(FYI, sorry for terrible formatting -- docs can’t import reST. Use the github version if it helps.)

Clang C++ Services

This is a design document, and a request for comments on a proposed Clang C++ service model.
The overarching goal is to build support for running a persistent, caching service layer adjacent to a users' editor(s) of choice. This layer would provide much of the functionality that might traditionally be found in an IDE, but designed to work with different, very "unintegrated" editors and the widely used and popular command line development tools on unix-ish operating systems. It would be implemented in terms of Clang/LLVM's libraries to support C/C++/Obj-C/Obj-C++ development. It should be heavily integrated into existing Clang layers such as the Tooling library, libclang, and potentially the plugin architecture.

Concrete Goals and Non-Goals

Goals:

- Provide a restartable, long-lived background process which manages caching, compilation, indexes, and performs the business logic.
- Define an inter-process communication protocol to allow command line tools and libraries to communicate with the service layer.
    - This IPC layer should enable cross-machine usage in theory (so we would like to avoid shared memory), but it's not likely to be implemented in the initial round.
- Take advantage of multiple cores to parallelize tasks across multiple files automatically.
- Support very low-latency queries for UI-interactive modes: code-completion.
    - The crazy stretch goal for this is O(1ms) for code-completion with fully warm and primed caches.
- Provide basic command-line tools for interacting with the service layer via IPC.
- Provide a stable C API (much like libclang) for interacting with the service layer via IPC.
    - Should be a strict subset of the existing libclang API.
    - Clients using only the narrow API should be able to switch trivially between the two libraries to get IPC vs. internal process behavior.
- Provide Python bindings around the stable C API for the IPC layer.
    - Same constraints as the C API w.r.t. existing Python bindings.
- Share all implementation with libclang. These should be two interfaces to the same core functionality.
- Effective interface strategy for generic open source editors.
    - At least VIM and Emacs must be easily supported as 1st class citizens.
    - I'd really like to have a Mac and Windows editor as well.

Non-Goals:

- Support languages outside of those supported by Clang already.

- Support code generation or other uses of Clang. (This might be interesting at some point, but it's not something we want to be constrained by in the initial design.)
- Anything remotely related to actually building an editor or a GUI or an actual IDE. This is a service platform proposal and will not creep toward an IDE or join in the editor wars. ;]

High-Level Architecture

The high-level architecture of the service platform follows a traditional client/server model, with the exception that the client is expected to be able to start an initial server if one is missing, and for the server to (primarily) run on the same machine is the user's editor and any other clients. It is designed to use a socket-like message oriented IPC mechanism for communication between the client and the server.

Protocol Design

The communication protocol will take the form of serialized messages encoded using the LLVM bitcode system. We will define bitcode readers and writers for each message type. We will build an extremely simplified socket-oriented IPC mechanism which handles the following tasks:

- Discovering the existence of a running server for a particular project/user combination.
- Establishing a connection from the client to the server.
    - Writing a request message to the server in serialized form.
    - Wait for a response message to arrive from the server.
    - Read the response message in serialized form.

This is intended to map cleanly onto an implementation using TCP or Unix domain sockets, as well as other socket-like libraries.
The types of requests and response provided by the protocol is sketched here to give a rough idea. The specifics of these will be hashed out as the implementation and specific tools evolve and mature. We expect the requests to be *extremely* high level in nature. Here are some example request/response pairs:

```
Request: {
  Kind: Syntax Check Request
  File: src/foo.cpp
  Dirty Buffers: {
    src/foo.cpp: /tmp/xyz123
    src/foo.h: /tmp/abc123
  }
}
Response: {
  Kind: Syntax Check Response
  File: src/foo.cpp
  Errors: {
    # A Serialized Diagnostic object, or some similar representation.
  }
  Warnings: {
    # ...
  }
}

Request: {
  Kind: Code Complete Request
  Token: <...> # An optional token to resume a previous code-completion
  Location: {
    # Serialized canonical source location
  }
  Dirty Buffers: {
    src/foo.cpp: /tmp/xyz123
    src/foo.h: /tmp/abc123
  }
}
Response: {
  Kind: Code Complete Response
  Token: .... # Arbitrary token used to resume this with a new request
  Completions {
    # A list of code completion suggestions
  }
}
```

Likely one of the more interesting parts of the protocol is the dirty buffers section. The goal is to avoid sending large chunks of data across the IPC mechanism where possible, and instead to allow normal file I/O in most cases. The idea is that the editor can stash copies of dirty edit buffers into temporary storage, potentially in-memory temporary storage, and provide a mapping from the source file to the storage location for the dirty buffer to allow the clang service to act as-if the currently in-progress edits were saved when operating on the file. That said, this explicitly will not preclude future work to extend the dirty buffer system to support patch deltas or complete files in the IPC protocol as necessary to support systems without a sufficiently low-overhead file system or to support cross-machine operation.

Each of these nested groups will be implemented with re-usable serialization and de-serialization logic built on top of the bitcode reader/writer so that we can build up a collection of common message data types that can be quickly combined by clients to form particular protocols.

The intent is that the set of protocol interfaces exposed closely resembles the most high-level of the libclang interfaces. These are necessarily stable, long-lived interfaces, and so they share many design constraints with libclang. There should be close parity in design, structure, and available functionality between the two. It is entirely possible that we will naturally converge on supporting the full width of libclang's API here, but it's not necessary initially.

Clang Server

The primary role of the server is to combine two existing constructs in Clang: the libclang/ASTUnit translation unit caching and management system, and the lib/Tooling tool

running and compilation database management system. It also includes the functionality to listen for incoming requests, and respond to them potentially in parallel.
Several changes are needed to the core Clang libraries to support the server model:

- Support arbitrarily complex file re-mappings, including in the driver and header search logic. This is essential to fully support dirty buffers.
- Thread safety when two threads are concurrently parsing different TUs.
    - One potential requirement will be the ability to share the cached open files between different file managers with different file re-mappings. This will reduce the memory overhead significantly, but introduces synchronization complexity.
- More optional callback instrumentation of the compilation, for example to pause and resume parsing or other operations while communicating with the client.
- Factoring some of the logic currently in libclang to implement high-level operations into C++ APIs that can be shared by libclang and the server.

The Clang server will operate on one or more compilation databases [1] associated with a project. These will be found either by an explicit database file or using a '.clangrc' file [2]. If the RC file specifies multiple compilation databases for a given project, potentially for different build configurations, the server will utilize the union of them. These databases will provide the basis for running tools over source code. The server will expose an interface for directly adding a compilation to the set managed by the server as well in order to match expected libclang functionality. This will also allow IDEs to potentially dynamically update the compilation database as new build information is available.

[1]	TODO: Link to compilation database documentation on clang.llvm.org.

[2]	TODO: Specify the format of this file, and add support for it to the Tooling library as it is generally useful. Current plan is a flat YAML block of key-value mappings. Turn this into a link to that documentation.

The server will also monitor the compilation database and refresh its view if the database file changes underneath it. This will be accomplished through the file system if supported, or through polling and checksumming if not.
The Clang server when started will expose its connection through a file. This will be a platform-specific file allowing a connection to be made. It is expected to Unix Domain Socket on Linux at least, and likely Mac. Windows support mechanism here is TBD. If the server is started around a single compilation database, the connection file will be placed adjacent to it, and called '.clang_server_connection' [3]. If the server is started using an RC file, that RC file will specify a location for the connection file.

[3]	TODO: Is there a better name for this?

If the connection file is relocated or removed at any point, the server is required to detect this eventually and shut down. It is expected that in the event of failure modes multiple servers will be running concurrently for a particular project for a brief period of time. The server should be able to gracefully cope with the this, and thus avoid holding locks on shared files for long periods of time, but use file system locking whenever updating files.

Clang C++ Client Libraries

The client code will have at its core a set of C++ libraries that use the IPC mechanism to communicate with a running server, and include code to automatically launch a server if it is not currently running. The strategy to locate or launch a server follows:

1. Starting from the working directory of the client (or a given directory), walk up from that directory through parent directories looking for one of three files:
    a. A '.clangrc' file [4] which specifies the location of compilation database(s) and (optionally) the means of connecting to a running Clang server, or
    b. A file which allows connecting to the running Clang server, or
    c. A compilation database.
2. If a means of connecting to a Clang server has been established, the client will attempt to connect to the server. If that connection fails for any reason, it will remove or relocate the connection file to allow forward progress and continue as if it had not been found.
3. If no connection is available, the client code will spawn a new server for the given RC file and/or compilation database.

[4]	TODO: Link to .clangrc docs as above.

Once the client has connected, it can send and receive messages to complete whatever tool requests are necessary. The C++ libraries will include convenient wrapper APIs matching the high-level Clang-based tool functionality which manage the IPC necessary to implement them in the context of a client program. These libraries will be as light weight and minimal as possible. Note that these will not represent a stable API in any way. They will follow the same rapid-update development philosophy of Clang and LLVM in general. Stable APIs will always be through an *ABI* safe interface.

Clang C Client Libraries

Wrapping the C++ client libraries will be a API and ABI stable C library. This will very closely resemble (and ideally end up largely source-compatible with) the highest-level libclang APIs. The goal is to evolve them in parallel, and wherever it makes sense, expose similar functionality through both using similar interfaces. Ideally clients can choose whether to link in the functionality or reach out to an external process, while coding against largely similar interfaces. Note however that not all libclang interfaces are suitable to the client/server model. It is not currently expected that the full richness of the cursor interface will make sense in a client/server model.

It is an essential property of the C APIs that they are extremely light weight both in binary size and runtime. These are expected to loaded and unloaded into interactively started editors and other processes. They should form the low-overhead alternative to the heavyweight libclang model.

Clang Client Bindings

In the same way that the C API in libclang has Python bindings, we expect to provide Python bindings for the C client APIs. These bindings will likely be a common means of interfacing with the server due to the ubiquity of Python plugin support in editors. The Python API should expose all the same high-level functionality as the C API does, and it should be significantly cheaper to load and interact with in an editor or plugin context due to the light weight nature of the C client APIs.

It is also likely that several other binding languages will be provided. Due to the inherently high level nature of the interfaces, and the reliance on IPC for all of the implementation details, bindings are expected to be very easy and light weight. These will ease the editor integration process. Beyond the Python bindings which are seen as absolutely necessary, the following bindings would be very useful:

- Emacs Lisp, for Emacs integration obviously.
- Ruby, for RoR and web application integration.
- Lua, often used as a lighter-weight plugin glue language.

Clang Client Commandline Interfaces

The final client interface layer is the CLI. These will take the form of extremely small, focused command line tools that wrap a single tool functionality. They have three primary roles: First, these will form a command line tool set that is the maximally generic and minimal entry-bar point of integration. Anyone with a command line can use them directly in their workflow. Any editor which runs in an environment with a command line can integrate with them.

Without a CLI, we can't support 'ed'. If we can't support 'ed', then someone somewhere will have their personal favorite editor that we don't play nicely with, and that hurts adoption.

The second role is to provide a bridge between the classical and well understood and used Unix development tool set and C++ tools. The CLIs should integrate cleanly and powerfully with tools and scripts that have been built up around the standard tools of find, grep, awk, sed, perl, etc. By integrating cleanly with these other tools, many power use cases which would be hard to express in a single high-level tool interface can be accomplished by composing tools on the command line.

The final role for the CLIs is to provide a basis for testing. These will drive the regression and feature tests, and allow the complete client/server system to be easily exercised in a repeatable fashion.

Implementation Strategy

N.B.: This section is *extremely* rough, much more-so than the rest of the document. Take it with big, big grains of salt.

We're planning on implementing this design with a two-pronged meet-in-the-middle approach for the initial functionality. One side will start at the bottom of the infrastructure stack with Clang:

1. Fix filesystem layer in LLVM & Clang to support proper cross-platform VFS, dirty buffer

interposition even for the Clang driver, the FileManager use cases, etc.
    - A necessary step will be to unify / remove PathV1 and PathV2. This should end the long standing PathV* / FileManager confusion in addition to paving the way for the features needed by a server model.
    - Will also lay ground work for thread safely sharing some file resources, any local filesystem caches, etc.
2. Prepare Clang for concurrent operation.
    - Fix some lingering global variables in the Clang logic.
    - Also Ensuring that none of the thread-hostile LLVM commandline flags are used by normal Clang parsing, even in the presence of inline assembly.
    - Add concurrency primitives in addition to synchronization -- work queue, executors, etc.
    - Write a concurrent testing tool.
3. Connect Tooling layer to concurrency layer, parallelize tools by default.
4. Add necessary RC-file wiring.
5. Connect basic server request system to socket library, demo persistent server running.
6. Implement server logic (request processing, queuing, etc.)

Concurrently, the second path of attack:

1. Build socket-like I/O library, discovery, connection management layer.
    - Will include basic testing utilities, mostly unit tests.
    - Some performance measurement tools possible, measure latency & throughput.
    - Likely only to support Linux and local sockets in the initial implementation.
2. Build framework for message serialization & de-serialization w/ bitcode.
    - Request/response boiler plate.
    - Core message structures (source location, dirty buffers, etc)
    - Clang-check diagnostic message structures
    - Test tools to read and write messages so that they can be round-tripped.
3. Build initial C++ client library, basic functionality
    - Start, stop, discover server
    - Connection opening, request, wait, response reading
    - Hook clang-check up to client library, demo round-trip of clang-check through server.

From this point, we'll start to branch outward feature-wise rather than diving toward each other.