Current state of the Userspace Tracer (UST)

David Goulet
david.goulet@polymtl.ca

August 9, 2010
Tracing Mini-Summit, LinuxCon 2010
Content

1. Quick Overview
2. Basic Functionality
3. Use Case
4. Performance
5. Future Work
Who am I

- B.sc in Computer Science at Université de Sherbrooke (Québec, CA)
- Worked at Revolution Linux for the last 2 years as an infrastructure analyst
- **Now**, Graduate Student at Polytechnique de Montréal with Prof. Michel Dagenais

- Research subject:
  - *Efficient tracing for large scale systems using UST*
• Services and Consulting
• Large Scale Infrastructure
• Thin Client / Applications Server
• All open source software and Linux

For us, **tracing** with low overhead is the name of the game
Quick Overview

Basic Functionality
Use Case
Performance
Future Work
1. Overview (1 of 3)

What is UST?

- Userland tracing
- Low overhead tracer
- Build with LTTng technology
- LTTV or/and TMF trace viewer
- Written by Pierre-Marc Fournier at Polytechnique

➢ How userspace tracing is achieved with UST?
1. Overview (2 of 3)

- *Ex: instrument with tracepoints bind, search and modify requests*
1. Overview (3 of 3)

• Core code is mostly from LTTng
  • Lockless tracer (ring buffer) (Mathieu Desnoyers)
  • Userspace RCU (Paul E. Mckenney and Mathieu Desnoyers)

• Tools for tracing:
  • `ustd` : daemon for collecting data
  • `usttrace` : script for trace start up
  • `ustctl` : control the tracing
Quick Overview
Basic Functionality
Use Case
Performance
Future Work
2. Basic Functionality

2.1 Instrumentation (1 of 3)

• Using *Markers*

```c
servers/slapd/search.c

int do_search(
    Operation *op, /* info about the op to which we're responding */
    SlapReply *rs /* all the response data we'll send */ )
{
    struct berval base = BER_BVNULL;
    ber_len_t siz, off, i;

    trace_mark(ust, search_event, "DN %s", op->o_req_dn.bv_val);

    # ldapsearch -b "dc=rlnx,dc=com" ...

    Output --> { "DN" = "dc=rlnx,dc=com" }
```
2. Basic Functionality

2.1 Instrumentation (2 of 3)

• Using *Tracepoints*

servers/slapd/tp.c (NEW)

```c
#include <ust/marker.h>
#include <time.h>
#include <string.h>
#include "tp.h"

DEFINE_TRACE(ust_ldapsrch);

void ust_ldapsrch_probe(char *dn, time_t o_time)
{
    /* Filter to only get CN from RLNX datacenter DN */
    if(strncmp(dn, "dc=datacenter,dc=rlnx,dc=com", 29) == 0)
        trace_entry(ust, ldapsrch, "req_time %s", o_time);
}

static void __attribute__((constructor)) init()
{
    register_trace_ust_ldapsrch(ust_ldapsrch_probe);
}
```

servers/slapd/search.c

```c
#include "tp.h"

int do_search(
    Operation *op, /* info about the op to which we'...
    SlapReply *rs /* all the response data we'll send */
{...

    trace_ust_ldapsrch(op->o_req_dn.bv_val, op->o_time);
```
2. Basic Functionality

2.1 Instrumentation (3 of 3)

- What can be instrumented?
  - Multi-threaded applications
  - Signal Handlers
  - Shared Libraries

Programs must be compile with **libust (-l ust)**

Also, **LD_PRELOAD=/usr/local/lib/libust.so.0** if some part of the app is instrumented (Ex: dynamic libraries) but not the main app.
2. Basic Functionality

2.2 LD_PRELOAD (1 of 2)

- What if we can't instrument?
  - Old application (difficult to recompile)
  - Instrumentation MAY be too much work

Solution:

```
# gcc -fpic -shared -ldl -lust -O2 -o libldaptrace.so CODE.c
# LD_PRELOAD=./libldaptrace.so ./slapd
```

* But only for linked symbols
2. Basic Functionality

2.2 LD_PRELOAD (2 of 2)

- `malloc` example

```c
void *malloc(size_t size)
{
    static void *(*plibc_malloc)(size_t size) = NULL;
    void *retval;
    if(plibc_malloc == NULL) {
        plibc_malloc = dlsym(RTLD_NEXT, "malloc");
        if(plibc_malloc == NULL) {
            fprintf(stderr, "mallocwrap: unable to find malloc\n");
            return NULL;
        }
    }
    retval = plibc_malloc(size);
    trace_mark(ust, malloc, "size %d ptr %p", (int)size, retval);
    return retval;
}
```
2. Basic Functionality

2.3 Tools (1 of 3)

- *ustd*(1)
  - Collects trace data
  - Writes that data to disk

Communication: `traced app <-> ustd`

- Unix Socket: `/tmp/ust-app-socks/PID`
2. Basic Functionality

2.3 Tools (2 of 3)

- *ustctl(1)*
  - Control the tracing of userspace apps.
  - create/alloc/start/stop/destroy a trace
  - enable/disable/list markers
  - set/get subbuffers info
2. Basic Functionality
2.3 Tools (3 of 3)

• **usttrace**(1)
  - Script tool for trace recording

1. Creates a daemon
2. Enables all markers
3. Runs the command
4. At the end, prints the location of the trace

```
# usttrace ./slapd
[...] Waiting for ustd to shutdown...
Trace was output in: /home/dave/.usttraces/raoul-20100805143324547483745
```
Quick Overview
Basic Functionality
Use Case
Performance
Future Work
3. Use Case (1 of 3)

- We've instrumented OpenLDAP
  - `ldap_seach`
    - `trace_mark(ust, ldap_search, "Filter %s", filter.var)`
  - `ldap_modify`
    - `trace_mark(ust, ldap_modify, "CN %s", cn.var)`
  - `ldap_bind`
    - `trace_mark(ust, ldap_bind, "DN %s", userdn.var)"`
3. Use Case (2 of 3)

1. Launch OpenLDAP normally
   
   ```bash
   # /etc/init.d/slapd (PID : 1234)
   ```

2. **Setup** the trace
   
   ```bash
   # ustd
   # ustctl --enable-marker ust/ldap_search 1234
   # ustctl --create-trace 1234
   ```

3. **Start** the trace
   
   ```bash
   # ustctl --start-trace 1234
   ```

4. Enable a second marker
   
   ```bash
   # ustctl --enable-marker ust/ldap_modify 1234
   ```
3. Use Case (3 of 3)

5. Stop the trace

# ustctl --stop-trace 1234
# ustctl --destroy-trace 1234

6. View the trace with **Ittv/tmf**

```
# ls /tmp/usttrace/1234_5502332342922775910
..  metadata_1  metadata_3  metadata_5  metadata_7  ust_1  ust_3  ust_5  ust_7
.   metadata_0  metadata_2  metadata_4  metadata_6  ust_0  ust_2  ust_4  ust_6
```

- Ittv-gui or Eclipse TMF
Quick Overview
Basic Functionality
Use Case
Performance
Future Work
4. Performance (1 of 4)

Who cares about tracing in *userland*?

Good debug for a lots of large apps:

- Apache, LDAP, Samba, etc
4. Performance (2 of 4)

1. **Flexibility**
   - Enable what marker you want... at any time!
   - Granularity

2. **Synchronization**
   - Buffer protected
   - Program crashes, data still available
4. Performance (4 of 4)

3. Signal Safe

4. Scalability
   - Multithreaded applications

5. Low-intrusiveness
   - Applications normal behaviour not changed
   - Block signal for listener thread
4. Performance (3 of 4)

- Quick look at performance
  - `trace_mark`:
    - \(\approx 247\) ns / per event
  - `tracepoint + trace_mark`:
    - \(\approx 271\) ns / per event
  - `tracepoint + custom_probe`:
    - \(\approx 189\) ns / per event

1000 iterations
5 000 000 events
1 thread
Quick Overview
Basic Functionality
Use Case
Performance
Future Work
5. Future Work

- Session tracing
- Time sync with Kernel (vdso?)
- `TRACE_EVENT` integration
- UST streaming: TCF (almost there)
- Event filtering (Nils Carlson @ Ericsson)
Questions ?