Writing Babeltrace 2 plugins

Tracing summit 2019 (20 August 2019)

Simon Marchi <simon.marchi@efficios.com>

simark on GitHub/IRC/SO
Contents

1) Babeltrace 2, reminder and update
2) Important concepts
3) Let’s write some components
4) Next steps
5) Questions

Questions?
Note slide number and ask at the end.
Babeltrace 2, reminder and status
Babeltrace 2, a reminder

- Process, analyze, convert traces of various formats.
- Shortcomings of Babeltrace 1:
  - Intermediary representation (IR) coupled to CTF
  - No external plugin system
- Cross-platform: Linux, macOS, Windows
- https://github.com/efficios/babeltrace
Babeltrace 2, a status update

• All expected API changes for 2.0 are **done**.
• **Documentation** is being written.
• **RC1** is expected in *A Few Weeks™*
Important concepts
It all starts with a graph

A graph is made of several components connected together
Communication between components

- **Messages** flow from upstream components to downstream components.

- Some message types:
  - Stream beginning message
  - Event message
  - Stream end message
Lifecycle of a graph (simplified)

• User adds components and connects them
• Sinks create iterators on their input ports
• The graph asks sinks to consume from their iterators
• When all iterators of all sinks have reached the end, the graph execution has completed successfully.
The library vs the command-line tool

- `libbabeltrace2` is a library to build and execute a graph
  - C and Python bindings
- `babeltrace2` is a CLI tool build around `libbabeltrace2` to build and run a graph from the command line
Your component classes...

- ... can be written in C or Python
- ... can be written directly in your application that uses libbabeltrace2 (either in C or Python)
- ... can be distributed as plugins, loaded by another application using libbabeltrace2 (including the babeltrace2 CLI)
- C plugins are distributed as .so/.dll shared libraries
- Python plugins are distributed as .py source files
Let’s write some components
Boilerplate for a Python plugin

- Named `bt_plugin_*.py`
- Registration: `bt2.register_plugin(__name__, 'foo')`
My first sink

- In \_\_init\_, create input port.
- In \_\_user\_graph\_is\_configured, create iterator on the input port (on the upstream component).
- In \_\_user\_consume, consume messages from the iterator and do something useful with them.

Let’s go try it 🖥️.
My first source

• In `__init__`, create trace class, stream class, event class and output port.

• Define source’s iterator class.

• In the iterator’s `__next__`, return some messages.

Let’s go try one 🖥️.
Next steps
Next steps

• Use **parameters** for component configuration.

• Support `babeltrace.support-info` query to allow for automatic source discovery.
  
  • This makes `babeltrace2 <mytrace>` just work.

• Use error error system to provide user-friendly error messages.
Plugin examples

- Multiple in-tree component classes
- Some examples here [1]:
  - CAN Bus messages source
  - Plot-drawing sink

Questions

Thanks for your attention. Any questions?
Bonus slides!
Queries

Queries are a way to poke a component class to get some information, before a component of that class is instantiated.

- Can be queried from the CLI or programmatically.
- Arbitrary query object (a string) and parameters.
- In Python, implement static/class method `_user_query`. 

Automatic source discovery

- User-friendly alternative to having to specify components explicitly (with `-c source.foo.bar --params ...`).
- When a non-option string is passed to the CLI (e.g. `babeltrace2 mytrace`), it queries all known source component classes (CC) with the `babeltrace.support-info` object. CC respond with a weight in the [0, 1] range.
- Recurses into directories.
- Works with paths (files and directories) and other strings (e.g. `babeltrace2 net://somehost:1234`).
Error handling

When an error occurs, your plugin can append error causes, such that when a critical failure happens, the user can see precisely where things went wrong.

- In Python, simply raise an exception, the native side translates it to an error cause.
- In C, you have to do it manually, with e.g. `BT_CURRENT_THREAD_ERROR_APPEND_CAUSE_FROM_COMPONENT`
Error handling

Here’s an example of an error stack printed by the CLI.

ERROR: [Babeltrace CLI] (/home/smarchi/src/babeltrace/src/cli/babeltrace2.c:2364)
  Cannot create components.

CAUSED BY [Babeltrace CLI] (/home/smarchi/src/babeltrace/src/cli/babeltrace2.c:2187)
  Cannot create component: plugin-name="demo", comp-cls-name="MyFirstSource", comp-cls-type=0, comp-name="source.demo.MyFirstSource", comp-class-type=SOURCE, comp-class-name="MyFirstSource", comp-class-partial-descr="", comp-class-is-frozen=false

CAUSED BY [Babeltrace library] (/home/smarchi/src/babeltrace/src/lib/graph/graph.c:1343)
  Component initialization method failed: status=ERROR, comp-addr=0x55fbeca8890, comp-name="source.demo.MyFirstSource", comp-class-type=SOURCE, comp-class-name="MyFirstSource", comp-class-partial-descr="", comp-class-is-frozen=false

  Traceback (most recent call last):
    File "/tmp/babeltrace/lib/python3.6/site-packages/bt2/component.py", line 522, in _bt_init_from_native
      self.__init__(params, obj)
    File "/bt_plugin_foo.py", line 34, in __init__
      this_is_an_error()
NameError: name 'this is an error' is not defined
Details sink

The `sink.text.details` component class (provided with BT2) prints details about all messages it receives (even what is not directly user-visible), in a deterministic way. Useful for:

• Debugging while developing a source of filter.

• Automated tests, compare the `sink.text.details` output to an expected output.

• Verifying that the Python component you are re-writing in C provides the same results.
Details sink

An example:

Stream class (ID 0):
  Supports packets: No
  Supports discarded events: No
  Supports discarded packets: No
Default clock class:
  Frequency (Hz): 1,000,000,000
  Precision (cycles): 0
  Offset (s): 0
  Offset (cycles): 0
  Origin is Unix epoch: Yes
Event class: my-event (ID 0):

[Unknown]
{Trace 0, Stream class ID 0, Stream ID 0}
Stream beginning:
  Trace:
    Stream (ID 0, Class ID 0)

[123 cycles, 123 ns from origin]
{Trace 0, Stream class ID 0, Stream ID 0}
Event `my-event` (Class ID 0):

[Unknown]
{Trace 0, Stream class ID 0, Stream ID 0}
Stream end