Introducing DBus-ASIO

How and why we built a new D-Bus library from the ground up

FOSDEM
3rd February 2019

by Steven Goodwin, BrightSign LLC

@marquis de geek

www.MarquisdeGeek.com

https://github.com/dbus-asio

DBus-ASIO

- What is D-Bus?
- What is ASIO?
- Existing libraries
- Why this one was built...
- ...And how

What is D-Bus?

- D-Bus is an inter-process mechanism that allows communication between multiple programs running on the same machine. It does this by serializing method calls, with parameter data.

- Basic workings:
  - Dbus-daemon
  - Apps say ‘Hello’ to register themselves
  - Uses sockets
  - Supports namespaces
  - Supports enumeration

- Example:
  - dbus-send --session --dest=org.gnome.ScreenSaver --type=method_call --print-reply /org/gnome/ScreenSaver org.gnome.ScreenSaver.Lock

What is ASIO?

- Asynchronous input/output
- ASIO is a cross-platform C++ library for network and low-level I/O programming.
- Part of Boost
- Intended for *external* resources, rather than threads

DBus-ASIO

- Therefore, this is a library which:
  - Open a socket to the dbus-daemon
  - Say ‘Hello’ to the daemon
  - Serialise a method call into the correct format
  - Send this formatted message to the dbus-daemon
  - Listen on the socket for the reply, de-serialises it, and sends it to the correct callback

Current solutions

- Existing libraries
  - libdbus
  - GDBus
  - QTBus
  - libdbus-c++
- And others…

What is our use case?

What is our use case?

- Multi-threaded
- Embedded, 32 and 64 bits
- Few dependencies
- Active development (!?)
- License compatible

Our options

- **Build** on an existing library
  - Learning the codebase
  - Changing it in an idiosyncratic manner
  - Will upstream changes get taken?
  - Risk of regressions. Tests?
- **Fork** an existing library
  - Will we be continuously merging?
- Write a **new** one
  - Can we afford the time?
  - Do we have the skill set?

Why we built it

Design decisions

- Focus on the use case
  - Small
  - No features/fluff “because we can”
  - No library backward compatibility
  - No need to support old tool chain(s)
- Which meant:
  - So, the code would be modern and use whatever language features were appropriate.

Design decisions

- Which language?
  - Go
  - Rust
  - C
  - C++ 11
  - C++ 14

(Nothing wrong with C!)

```
static DBusList*
alloc_link (void *data)
{
  DBusList *link;

  if (!_DBUS_LOCK (list))
    return FALSE;

  if (list_pool == NULL)
    {
      list_pool = _dbus_mem_pool_new (sizeof (DBusList), TRUE);

      if (list_pool == NULL)
        {
          _DBUS_UNLOCK (list);
          return NULL;
        }

      link = _dbus_mem_pool_alloc (list_pool);
      if (link == NULL)
        {
          _dbus_mem_pool_free (list_pool);
          list_pool = NULL;
          _DBUS_UNLOCK (list);
          return NULL;
        }
    }
  else
    {
      link = _dbus_mem_pool_alloc (list_pool);
    }

  if (link)
    link->data = data;

  _DBUS_UNLOCK (list);

  return link;
}
```

But...

- return new DBusList(data);

Design decisions – The Language

- C++11 or C++14
  - We’re always moving forward
  - Picking up legacy users
  - Toolchain restrictions

Design decisions – The Library

- Which async library to use?

Design decisions – ASIO

- Nothing as advanced in the standard
- Destined to standardisation
- Boost has history

Design decisions – Not BrightSign

- Logging
- Streaming
- Sockets code
- Threaded libraries

- std::unique_ptr<boost::asio::io_service::work> work(new
  boost::asio::io_service::work(m_io_service));
- std::thread io_service_thread(boost::bind(&boost::asio::io_service::run, &m_io_service));
-
- m_socket.connect(m_Busname.c_str());

Design decisions - API compatibility

- A couple of words on this...

Design decisions - API compatibility

- A couple of words on this…
- … we didn’t!

How we built it

Step 1 - Pre-production

- Read the spec
  - https://dbus.freedesktop.org/doc/dbus-specification.html
- Fiddling with dbus-send
- Experiments with d-feet
- Using socat

Step 2 - MVP

- Says ‘Hello’ to the daemon
- Sent pre-built messages
- Build a pre-built message from code
- Plan tests

Step 3 - Implementation

- Threads throughout
- Callbacks with lambda
- Easier to send data, than to receive
- Start simple data types, and build up

Step 4 – The problems

- Serial protocol
- Padding non-aligned data
- Zero-length strings
- Variable data types
  - boost::any

An example

```
DBus::Native native("/var/run/dbus/system_bus_socket");

 native.BeginAuth(DBus::AuthenticationProtocol::AUTH_BASIC);

 DBus::MethodCall hello("/org/freedesktop/Dbus",
                        "org.freedesktop.DBus", "Hello");

 native.sendMethodCall(hello,
   [] (const DBus::MessageType::MethodReturn &msg) {
     DBus::Type::General data =
                 DBus::Type::unmarshallString(msg.m_Body);

     printf("REPLY FROM HELLO : This is our unique name : %s\n",
            DBus::Type::asString(data).c_str());
   }
 );
```

Schedule

- 4 weeks to build
  - 1 developer (full-time)
- 6 weeks to review
  - 3 developers (not full-time)
- Far too long deciding on a name :)

Conclusions

- Sometimes re-writing is necessary
- Stick to the known use cases
- Choose the language to fit both task and team
- Step-by-step debugging comes first
- Writing to protocols is easier than reading from

Any Questions?

@marquis de geek

www.MarquisdeGeek.com / www.BrightSign.biz (yes, we're hiring)

https://github.com/dbus-asio

FOSDEM Scorecard:

Attended: 18
Diaries written:16
Talks given: 12 (on 10 different topics)

marquisdegeek.com/words_fosdem

Any Questions?

@marquis de geek

www.MarquisdeGeek.com / www.BrightSign.biz (yes, we're hiring)

https://github.com/dbus-asio

FOSDEM Scorecard:

Attended: 19
Diaries written:16
Talks given: 13 (on 11 different topics)

marquisdegeek.com/words_fosdem