libstapsdt Documentation

Release 0.1.1

Mary Marchini

Oct 16, 2021

Contents

libstapsdt is a library which allows creating and firing Systemtap's USDT probes at runtime. It's inspired on chrisa/libusdt. The goal of this library is to add USDT probes functionality to dynamic languages.

CHAPTER 1

Motivation

Systemtap's USDT implementation allows only statically defined probes because they are set as ELF notes by the compiler. To create probes at runtime, *libstapsdt* takes advantage of shared libraries: it creates a small library with an ELF note and links it at runtime. This way, most existing tools will keep working as expected.

CHAPTER 2

Table of Contents

2.1 Getting Started

`libstapsdt`

2.1.1 Packages

We currently offer package installation for Ubuntu via PPA.

Ubuntu via PPA

To install `libstapsdt` on Ubuntu, you need to add Sthima's Open Source Software PPA first, and then you'll be able to install it:

```
sudo add-apt-repository ppa:sthima/oss
sudo apt-get update
sudo apt-get install libstapsdt0 libstapsdt-dev
```

2.1.2 Build from source

If you need to install libstapsdt in a different distribution or want to use the latest version, you can build it from source as follow.

Dependencies

First, you need to install `libstapsdt` dependencies. For now, the only dependency is libelf (from elfutils).

Ubuntu

```
sudo apt install libelf1 libelf-dev
```

Build

To build and install libstapsdt, you just need to run:

```
make
sudo make install
sudo ldconfig
```

Now `libstapsdt` is installed on your system!

Demo

There's a demo program available. To build it, run:

```
make demo  # Executable will be available at ./demo
```

You can then try it by running:

```
./demo PROVIDER_NAME PROBE_NAME
```

After running the demo program, you can.

Here's an example using the latest version of iovisor/bcc's `trace` tool:

```
sudo /usr/share/bcc/tools/trace -p $(pgrep demo) 'u::PROBE_NAME'
```

2.2 How it works

2.2.1 libstapsdt explained

`libstapsdt` has a small API with only 7 functions and one possible flow. To use `libstapsdt` you need to create a provider and register probes to it. A provider has two possible states: unloaded and loaded. After creating it, it's state is unloaded. After all probes are registered, you need to load your provider, effectively allowing tracing tools to access the provider's probes.

After you finish using those probes, you can unload the provider, freeing all structures allocated to it during the load step. Only after that you can destroy the provider, removing all registered probes with it.

This flow is illustrated in the image below:

Example

The following example goes through all steps of `libstapsdt`'s flow:

```
#include <unistd.h>
#include <stdio.h>
#include <libstapsdt.h>

int main() {
  int i=1;
  SDTProvider_t *provider;
  SDTProbe_t *probe;

  provider = providerInit("myLittleProvider");
  probe = providerAddProbe(provider, "myLittleProbe", 1, uint64);

  if(providerLoad(provider) == -1) {
    printf("Something went wrong...\n");
    return -1;
  }

  while(i) {
    printf("Firing probe...\n");
    if(probeIsEnabled(probe)) {
      probeFire(probe, "I'm a runner!");
      printf("Probe fired!\n");
      i = 0;
    }
    sleep(1);
  }

  providerUnload(provider);
  providerDestroy(provider);

  return 0;
}
```

Let's take a better look at the code. The first lines are used to create a provider named `"myLittleProvider"` (using *`providerInit()`*) and then it registers a probe named `"myLittleProbe"` (using *`providerAddProbe()`*). You may have noticed those extra arguments to *`providerAddProbe()`*: they are used to determine how many arguments the probe will accept when fired, and the type of those parameters (you can see all available types in *`SDTArgTypes_t`*).

```
SDTProvider_t *provider;
SDTProbe_t *probe;

provider = providerInit("myLittleProvider");
probe = providerAddProbe(provider, "myLittleProbe", 1, uint64);
```

After creating our provider and registering our probe, we need to load our provider (otherwise we won't be able to fire our probes). This is done by simply calling *`providerLoad()`* with the provider as argument. It's important to handle any errors that may occur to avoid problems in execution later on.

```
if(providerLoad(provider) == -1) {
  printf("Something went wrong...\n");
  return -1;
}
```

Now we can use *`probeIsEnabled()`* and *`probeFire()`*. *`probeIsEnabled()`* will only return *True* if the program is being traced. Therefore, in this example we'll be on an infinite loop until our program is traced. You can use iovisor/bcc `trace` tool for this (`sudo /usr/share/bcc/tools/trace -p PID`

'u::myLittleProbe').

After using the trace tool, our probe will be replaced by a breakpoint, and *probeIsEnabled()* will return True, firing the probe with *probeFire()* inside our if-statement and then stepping out of our loop.

```
while(i) {
  printf("Firing probe...\n");
  if(probeIsEnabled(probe)) {
    probeFire(probe, "I'm a runner!");
    printf("Probe fired!\n");
    i = 0;
  }
  sleep(1);
}
```

Those last lines of code are used to unload and cleanup our provider. It is imporant that you run both *providerUnload()* and *providerDestroy()* in this exact order after you don't need the probes anymore, to avoid memory leaks and Segmentation faults.

```
providerUnload(provider);
providerDestroy(provider);
```

2.2.2 Behind the Scene

Attention: This page is intended for advanced users, and it assumes working knowledge of **Elf** files, **shared libraries** and **software instrumentation**.

libstapsdt uses Systemtap SDT format to create runtime SDT probes - therefore the reason for its name. So why write yet another library for SDT probes instead of using Systemtap?

How Systemtap SDT works

Systemtap uses compiler macros to register its SDT probes, making it impossible to have probes registered during runtime. An example is shown below, where we register a probe called *Probe* to a provider called *Provider*.

```
#include <sys/sdt.h>

int main() {
  DTRACE_PROBE(Provider, Probe);
  return 0;
}
```

The resulting binary from this code will have a new Elf section called .stapsdt.base, located right after the code (usually being the .text section). This base is relevant to help tracing tools to calculate the memory address of any probe after the binary is loaded into memory.

It will also have a Elf note, where all probes data (name, address, semaphores, arguments) will be stored to be read later by any tracing tool. The compiler will also replace our DTRACE_PROBE macro with a function call, and that's where the probe points to, allowing it to easily pass arguments to the probe. This function is a no-op.

```
Displaying notes found at file offset 0x00001064 with length 0x0000003c:
  Owner                 Data size       Description
  stapsdt               0x00000028      NT_STAPSDT (SystemTap probe descriptors)
```

(continues on next page)

(continued from previous page)

```
    Provider: Provider
    Name: Probe
    Location: 0x00000000004004da, Base: 0x0000000000400574, Semaphore:␣
↪0x0000000000000000
    Arguments:
```

There's more information about how Systemtap implements their SDT probes here.

Roses are Red, Violets are Blue...

And shared libraries are Elf files! (on most UNIX systems at least)

Ok, so now we know that Systemtap uses Elf properties to inform tracing tools about registered probes. We also know that they have a well-defined and rather simple strucutre. One which can easily be implemented.

But we can't edit our binary just to add new Elf notes pointing to new probes, and most Systemtap-SDT-capable tracing tools will only look at the binary and not at the running process for this information.

That means we need to generate an Elf file at runtime, add our probes to it, and then use it in our running process in a way that our tracing tools will find it, which means... Shared libraries!

The "secret source" used by `libstapsdt` to allow Systemtap SDT probes registration at runtime is shared libraries. Our public API is rather simple, but the library has quite some code. Most of this code is used to generate a shared library from scratch, dynamically adding code to it and registering all probes as Elf notes.

The shared library is created (with help from `libelf`) and loaded into memory (by using `dlopen()` when *`providerLoad()`* is executed. That's why it's not possible to add new probes after a provider is loaded. It's also worth noting that each provider will generate exactly one shared library when loaded, and providers don't share a shared library.

2.3 Wrappers

Here is a list of wrappers for other languages:

- Python
- NodeJS
- Go

2.3.1 Write your own wrapper

`libstapsdt` is written in C, which makes it very portable to almost any language. Most dynamic languages provide a way to wrap C code. Feel free to develop a wrapper in your language of choice. If you do, please let us know to update our wrappers list!

2.4 API

2.4.1 Types

struct **SDTProvider_t**
Represents a USDT Provider. A USDT Provider is basically a namespace for USDT probes. Shouldn't be created manually, use *providerInit()* instead.

char ***name**
Provider's name

SDTProbeList_t ***probes**
Linked-list of registered probes

SDTError_t **errno**
Error code of the last error for this provider

char ***error**
Error string of the last error for this provider

struct **SDTProbe_t**
Represents a USDT Probe. A probe is basically a software breakpoint. Shouldn't be created manually, use *providerAddProbe()* instead.

char ***name**
Probe's name

ArgType_t argFmt[MAX_ARGUMENTS]
Array holding all arguments accepted by this probe.

struct SDTProvider ***provider**
Pointer to this probe's provider

int **argCount**
Number of accepted arguments

enum **SDTArgTypes_t**
Represents all accepted arguments defined by Systeptap's SDT probes.

noarg
No argument

uint8
8 bits unsigned int

int8
8 bits signed int

uint16
16 bits unsigned int

int16
16 bits signed int

uint32
32 bits unsigned int

int32
32 bits signed int

uint64
64 bits unsigned int

int64
64 bits signed int

enum **SDTError_t**
Represents all errors thrown by libstapsdt.

noError
This error code means that no error occured so far

elfCreationError
This error code means that we were unable to create an Elf file to store our probes

tmpCreationError
This error code means that we were unable to open a temporary file at `/tmp/`. A common mistake here is having a `/` in the provider name, which will be interpreted by the operating system as a folder.

sharedLibraryOpenError
This error code means that we were unable to open the shared library that we just created

symbolLoadingError
This error code means that the we were unable to load a symbol from the shared library we just created

sharedLibraryCloseError
This error code means that we were unable to close the shared library for this provider

struct **SDTProbeList_t**
Represents a linked-list of *SDTProbe_t*. Shouldn't be handled manually, use *providerAddProbe()* instead.

SDTProbe_t **probe**

struct SDTProbeList_ ***next**

2.4.2 Functions

SDTProvider_t ***providerInit** (const char **name*)
This function received a name as argument, creates a *SDTProvider_t* with all attributes correctly initialized and then returns a pointer to it, or `NULL` if there was an error.

Parameters

- **name** (*string*) – The name of this provider

Returns A pointer to the new provider

Return type SDTProvider_t*

SDTProbe_t ***providerAddProbe** (*SDTProvider_t* **provider*, const char **name*, int *argCount*, ...)
This function received a *SDTProvider_t* created by *providerInit()*, a name and any number of `ArgType_t` (you must pass the actual number of arguments as argCount). It will then create a *SDTProbe_t* with all attributes correctly initialized and register it to the given *SDTProvider_t*. The return would be a pointer to the created *SDTProbe_t*, or `NULL` if there was an error.

Parameters

- **provider** (`SDTProvider_t*`) – The provider where this probe will be created
- **name** (*string*) – The name of this probe
- **argCount** (*int*) – The number of arguments accepted by this probe
- `...` (`SDTArgTypes_t`) – Any number of arguments (number of arguments must match argCount)

Returns A pointer to the new provider

Return type SDTProbe_t*

int **providerLoad**(*SDTProvider_t *provider*)

When you created all probes you wanted, you should call this function to load the provider correctly. The returning value will be `0` if everything went well, or another number otherwise.

After calling this function, all probes will be efectively available for tracing, and you shouldn't add new probes or load this provider again.

Parameters

- **provider** (`SDTProvider_t*`) – The provider to be loaded

Returns A status code (`0` means success, other numbers indicates error)

Return type int

int **providerUnload**(*SDTProvider_t *provider*)

Once you don't want your probes to be available anymore, you can call this function. This will clean-up everything that *`providerLoad()`* did. The returning value will be `0` if everything went well, or another number otherwise.

After calling this function all probes will be unavailable for tracing, and you can add new probes to the provider again.

Parameters

- **provider** (`SDTProvider_t*`) – The provider to be unloaded

Returns A status code (`0` means success, other numbers indicates error)

Return type int

void **providerDestroy**(*SDTProvider_t *provider*)

This function frees a *`SDTProvider_t`* from memory, along with all registered *`SDTProbe_t`* created with *`providerAddProbe()`*.

Parameters

- **provider** (`SDTProvider_t*`) – The provider to be freed from memory, along with it's probes

void **probeFire**(*SDTProbe_t *probe*, ...)

This function fires a probe if it's available for tracing (which means it will only fire the probe if *`providerLoad()`* was called before).

Parameters

- **probe** (`SDTProbe_t*`) – The probe to be fired
- **...** (*any*) – Any number of arguments (must match the expected number of arguments for this probe)

int **probeIsEnabled**(*SDTProbe_t *probe*)

This function returns `1` if the probe is being traced, or `0` otherwise.

Parameters

- **probe** (`SDTProbe_t*`) – The probe to be checked

Returns `1` if probe is enabled, `0` otherwise

Return type int

Index

P

S