babelc

“Now the whole world had one language and a common speech”

- **Genesis 11:1**

Contents of this presentation

- The first part is an overview of babelc in general
- The second part describes the inner design

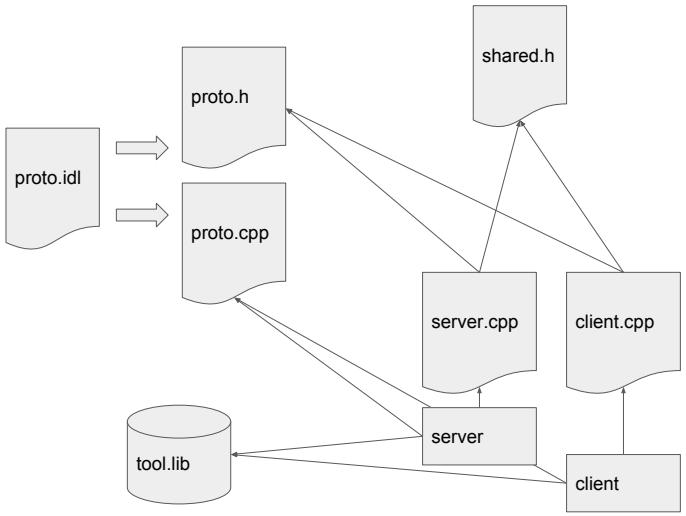
What is babelc?

- A C++ 14 code extension generator
  - Input is a C++ subset
  - A valid babelc code snippet is always a valid C++ 14 snippet
  - But not vice versa!
- An extensible POD transformer
  - Support of today:
    - POD-to-text, text-to-POD
    - POD-to-dbus, dbus-to-POD
- An extensible IPC front end
  - Actual IPC is determined by user defined policies written in C++ 14
    - dbus is supported today
- GPL2
  - Generated code is license free

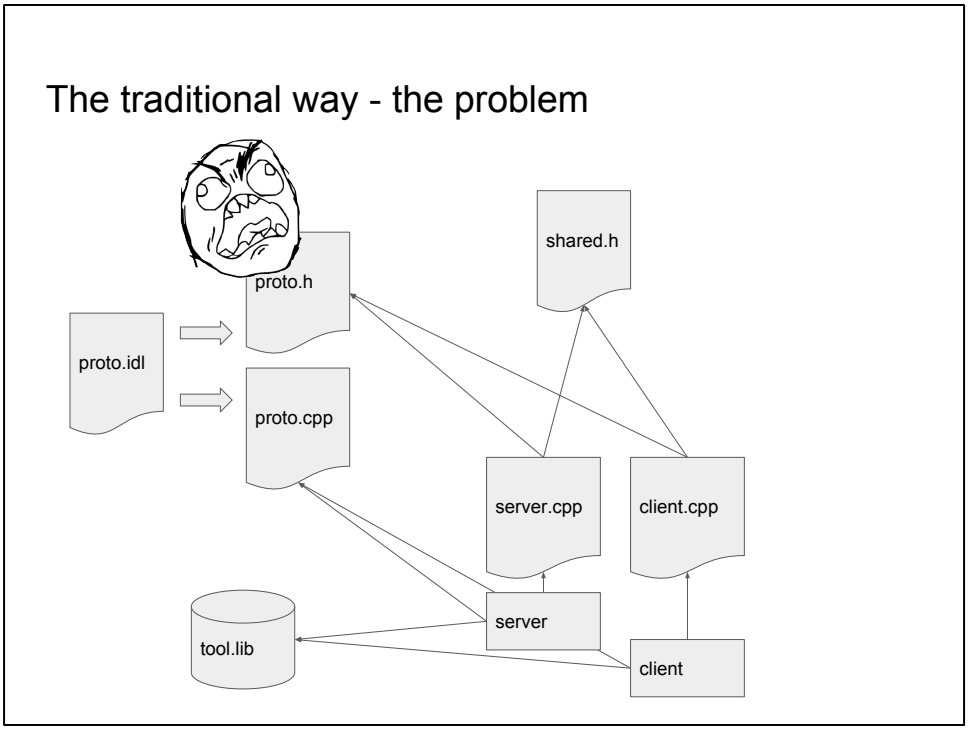
Why babelc?

- I needed a fast way of creating IPC code and couldn't find one
  - IPC shall be a solution - not a problem
- I like to focus upon "what" - not "how"
  - I find it easier to "think" in C++ than most IDL languages
- I like the idea where the IPC mechanism can be changed
  - Needless to say with a minimum of effort
- I don't like hand crafting conversion code (to text, serializing…)
  - I always get it wrong…
  - It sucks to test
- I don't like contagious software
  - E.g. a whole eco system (QT), no use of the standard libraries, stupid #defines...

The traditional way

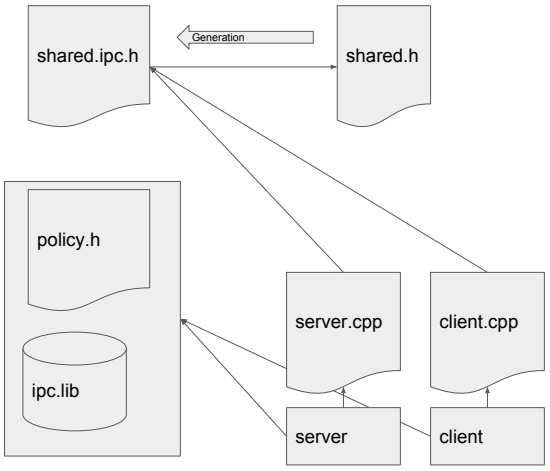


Typically, when working in the traditional way, you use an Interface Description Language (IDL) to define your interface. This text is then translated to a something your ordinary compiler can understand, e.g. a C++ header file and (most often) a C++ implementation file. These are compiled and linked into your application. Quite often you will also need a library tied to the specific language and IDL compiler.
The strength of this way of working is that the IDL is language agnostic - for popular IDLs there will be translations into just about any language.
There is however a drawback to this.

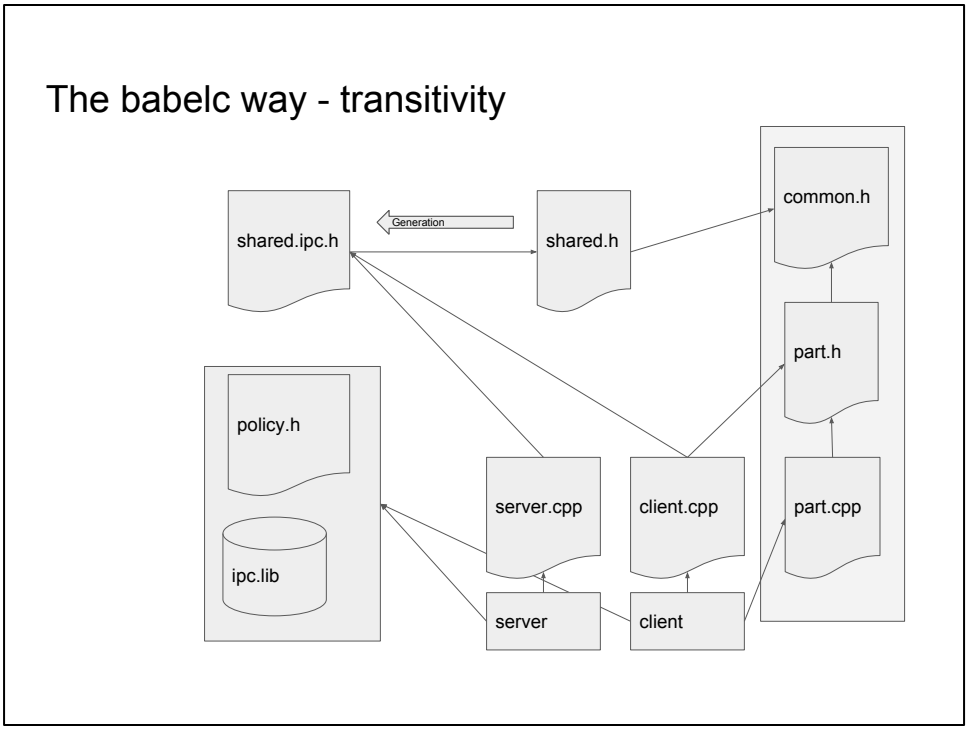


To be honest; I have yet to see an IDL compiler that generates code that is at least remotely usable as is. Code may be clunky or it may force upon you yet another string and container library to learn and use.
Quite often you need to do wrappers for the generated code to avoid its peculiarities and dependencies to leak into your application code. Because if you let this leakage happen, you are most likely stuck with this IDL compiler for ever - and in worst case with a *specific version* of it! The chances that you will be able to switch IPC mechanism (if that's what your IDL helps you with) are indeed slim.
You may argue that this is an unlikely case, but imagine for a moment that your IPC is working well except for in one case where the performance is bad. Imagine the sweetness of the possibility to change the IPC mechanism in only that particular case.

The babelc way



When using babelc, you instead write the interface in a subset of C++ 14. This subset allows all numeric types, bool, enums, standard containers, smart pointers and POD structs of all these and also abstract interfaces. You are in complete control of how and which of those types that are used from that subset.
From that header file, babelc generates another header file which you include in your application. I.e. there is a one-to-one correspondence between your header file and the generated one, which of course make it easier for an automated build system.



Since your interface definition is included in the generated file, there is no need to include it directly in those parts of the code that need to use the capabilities of the generated code.
It is also possible to let your interface definition share common definitions with the part of your code that is not interested in the interface definition itself. Of course, common types used by a babelc definition must follow the babelc subset (but shared types that are not used by the babelc definition does not have to follow these rules).

So what does babelc support?

- Simple types
  - <cstdint>, bool, double, scoped enum with underlying type
- Character data
  - std::string - no pesky char stars
- Some containers
  - vector, list, array, map of the supported types, boost::variant, boost::optional
- Managed pointers
  - shared_ptr, unique_ptr
- POD structs
  - Recursive structs consisting of elements of the above types
  - No constructors yet
- Abstract interfaces
  - Using supported types as return type and argument types
- Remote exception propagation
  - A remote exception is passed backed to the caller “sensibly”

In addition to this, template instantiation must be explicit in order for all types to have a name.
Certain babelc backends may have restrictions of their own (e.g. it is not possible to pass a signed byte or an array of booleans over the D-Bus protocol - don’t ask me how I know this).

babelc interoperability

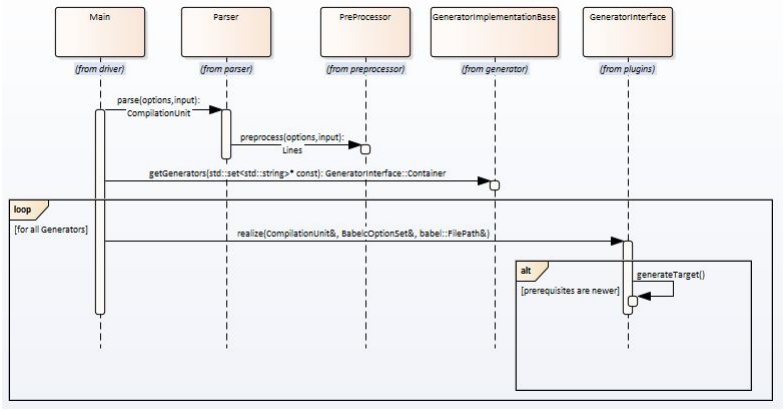
- Dbus works nice
  - You can extract XML representations by introspection for use with other languages
  - In case of need you can mirror a third party interface in C++ - and automatically check it
- It is not hard to write an IDL generator for other protocols
  - E.g. for protobuf, JSON schema…

The policy based ipc implementation makes it rather easy to change the actual ipc used. It also allows for publishing babelc generated information on several interfaces with the same generated code.

Design

at a rather high level

babelc - Generation flow



The babelc application analyzes the input source in several steps. First the gcc preprocessor is let those on the code (that is why babelc need all your compiler flags). If this step fails, the diagnostics are output and the rest of the work is abandoned. If succesful, the preprocessed lines are passed to gcc itself in order to check for syntactic or semantic C++ errors. Again, if this fails the diagnotics are output and babelc aborts.
A succesful conclusion, however, will lead to further checks by babelc, which can bathe in full bliss in the knowledge that the code to analyze is correct C++ code. This makes the code much simpler to parse using regular expressions.
If the internal checks made by babelc are succesfull, the generators requested on the command line are looked up and called to generate code.
A historicsl note; babelc uses gcc an parses its output. Another way of doing it would be to use clang as front end an use clangs interface to gain access to the parsed code. The reason for not doing this is twofold; pro primo I wanted to hone my skills in C++ 14 and pro secundo the clang interface is in C (why this is so is beyond my comprehension; compiler innards are about as close to an object oriented school book example as they get).

babelc - Generation

- The preprocessor and parser are the same for all generators
- The generators shares a common base class
  - Methods for IO
  - Methods for common constructs
  - Methods for prerequisite checks

To keep the front end common is of course paramount.
Sometimes there is a need to convey details from the source code to a specific backend but this is handled by C++ 14 attributes. These have a general syntactic format and are only parsed and not analyzed by the frontend.

babelc - Preprocessor

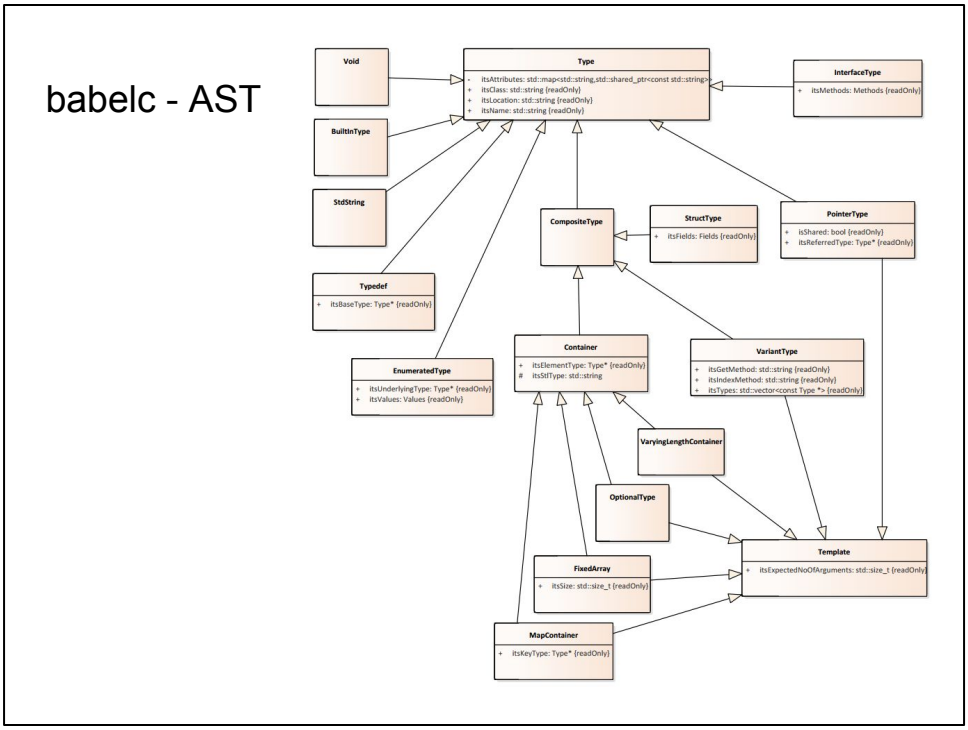
- The preprocessor uses the passed compiler to preprocess the input
  - It forks and executes the compiler and reads the result from the child process output
- If the compiler returns an error status, further babelc processing is cancelled
- Output is returned a list of instances of a `Line`
  - Holds one line of text
  - Has capability to hold name or originating file and line number
- A `LinesIterator` class is provided which enables character iteration over several lines
  - Makes it easy to use regular expression matching spanning several lines

The preprocessing step is made to avoid implementing an own preprocessor. There is also another good reason; to get at the file and line number information that the preprocessor outputs. This is later used in diagnostic messages.

babelc - Parser

- The parser accepts the Lines produced by the preprocessor as input and performs the following steps:
  a. Pass the input lines to the compiler to get a semantic check
     - If this fails, further babelc processing is cancelled
  b. Wash away preprocessor file and line information and add this to the Line instances
  c. Extract the declarations in the original header file (this defines the closure)
  d. Clean up literal strings and characters to make the code easier to parse
     - This includes removing curly braces and making them uppercase
  e. Determine which of the prerequisites (including babelc itself) that has the newest time stamp
  f. Divide the input lines into a map keyed with C++ namespace names
     - This is made to ease the lookup of types
  g. Parse the closure
     - This is done via regular expressions and some simple hand coded recursive descent
- The result of a successful parse is an abstract syntax tree

The first steps are made to clear up the last remnants of prepressor specific output. The modification of character strings and literals are benign since they have no meaning for babelc itself, but it eases the later parsing considerably if you don't need to expect curly braces and C++ keywords anywhere else than within pure code. A typical example is where the parser wishes to skip over a balanced set of curly braces, which is coalesced to a simple character search for [{}] and a counter for the matches.
The closure, which is made up of the one and only complete namespace within the babelc input file, is important because it determines what will be the output. Only types within that namespace and the types referred by them (recursively) participates in the final generation.

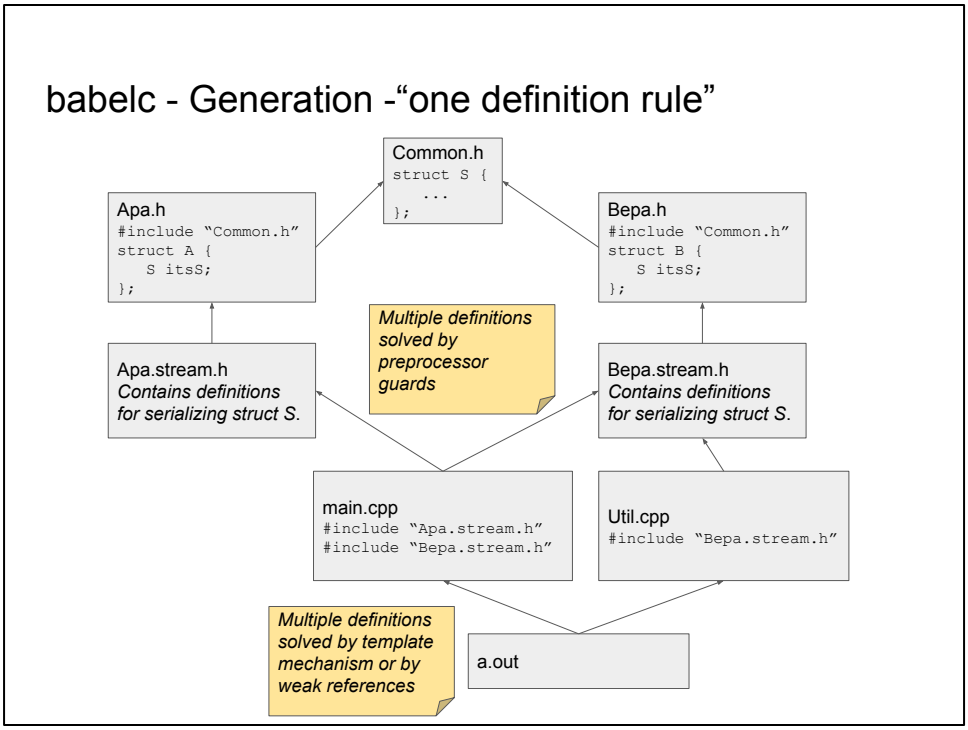


The AST is straightforward. A peculiarity is the BuiltInType since it mixes boolean and numeric types. This will probably change since more needs for a specialization have arised.

babelc - Generation

- Each generator is rather specific, but they share some common strategies:
  a. One input file generates one output target file
     - The input file closure is used to determine what's generated, transitively
     - This makes it easy to create makefiles with "on-the-fly" generation
  b. Output is "header only". C++ "one definition rule" thus forces the use of
     - Templates
     - "weak" definitions
     - `HAS_*` preprocessor guards
  c. The target file is the same as input filename with a "pre-suffix"
     - `Foo.h => Foo.ipc.h`
     - `Bar.h => Bar.stream.h`
  d. The target file is only regenerated iff one or more of these conditions are true:
     - The input file is newer than than the target file
     - An directly or indirectly `#include`d file is newer than the target file
     - The babelc compiler itself is newer than the target file

  Note! It is up to the build system to ensure that the target is regenerated if the C++ compiler, its flags, makefiles, environment etc have changed



One might argue that keeping everythig “header only" will incur long build times. This i a legitimite concern, but one also has to bear in mind that there are not that many compilation units that will need the generated code; most are better off by only using the input header to babelc. This is a good advice anyhow, since the generated code is really an implementation detail.
There may of course be instances where one want a paricular functionalty, say output of a human readable format of a data structure, to be visible in several compilation units. This could be partially handled by precompilation of the header in question, or a hand crafted front end supplies the top level calls and its implementation includes the generated header.

babelc - Unit and module tests

- Some unit tests are included in the binary and are run executed by giving the flag `-babelcUnitTest` when invoking babelc and the result is given by the exit code
- An extensive DBUS IPC and stream test is found in the tests directory
  - The makefile will clone the Google test framework git repo if needed
  - IPC tests are made for every possible IPC type as both return value and parameter via generated test cases from a textual template
  - IPC tests are run both over DBUS and via internal routing

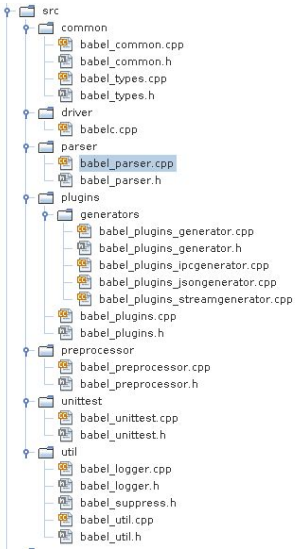
The reason for the somewhat pecuiliar “built in" unit tests are that I didn’t want to export some internal functionalty. It also has the benefit of always keeping them compilable duriing the development cycle.

babelc - DBUS IPC

- The IPC generator is protocol agnostic and instead relies upon a policy
- A simple DBUS policy is provided with babelc
- This policy is at the moment the only documentation of how to write a policy
- Basic design:
  - The policy provides a “message type”
  - Values can be read from/written to a message instance via variadic template functions
  - Message instances can be sent and received via functions and subscriptions
  - Interfaces and methods can be registered via variadic template functions
  - The policy provides functions to drive a protocol main loop

It is not entirely true that the ipc backend is protocol agnostic. Since it was developed with D-Bus in mind there have been certain patterns included which might not have been if the target was not D-Bus. The backend may need some tinkering if a new exotic protocol is needed.

babelc - Source code structure



- About 2500 NCLOCs
- Compiles clean with -Wall and -Wextra + some more
- Extensive unit and module tests
- Note: No memory management - this is a one-pass compiler!

This is of course subject to changes!

babelc wish list

- DBUS properties - a can of worms but used by third-party libraries
- Binary serialization
  - Dead simple compared to dbus :)
- Protobuf
- JSON exists as “write only” today, but “read” would be nice
- ASN.1 replacement?
  - Not bloody likely. A PER backend is horrendously complex