locationd Documentation

Release 4.0.0

Canonical Ltd.

Jan 30, 2017

Contents

1 Introduction **3**
1.1 Vocabulary . . . 3
1.2 Architectural Overview . . . 3
1.3 Privacy & Access Control . . . 4

2 CLI **5**
2.1 Snap-Specific Command Names . . . 5
2.2 Testing Scenarios . . . 5

3 API **7**

4 Hacking **9**
4.1 Building the code . . . 9
4.2 Running the tests . . . 9
4.3 Coverage . . . 9
4.4 Code style . . . 10
4.5 Thread and address sanitizer . . . 11
4.6 Updating symbols file . . . 11
4.7 ABI compliance test . . . 11

5 Indices and tables **13**

Contents:

CHAPTER 1

Introduction

locationd is a central hub for multiplexing access to positioning subsystems available via hard- and software. It provides a client API offering positioning capabilities to applications and other system components, abstracting away the details of individual positioning solutions.

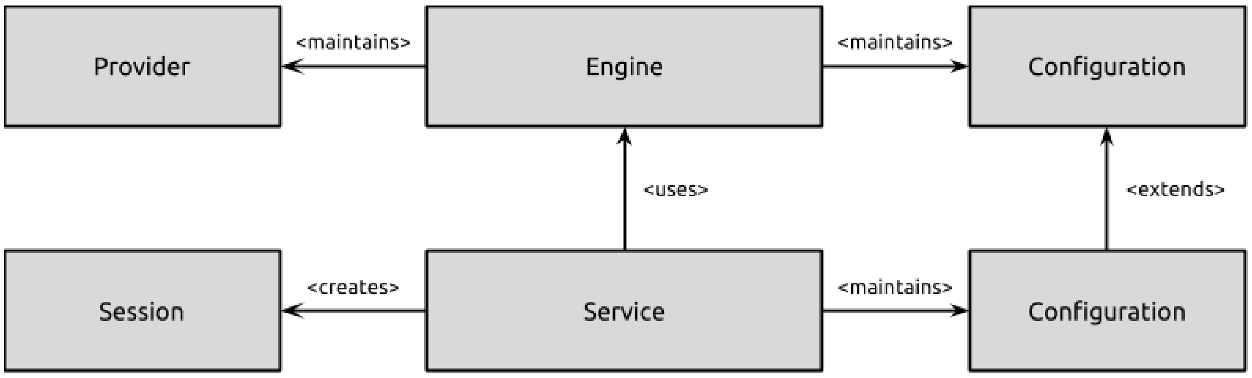
1.1 Vocabulary

To make the remainder of this documentation as easily understandable as possible, we start over with introducing some vocabulary:

- `Engine`: Responsible for handling input from multiple positioning subsystems and maintaining the state of the overall system. Think about it like the heart of the system.
- `Provider`: A positioning subsystem that feeds into the positioning engine. Common examples are a GPS provider or a network-based positioning provider.
- `Service`: The point of entry for applications and services that would like to receive position data.
- `Session`: In order to receive position information, every application or service has to create a session with the location Service. Session creation is subject to security mediation and contextual prompting.
- `Update`: An update is a timestamped entity to a certain type of data.
- WGS84: The coordinate system that is used throughout the entire location subsystem.

1.2 Architectural Overview

The high-level architecture of the service is shown in the following diagram:



In this diagram, the configuration of the engine refers to:

- The current state of any satellite-based positioning subsystems. Can either be off or on.
- The current state of reporting facilities responsible for harvesting wifi and cell id measurements together with location information and sending them off to remote services. Can either be off or on.
- The overall state of the engine. Can either be off, on or active.

The Service takes this configuration and exposes it to client applications. In addition, mainly for debugging purposes, the set of currently visible satellites (if any) is maintained and exposed to privileged client applications. The service supports multiple different satellite-based positioning operating in parallel. The following GNSSs are known:

- Beidou: People's Republic of China's regional system, currently limited to Asia and the West Pacific.
- Galileo: A global system being developed by the European Union and other partner countries, planned to be operational by 2014 (and fully deployed by 2019).
- Glonass: Russia's global navigation system. Fully operational worldwide.
- GPS: Fully operational worldwide.
- Compass: People's Republic of China's global system, planned to be operational by 2020.
- IRNSS: India's regional navigation system, planned to be operational by 2014, covering India and Northern Indian Ocean.
- QZSS: Japanese regional system covering Asia and Oceania.

1.3 Privacy & Access Control

Location information is highly privacy relevant. For this reason, the locationd is deeply integrated with AppArmor and Ubuntu's overall trust infrastructure. Every incoming session request is validated and if in doubt, the user is asked to explicitly grant trust to the application requesting access to positioning information. Please see [@ref com::ubuntu::location::service::PermissionManager] for further details.

In addition, the locationd allows for selectively adjusting the accuracy and reporting setup of the location Engine to provide further fine-grained control over the exposed data to user. Within this setup, a user is able to entirely disable all positioning.

CHAPTER 2

CLI

locationd offers a command-line interface for controlling and monitoring the service. The following commands are available:

- `list`: Lists all provider implementations known to the service.
- `monitor`: Connects to a locationd instance, monitoring its activity.
- `provider`: Executes a known provider implementation in an out-of-process sandbox.
- `run`: Executes the service.
- `status`: Queries the status of a service instance.
- `test`: Executes runtime tests against known provider implementations.

For all of the commands, an exit status of 0 indicates success. An exit status of 1 indicates an error. Normal output goes to stdout, while all errors/warnings are output to stderr.

2.1 Snap-Specific Command Names

If you are using the cli from a snap (`snap install locationd --channel edge`), the commands will be wrapped up for you in a convenient way, following the pattern locationd.$COMMAND. With that, if you want to check on the status of the service, simply run:

```
$ locationd.status
```

2.2 Testing Scenarios

For testing purposes, it is often handy to inspect position/velocity/heading estimates on the command line. The `monitor` command helps here. It connects to the service, starts the positioning engine and outputs position estimates to stdout until it receives a SIGTERM.

CHAPTER 3

API

Location service exposes a DBus API to interact with a service instance. We do not expose introspection for the API, yet. Instead, we provide a C++ client API that abstracts away from the underlying IPC mechanism.

A client application then uses the API to establish a session with a service, register observers to receive updates and to control the status of updates. The following snippet illustrates basic usage of the client API:

```
auto service = location::connect_to_service(...);
auto session = service->create_session_for_criteria(location::Criteria{});

session->updates().position.changed().connect([this](const location::Update
↪<location::Position>& pos)
{
  std::cout << pos << std::endl;
});

session->updates().heading.changed().connect([this](const location::Update
↪<location::units::Degrees>& heading)
{
  std::cout << pos << std::endl;
});

session->updates().velocity.changed().connect([this](const location::Update
↪<location::units::MetersPerSecond>& velocity)
{
  std::cout << pos << std::endl;
});

session->updates().position_status =␣
↪location::Service::Session::Updates::Status::enabled;
session->updates().heading_status =␣
↪location::Service::Session::Updates::Status::enabled;
session->updates().velocity_status =␣
↪location::Service::Session::Updates::Status::enabled;
```

CHAPTER 4

Hacking

4.1 Building the code

By default, the code is built in release mode. To build a debug version, use

```
$ mkdir builddebug
$ cd builddebug
$ cmake -DCMAKE_BUILD_TYPE=debug ..
$ make
```

For a release version, use -DCMAKE_BUILD_TYPE=release

4.2 Running the tests

```
$ make
$ make test
```

Note that "make test" alone is dangerous because it does not rebuild any tests if either the library or the test files themselves need rebuilding. It's not possible to fix this with cmake because cmake cannot add build dependencies to built-in targets. To make sure that everything is up-to-date, run "make" before running "make test"!

4.3 Coverage

To build with the flags for coverage testing enabled and get coverage:

```
$ mkdir buildcoverage
$ cd buildcoverage
$ cmake -DCMAKE_BUILD_TYPE=coverage
$ make
$ make test
$ make coverage
```

Unfortunately, it is not possible to get 100% coverage for some files, mainly due to gcc's generation of two destructors for dynamic and non- dynamic instances. For abstract base classes and for classes that prevent stack and static allocation, this causes one of the destructors to be reported as uncovered.

There are also issues with some functions in header files that are incorrectly reported as uncovered due to inlining, as well as the impossibility of covering defensive assert(false) statements, such as an assert in the default branch of a switch, where the switch is meant to handle all possible cases explicitly.

If you run a binary and get lots of warnings about a "merge mismatch for summaries", this is caused by having made changes to the source that add or remove code that was previously run, so the new coverage output cannot sensibly be merged into the old coverage output. You can get rid of this problem by running

```
$ make clean-coverage
```

This deletes all the .gcda files, allowing the merge to (sometimes) succeed again. If this doesn't work either, the only remedy is to do a clean build.

If lcov complains about unrecognized lines involving '=====', you can patch geninfo and gcovr as explained here:

https://bugs.launchpad.net/gcovr/+bug/1086695/comments/2

4.4 Code style

We use a format tool that fixes a whole lot of issues regarding code style. The formatting changes made by the tool are generally sensible (even though they may not be your personal preference in all cases). If there is a case where the formatting really messes things up, consider re-arranging the code to avoid the problem. The convenience of running the entire code base through the pretty-printer far outweighs any minor glitches with pretty printing, and it means that we get consistent code style for free, rather than endlessly having to watch out for formatting issues during code reviews.

As of clang-format-3.7, you can use

```
// clang-format off
void    unformatted_code  ;
// clang-format on
```

to suppress formatting for a section of code.

To format specific files:

```
${CMAKE_BINARY_DIR}/tools/formatcode x.cpp x.h
```

If no arguments are provided, formatcode reads stdin and writes stdout, so you can easily pipe code into the tool from within an editor. For example, to reformat the entire file in vi (assuming ${CMAKE_BINARY_DIR}/tools is in your PATH):

```
1G!Gformatcode
```

To re-format all source and header files in the tree:

```
$ make formatcode
```

4.5 Thread and address sanitizer

Set SANITIZER to “thread” or “address” to build with the corresponding sanitizer enabled.

4.6 Updating symbols file

To easily spot new/removed/changed symbols in the library, the debian package maintains a .symbols file that lists all exported symbols present in the library .so. If you add new public symbols to the library, it’s necessary to refresh the symbols file, otherwise the package will fail to build. The easiest way to do that is using bzr-builddeb:

```
$ bzr bd -- -us -uc -j8  # Don't sign source package or changes file, 8 compiles in␣
↪parallel
$ # this will exit with an error if symbols file isn't up-to-date
$ cd ../build-area/location-service-[version]
$ ./obj-[arch]/tools/symbol_diff
```

This creates a diff of the symbols in /tmp/symbols.diff. (The demangled symbols from the debian build are in ./new_symbols.)

Review any changes in /tmp/symbols.diff. If they are OK:

```
$ cd -
$ patch -p0 < /tmp/symbols.diff
```

4.7 ABI compliance test

To use this, install abi-compliance-checker package from the archives.

You can use abi-compliance-checker to test whether a particular build is ABI compatible with another build. The tool does some source-level analysis in addition to checking library symbols, so it catches things that are potentially dangerous, but won’t be picked up by just looking at the symbol table.

Assume you have built devel in src/devel, and you have a later build in src/mybranch and want to check that mybranch is still compatible. To run the compliance test:

```
$ cd src
$ abi-compliance-checker -lib libunity-scopes.so -old devel/build/test/abi-compliance/
↪abi.xml -new mybranch/build/test/abi-compliance/abi.xml
```

The script will take about two minutes to run. Now point your browser at

```
src/compat_reports/libunity-scopes.so/[version]_to_[version]/compat_report.html
```

The report provides a nicely layed-out page with all the details.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search