732 ftp(7223): <- sys_socket
  4775 ftp(7223): <- sys_socketcall
```

For more information about supported handler functions, refer to **man stapfuncs**.

2.3. Tapsets

Tapsets are scripts that form a library of pre-written probes and functions to be used in SystemTap scripts. When a user runs a SystemTap script, SystemTap checks the script's probe events and

handlers against the tapset library; SystemTap then loads the corresponding probes and functions before translating the script to C (refer to *Section 2.1, “Architecture”* for information on what transpires in a SystemTap session).

Like SystemTap scripts, tapsets use the filename extension **`.stp`**. The standard library of tapsets is located in **`/usr/share/systemtap/tapset/`** by default. However, unlike SystemTap scripts, tapsets are not meant for direct execution; rather, they constitute the library from which other scripts can pull definitions.

Simply put, the tapset library is an abstraction layer designed to make it easier for users to define events and functions. In a manner of speaking, tapsets provide useful "aliases" for functions that users may want to specify as an event; knowing the proper alias to use is, for the most part, easier than understanding how to specify a specific kernel function.

Several handlers and functions in *Section 2.2.1, “Events”* and *Section 2.2.2, “Handlers/Probe Body”* are defined in tapsets. For example, **`thread_indent()`** is defined in **`indent.stp`**.

Using SystemTap

This chapter instructs users how to install SystemTap, and provides an introduction on how to run SystemTap scripts.

3.1. Setup and Installation

To deploy SystemTap, you need to install the SystemTap packages along with the corresponding set of debug RPMs of your kernel. This means that if your system has multiple kernels installed, and you wish to use SystemTap on more than one kernel, you will need to install the debug RPMs for *each* of those kernels.

Preparing For Installation

To view what kernels and kernel versions are installed on your system, check the contents of **/boot**. Each installed kernel/kernel version has a corresponding **vmlinuz-*[kernel version]*** there.

To determine what kernel your system is currently using, use:

```
uname -r
```

Procedure 3.1. Deploying SystemTap

1. Once you've decided which kernels you need to use SystemTap with, install the following packages:

   - **systemtap**
   - **systemtap-runtime**

   This will install the SystemTap suite of tools.

2. Next, you'll need to download and install the necessary debug RPMs for your kernel. Most debugging RPMs for Red Hat Enterprise Linux 5 can be found at the following link:

   The necessary debugging RPMs are as follows:

   - **kernel-debuginfo**
   - **kernel-debuginfo-common**
   - **kernel-devel**

   For example, if you wish to use SystemTap on kernel version **2.6.18-53.el5**, then you need to download the following debugging RPMs:

   - **kernel-debuginfo-2.6.18-53.1.13.el5.i686.rpm**
   - **kernel-debuginfo-common-2.6.18-53.1.13.el5.i686.rpm**
   - **kernel-devel-2.6.18-53.1.13.el5.i686.rpm**

   Example 3.1. Sample List of Debugging RPMs

3. Install the debugging RPMs using **`rpm -Ivh [RPM]`** or **`yum localinstall [RPM]`**.

Cross-Compiling

TBD

3.2. Usage

Useful SystemTap Scripts

Understanding SystemTap Errors

Tips and Tricks

References

Appendix A. Revision History

Revision History

Revision 1.0 September 2, 2008 DonDomingo*ddomingo@redhat.com*

Built scratch build (pre-Alpha) of document, content to be added later.