Apache Thrift

A brief introduction

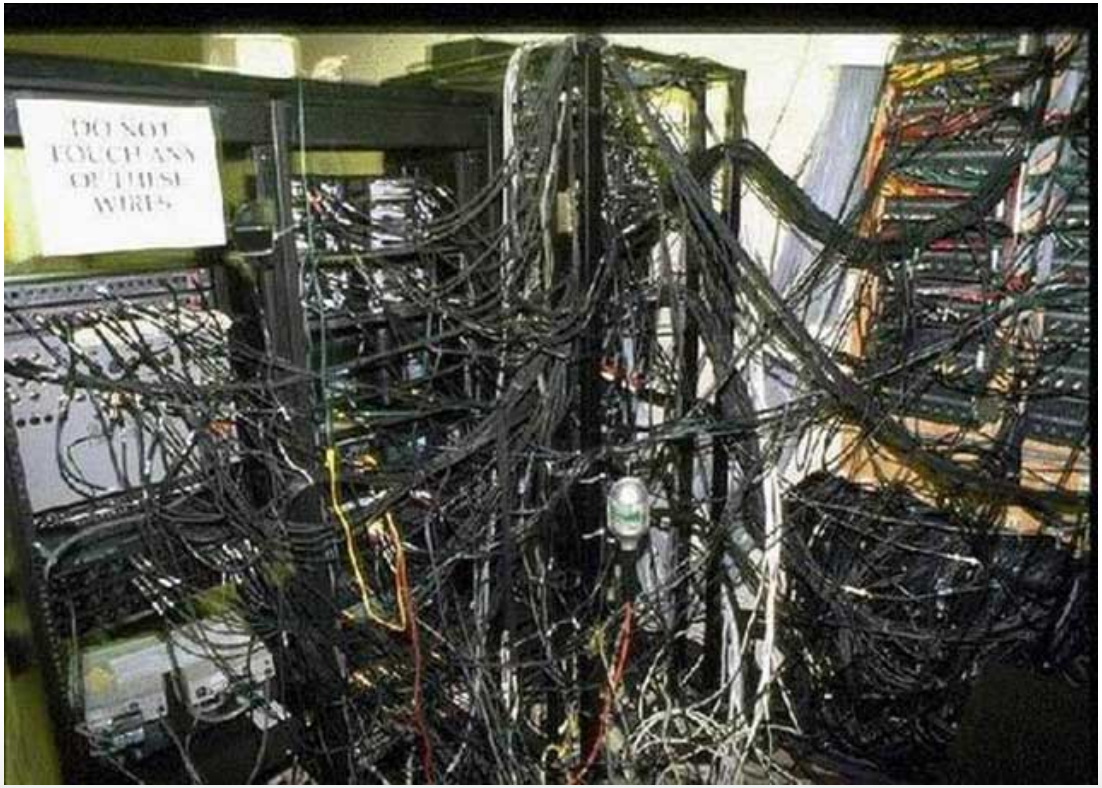
2011 Dvir Volk,
System Architect, DoAT
dvir@doat.com | http://doat.com | @dvirsky

So you want to scale your servers...

- When you grow beyond a simple architecture, you want..
  - redundancy
  - modularity
  - flexibility
  - ability to grow
  - and of course - ***you want it to be simple***

So you end with up with...

Something like this!



Joking aside,
Scalable, modular systems tend to be very complex.

We need a simple way to manage our services.

How components talk

- Database protocols - fine.
- HTTP + maybe JSON/XML on the front - cool.
- But most of the times you have internal APIs.
- HTTP/JSON/XML/Whatever
  - Okay, proven, yada yada
  - But lack protocol description.
  - You have to maintain both client and server code.
  - You still have to write your own wrapper to the protocol.
  - XML has high parsing overhead.

Enter Apache Thrift

- Cross platform, Cross Language, service development framework.
- Supports: C++, Java, Python, PHP, C#, Go, Erlang, JS, Ruby, ObjC, and more...
- Developed internally at Facebook, used there internally.
- An open Apache project.
- Allows you to quickly define your service.
- compiles client and server wrappers for your calls.
- Takes care of everything for you, and makes all the networking, serialization, etc transparent.
- Firing up a server is literally <20 lines of code.
- Example...

Example: MyFirstCppServer

that's 10 lines. HA!
you'll understand everything soon...

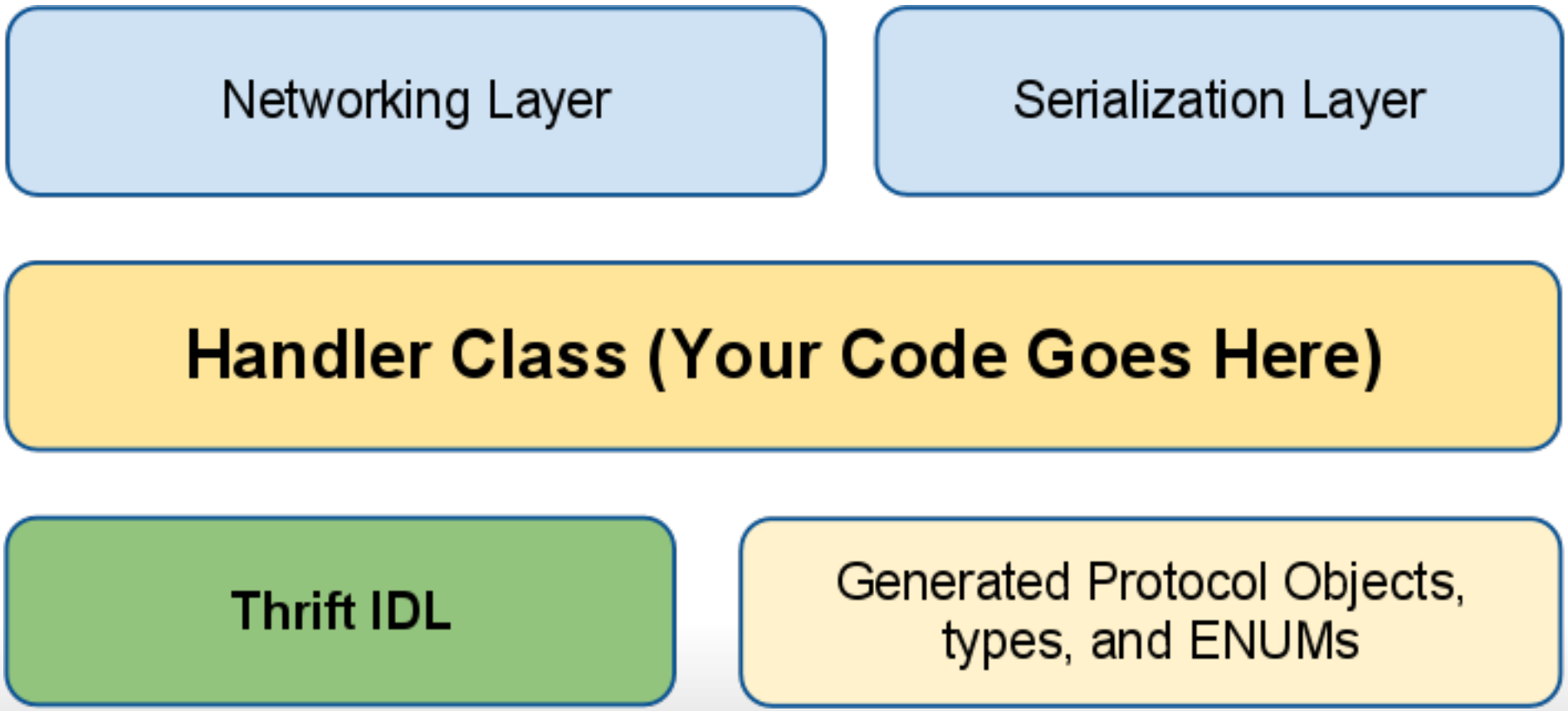
```
int main(int argc, char **argv) {
  int port = 9090;

  shared_ptr<UserStorageHandler> handler(new UserStorageHandler());
  shared_ptr<TProcessor> processor(new UserStorageProcessor(handler));
  shared_ptr<TServerTransport> serverTransport(new TServerSocket(port));
  shared_ptr<TTransportFactory> transportFactory(new
                                        TBufferedTransportFactory());
  shared_ptr<TProtocolFactory> protocolFactory(new TBinaryProtocolFactory());

  TSimpleServer server(processor, serverTransport, transportFactory, protocolFactory);

  server.serve();
  return 0;
}
```

The Anatomy of a Thrift Service



Okay, now from the beginning

- First, define data structures, enums, typedefs, and methods of the service.
- Types: i16, i32, double, string, bool, etc
- Containers: list<>, map<>, set<>
- structs - can contain primitives and other structs
- the numbers are there for a reason!

```
struct Image {
    1: string URL,
    2: i32 size
}

typedef i64 TUSerId

enum UserType { ADMIN = 1, USER = 2 }

struct User {
    1: TUSerId id,
    2: string name,
    3: string password,
    4: Image icon
}
```

Defining your service

- Using the primitives and data structures you've created, you define the service and function calls themselves:

```
service UserAuthenticator {

    User authenticateUser(1: string name, 2: string password),

    Image getUserIcon(1: TUSerId userId),

    bool isValidUser(1: TUSerId userId),

oneway void logoutUser(1: i64 userId)

}
```

Compiling clients and servers

- the thrift executable is a compiler from the weird IDL to any language:
- Example: **thrift --gen cpp MyProject.thrift**
- Most languages compile both client and server at once
- Outputs thousands of lines - but they remain fairly readable in most languages
- Namespaces per language
- Each language in a separate folder
- thrift --gen html => Output service documentation :)
- DO NOT EDIT!

Implementing your handlers

Now all that's left is to take a generated stub and fill the dots. For each call in the service IDL you should have a function in your class.

```
class UserAuthenticator(objcet):

    def authenticateUser(self, name, password):
      pass

    def getUserIcon(userId):
      pass

    def isValidUser(userId):
      pass

    def logoutUser(userId):
    pass
```

Filling the blanks

The structs you defined at your IDL are now classes available to you in your native code.
If a call needs to return a struct, just make the function return it.

```
class UserAuthenticator(objcet):

    def authenticateUser(self, name, password):

        #get a User object
        user = MyDatabase.loadUser(name = name, password = password)

        #with the members you defined...
        user.icon = Image('http://img.example.com/user/%s' % user.id, 'icon')

        #if the protocol demands a struct to be returned
        return user
```

Putting it all together - server side

- Thrift consists of several interchangeable layers: sockets, serializers, servers and processors.

- Choose the best server and serializer for your goal/lang:
  - blocking/non-blocking
  - SSL available for some languages
  - compression available
  - JSON for JS clients

- Some dependencies between layers exist.
- Add your own class to the mix.
- you're good to go!

That server example again...

```
//this is your own handler class...
shared_ptr<UserStorageHandler> handler(new UserStorageHandler());

//the processor is what calls the functions in your handler
shared_ptr<TProcessor> processor(new UserStorageProcessor(handler));

//the transport layer handles the networking
//it consists of a socket + transport
shared_ptr<TServerTransport> serverTransport(new TServerSocket(port));
shared_ptr<TTransportFactory> transportFactory(new
                                    TBufferedTransportFactory());

//the "protocol" handles serialization
shared_ptr<TProtocolFactory> protocolFactory(new TBinaryProtocolFactory());

//one server to rule them all, and in the service bind them
TSimpleServer server(processor, serverTransport, transportFactory,
                protocolFactory);

//TADA!
server.serve();
```

Calling client methods

Initialize a client, call the same methods in the same way.

```
# Create a transport and a protocol, like in the server
transport = TSocket.TSocket("localhost", 9090)
transport.open()

protocol = TBinaryProtocol.TBinaryProtocol(transport)


# Use the service we've already defined
authClient = UserAuthenticator.Client(protocol)

#now just call the server methods transparently
user = authClient.authenticateUser('dvirsky', '123456')
```

Different types of servers

TSimpleServer	Single threaded, mostly useful for debugging.
TThreadedServer	Spawns a thread per request, if you're into that sorta thing.
TThreadPoolServer	N worker threads, but connections block the threads.
TNonBlockingServer	Optimal in Java, C++, less so in other languages.
THttpServer	HTTP Server (for JS clients) optionally with REST-like URLs
TForkingServer	Forks a process for each request
TProcessPoolServer	Python - By Yours truly. Pre-forks workers to avoid GIL.

Gotchas

- IDL Limits:
  - No circular references
  - no returning NULLs
  - no inheritance
- No out-of-the-box authentication.
- No bi-directional messaging.
- In thread-pool mode, you are limited to N connections
- make your workers either very fast, or async, to avoid choking the server.
- In python, GIL problem means thread based servers suck.
- Make sure you get the right combination of transports on client and server.
- Make sure to use binary serializers when possible.

A Few Alternatives

Protocol Buffers
Developed by Google. Similar syntax. No networking stack.

Avro
Also an Apache project, only 4 languages supported

MessagePack
Richer networking API. New project. Worth checking!

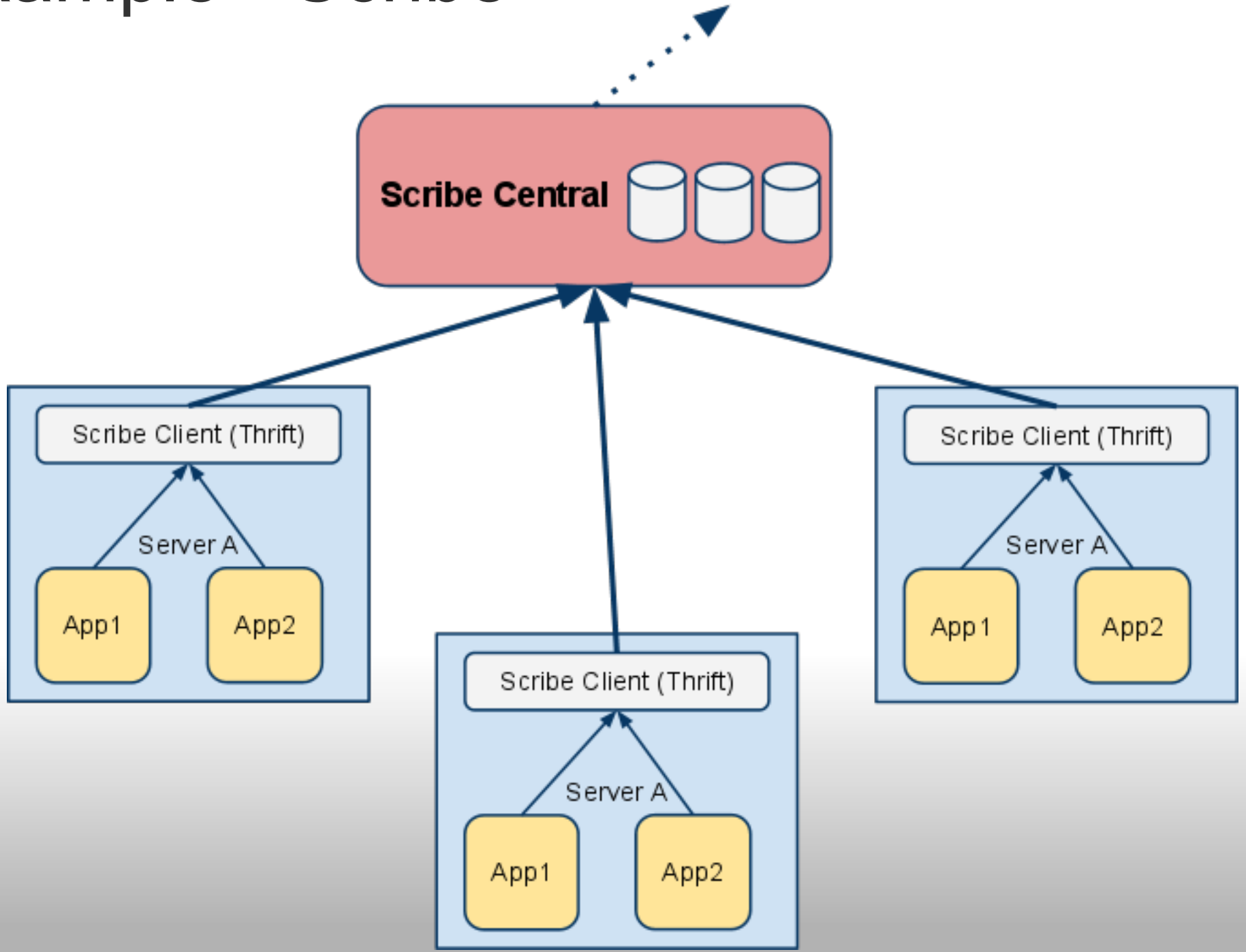
HTTP + JSON / XML / WHATEVER
No validation, no abstraction of calls unless you use SOAP or something similar.

Projects Using Thrift

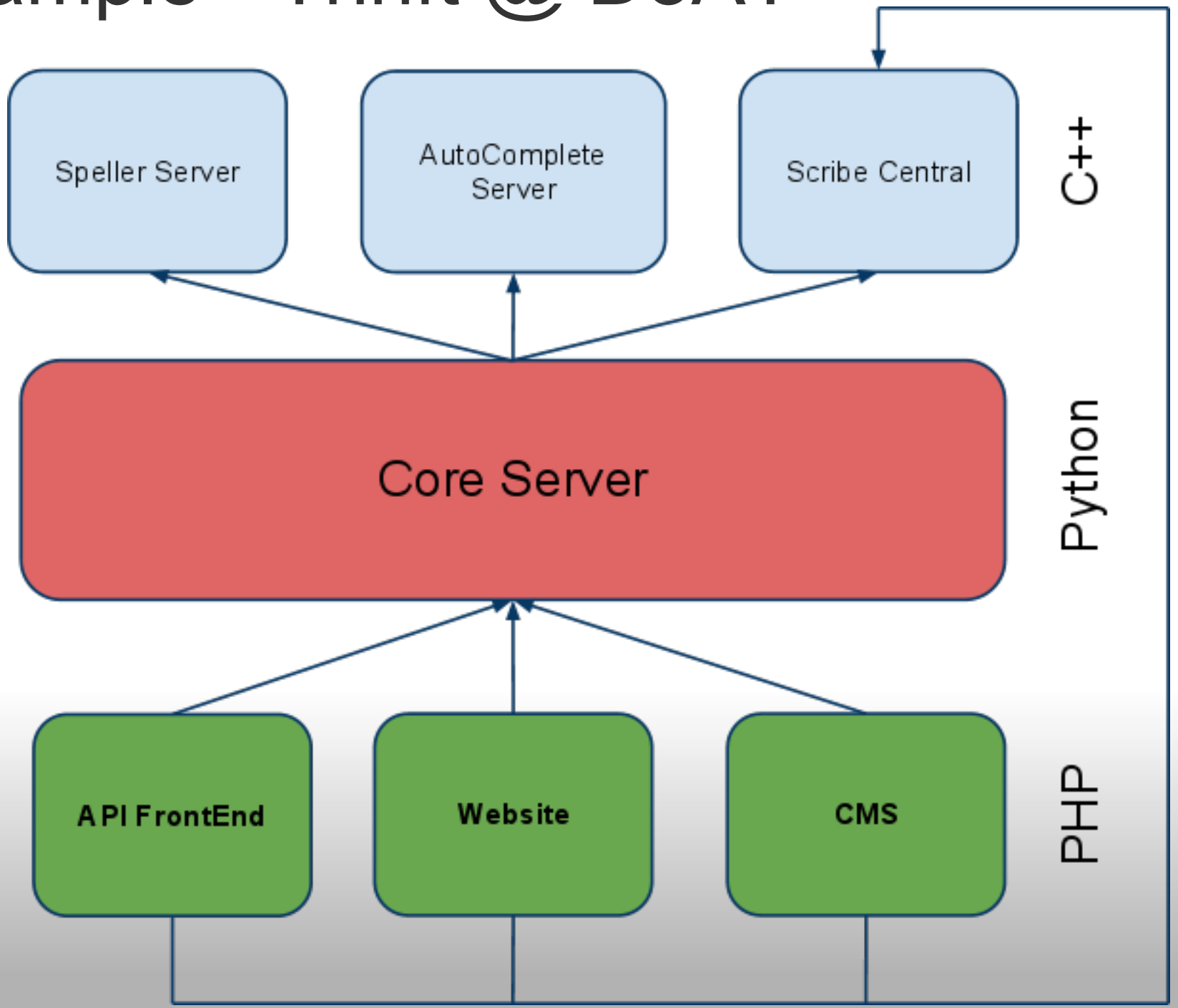
- Cassandra
- ThriftDB
- Scribe
- Hadoop / HBase
- Scribe

- Facebook
- LastFM
- DoAT :)

Example - Scribe



Example - Thrift @ DoAT



Thank you!

for more info:
http://thrift.apache.org/

Follow @DoatGeeks on twitter!