Thrift: Scalable Cross-Language Services Implementation

Mark Slee, Aditya Agarwal and Marc Kwiatkowski

Facebook, 156 University Ave, Palo Alto, CA

{mcslee,aditya,marc}@facebook.com

Abstract

Thrift is a software library and set of code-generation tools developed at Facebook to expedite development and implementation of efficient and scalable backend services. Its primary goal is to enable efficient and reliable communication across programming languages by abstracting the portions of each language that tend to require the most customization into a common library that is implemented in each language. Specifically, Thrift allows developers to define data types and service interfaces in a single language-neutral file and generate all the necessary code to build RPC clients and servers.

This paper details the motivations and design choices we made in Thrift, as well as some of the more interesting implementation details. It is not intended to be taken as research, but rather it is an exposition on what we did and why.

1. Introduction

As Facebook's traffic and network structure have scaled, the resource demands of many operations on the site (i.e. search, ad selection and delivery, event logging) have presented technical requirements drastically outside the scope of the LAMP framework. In our implementation of these services, various programming languages have been selected to optimize for the right combination of performance, ease and speed of development, availability of existing libraries, etc. By and large, Facebook's engineering culture has tended towards choosing the best tools and implementations avaiable over standardizing on any one programming language and begrudgingly accepting its inherent limitations.

Given this design choice, we were presented with the challenge of building a transparent, high-performance bridge across many programming languages. We found that most available solutions were either too limited, did not offer sufficient data type freedom, or suffered from subpar performance. [1]

The solution that we have implemented combines a language-neutral software stack implemented across numerous programming languages and an associated code generation engine that transforms a simple interface and data definition language into client and server remote procedure call libraries. Choosing static code generation over a dynamic system allows us to create validated code with implicit guarantees that can be run without the need for any advanced intropsecive run-time type checking. It is also designed to be as simple as possible for the developer, who can typically define all the necessary data structures and interfaces for a complex service in a single short file.

Surprised that a robust open solution to these relatively common problems did not yet exist, we committed early on to making the Thrift implementation open source.

[1] See Appendix A for a discussion of alternative systems.

In evaluating the challenges of cross-language interaction in a networked environment, some key components were identified:

Types. A common type system must exist across programming languages without requiring that the application developer use custom Thrift data types or write their own serialization code. That is, a C++ programmer should be able to transparently exchange a strongly typed STL map for a dynamic Python dictionary. Neither programmer should be forced to write any code below the application layer to achieve this. Section 2 details the Thrift type system.

Transport. Each language must have a common interface to bidirectional raw data transport. The specifics of how a given transport is implemented should not matter to the service developer. The same application code should be able to run against TCP stream sockets, raw data in memory, or files on disk. Section 3 details the Thrift Transport layer.

Protocol. Data types must have some way of using the Transport layer to encode and decode themselves. Again, the application developer need not be concerned by this layer. Whether the service uses an XML or binary protocol is immaterial to the application code. All that matters is that the data can be read and written in a consistent, deterministic matter. Section 4 details the Thrift Protocol layer.

Versioning. For robust services, the involved data types must provide a mechanism for versioning themselves. Specifically, it should be possible to add or remove fields in an object or alter the argument list of a function without any interruption in service (or, worse yet, nasty segmentation faults). Section 5 details Thrift's versioning system.

Processors. Finally, we generate code capable of processing data streams to accomplish remote procedure calls. Section 6 details the generated code and TProcessor paradigm.

Section 7 discusses implementation details, and Section 8 describes our conclusions.

2. Types

The goal of the Thrift type system is to enable programmers to develop using completely natively defined types, no matter what programming language they use. By design, the Thrift type system does not introduce any special dynamic types or wrapper objects. It also does not require that the developer write any code for object serialization or transport. The Thrift IDL file is logically a way for developers to annotate their data structures with the minimal amount of extra information necessary to tell a code generator how to safely transport the objects across languages.

2.1 Base Types

The type system rests upon a few base types. In considering which types to support, we aimed for clarity and simplicity over abundance, focusing on the key types available in all programming languages, ommitting any niche types available only in specific languages.

The base types supported by Thrift are:

- `bool` A boolean value, true or false
- `byte` A signed byte
- `i16` A 16-bit signed integer
- `i32` A 32-bit signed integer
- `i64` A 64-bit signed integer
- `double` A 64-bit floating point number
- `string` An encoding-agnostic text or binary string

Of particular note is the absence of unsigned integer types. Because these types have no direct translation to native primitive types in many languages, the advantages they afford are lost. Further, there is no way to prevent the application developer in a language like Python from assigning a negative value to an integer variable, leading to unpredictable behavior. From a design standpoint, we observed that unsigned integers were very rarely, if ever, used for arithmetic purposes, but in practice were much more often used as keys or identifiers. In this case, the sign is irrelevant. Signed integers serve this same purpose and can be safely cast to their unsigned counterparts (most commonly in C++) when absolutely necessary.

2.2 Containers

Thrift containers are strongly typed containers that map to the most commonly used containers in common programming languages. They are annotated using C++ template (or Java Generics) style. There are three types available:

- `list<type>` An ordered list of elements. Translates directly into an STL vector, Java ArrayList, or native array in scripting languages. May contain duplicates.
- `set<type>` An unordered set of unique elements. Translates into an STL set, Java HashSet, or native dictionary in PHP/Python/Ruby. (AdityaCheck: is PHP dictionary the right terminology for what we have in PHP?)
- `map<type1,type2>` A map of strictly unique keys to values Translates into an STL map, Java HashMap, PHP associative array, or Python/Ruby dictionary.

While defaults are provided, the type mappings are not explicitly fixed. Custom code generator directives have been added to substitute custom types in destination languages (i.e. `hash_map` or Google's sparse hash map can be used in C++). The only requirement is that the custom types support all the necessary iteration primitives. Container elements may be of any valid Thrift type, including other containers or structs.

2.3 Structs

A Thrift struct defines a common object to be used across languages. A struct is essentially equivalent to a class in object oriented programming languages. A struct has a set of strongly typed fields, each with a unique name identifier. The basic syntax for defining a Thrift struct looks very similar to a C struct definition. Fields may be annotated with an integer field identifier (unique to the scope of that struct) and optional default values. Field identifiers will be automatically assigned if omitted, though they are strongly encouraged for versioning reasons discussed later.

```
struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

In the target language, each definition generates a type with two methods, `read` and `write`, which perform serialization and transport of the objects using a Thrift TProtocol object.

2.4 Exceptions

Exceptions are syntactically and functionally equivalent to structs except that they are declared using the `exception` keyword instead of the `struct` keyword.

The generated objects inherit from an exception base class as appropriate in each target programming language, the goal being to offer seamless integration with native exception handling for the developer in any given language. Again, the design emphasis is on making the code familiar to the application developer.

2.5 Services

Services are defined using Thrift types. Definition of a service is semantically equivalent to defining a pure virtual interface in object oriented programming. The Thrift compiler generates fully functional client and server stubs that implement the interface. Services are defined as follows:

```
service <name> {
  <returntype> <name>(<arguments>)
    [throws (<exceptions>)]
  ...
}
```

An example:

```
service StringCache {
  void set(1:i32 key, 2:string value),
  string get(1:i32 key) throws (1:KeyNotFound knf),
  void delete(1:i32 key)
}
```

Note that `void` is a valid type for a function return, in addition to all other defined Thrift types. Additionally, an `async` modifier keyword may be added to a void function, which will generate code that does not wait for a response from the server. Note that a pure `void` function will return a response to the client which guarantees that the operation has completed on the server side. With `async` method calls the client can only be guaranteed that the request succeeded at the transport layer. (In many transport scenarios this is inherently unreliable due to the Byzantine Generals' Problem. Therefore, application developers should take care only to use the async optimization in cases where dopped method calls are acceptable or the transport is known to be reliable.)

Also of note is the fact that argument and exception lists to functions are implemented as Thrift structs. They are identical in both notation and behavior.

3. Transport

The transport layer is used by the generated code to facilitate data transfer.

3.1 Interface

A key design choice in the implementation of Thrift was to abstract the transport layer from the code generation layer. Though Thrift is typically used on top of the TCP/IP stack with streaming sockets as the base layer of communication, there was no compelling reason to build that constraint into the system. The performance tradeoff incurred by an abstracted I/O layer (roughly one virtual method lookup / function call per operation) was immaterial compared to the cost of actual I/O operations (typically invoking system calls).

Fundamentally, generated Thrift code only needs to know how to read and write data. Where the data is going is irrelevant, it may be a socket, a segment of shared memory, or a file on the local disk. The Thrift transport interface supports the following methods.

- `open()` Opens the tranpsort
- `close()` Closes the tranport
- `isOpen()` Whether the transport is open
- `read()` Reads from the transport
- `write()` Writes to the transport
- `flush()` Force any pending writes

There are a few additional methods not documented here which are used to aid in batching reads and optionally signaling completion of reading or writing chunks of data by the generated code.

In addition to the above `TTransport` interface, there is a `TServerTransport` interface used to accept or create primitive transport objects. Its interface is as follows:

- `open()` Opens the tranpsort
- `listen()` Begins listening for connections
- `accept()` Returns a new client transport
- `close()` Closes the transport

3.2 Implementation

The transport interface is designed for simple implementation in any programming language. New transport mechanisms can be easily defined as needed by application developers.

3.2.1 TSocket

The `TSocket` class is implemented across all target languages. It provides a common, simple interface to a TCP/IP stream socket.

3.2.2 TFileTransport

The `TFileTransport` is an abstraction of an on-disk file to a data stream. It can be used to write out a set of incoming thrift request to a file on disk. The on-disk data can then be replayed from the log, either for post-processing or for recreation and simulation of past events. (TFileTransport).

3.2.3 Utilities

The Transport interface is designed to support easy extension using common OOP techniques such as composition. Some simple utilites include the `TBufferedTransport`, which buffers writes and reads on an underlying transport, the `TFramedTransport`, which transmits data with frame size headers for chunking optimzation or nonblocking operation, and the `TMemoryBuffer`, which allows reading and writing directly from heap or stack memory owned by the process.

4. Protocol

A second major abstraction in Thrift is the separation of data structure from transport representation. Thrift enforces a certain messaging structure when transporting data, but it is agnostic to the protocol encoding in use. That is, it does not matter whether data is encoded in XML, human-readable ASCII, or a dense binary format, so long as the data supports a fixed set of operations that allow generated code to deterministically read and write.

4.1 Interface

The Thrift Protocol interface is very straightforward. It fundamentally supports two things: 1) bidirectional sequenced messaging, and 2) encoding of base types, containers, and structs.

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeFieldStop()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
writeListBegin(etype, size)
writeListEnd()
writeSetBegin(etype, size)
writeSetEnd()
writeBool(bool)
writeByte(byte)
writeI16(i16)
writeI32(i32)
writeI64(i64)
writeDouble(double)
writeString(string)

name, type, seq = readMessageBegin()
                  readMessageEnd()
name =            readStructBegin()
                  readStructEnd()
name, type, id =  readFieldBegin()
                  readFieldEnd()
k, v, size =      readMapBegin()
                  readMapEnd()
etype, size =     readListBegin()
                  readListEnd()
etype, size =     readSetBegin()
                  readSetEnd()
bool =            readBool()
byte =            readByte()
i16 =             readI16()
i32 =             readI32()
i64 =             readI64()
double =          readDouble()
string =          readString()
```

Note that every write function has exactly one read function counterpart, with the exception of the `writeFieldStop()` method. This is a special method that signals the end of a struct. The procedure for reading a struct is to `readFieldBegin()` until the stop field is encountered, and to then `readStructEnd()`. The generated code relies upon this structure to ensure that everything written by a protocol encoder can be read by a matching protocol decoder. Further note that this set of functions is by design more robust than necessary. For example, `writeStructEnd()` is not strictly necessary, as the end of a struct may be implied by the stop field. This method is a convenience for verbose protocols where it is cleaner to separate these calls (i.e. a closing `</struct>` tag in XML).

4.2 Structure

Thrift structures are designed to support encoding into a streaming protocol. That is, the implementation should never need to frame or compute the entire data length of a structure prior to encoding it. This is critical to performance in many scenarios. Consider a long list of relatively large strings. If the protocol interface required reading or writing a list as an atomic operation, then the implementation would require a linear pass over the entire list before encoding any data. However, if the list can be written as iteration is performed, the corresponding read may begin in parallel, theoretically offering an end-to-end speedup of $(kN - C)$, where N is the size of the list, k the cost factor associated with serializing a single element, and C is fixed offset for the delay between data being written and becoming available to read.

Similarly, structs do not encode their data lengths a priori. Instead, they are encoded as a sequence of fields, with each field having a type specifier and a unique field identifier. Note that the inclusion of type specifiers enables the protocol to be safely parsed and decoded without any generated code or access to the original IDL file. Structs are terminated by a field header with a special `STOP` type. Because all the basic types can be read deterministically, all structs (including those with nested structs) can be read deterministically. The Thrift protocol is self-delimiting without any framing and regardless of the encoding format.

In situations where streaming is unnecessary or framing is advantageous, it can be very simply added into the transport layer, using the `TFramedTransport` abstraction.

4.3 Implementation

Facebook has implemented and deployed a space-efficient binary protocol which is used by most backend services. Essentially, it writes all data in a flat binary format. Integer types are converted to network byte order, strings are prepended with their byte length, and all message and field headers are written using the primitive integer serialization constructs. String names for fields are omitted - when using generated code, field identifiers are sufficient.

We decided against some extreme storage optimizations (i.e. packing small integers into ASCII or using a 7-bit continuation format) for the sake of simplicity and clarity in the code. These alterations can easily be made if and when we encounter a performance critical use case that demands them.

5. Versioning

Thrift is robust in the face of versioning and data definition changes. This is critical to enable a staged rollout of changes to deployed services. The system must be able to support reading of old data from logfiles, as well as requests from out of date clients to new servers, or vice versa.

5.1 Field Identifiers

Versioning in Thrift is implemented via field identifiers. The field header for every member of a struct in Thrift is encoded with a unique field identifier. The combination of this field identifier and its type specifier is used to uniquely identify the field. The Thrift definition language supports automatic assignment of field identifiers, but it is good programming practice to always explicitly specify field identifiers. Identifiers are specified as follows:

```
struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

To avoid conflicts, fields with omitted identifiers are automatically assigned decrementing from -1, and the language only supports the manual assignment of positive identifiers.

When data is being deserialized, the generated code can use these identifiers to properly identify the field and determine whether it aligns with a field in its definition file. If a field identifier is not recognized, the generated code can use the type specifier to skip the unknown field without any error. Again, this is possible due to the fact that all data types are self delimiting.

Field identifiers can (and should) also be specified in function argument lists. In fact, argument lists are not only represented as structs on the backend, but actually share the same code in the compiler frontend. This allows for version-safe modification of method parameters

```
service StringCache {
  void set(1:i32 key, 2:string value),
  string get(1:i32 key) throws (1:KeyNotFound knf),
  void delete(1:i32 key)
}
```

The syntax for specifying field identifiers was chosen to echo their structure. Structs can be thought of as a dictionary where the identifiers are keys, and the values are strongly typed, named fields.

Field identifiers internally use the `i16` Thrift type. Note, however, that the `TProtocol` abstraction may encode identifiers in any format.

5.2 Isset

When an unexpected field is encountered, it can be safely ignored and discarded. When an expected field is not found, there must be some way to signal to the developer that it was not present. This is implemented via an inner `isset` structure inside the defined objects. (In PHP, this is implicit with a `null` value, or `None` in Python and `nil` in Ruby.) Essentially, the inner `isset` object of each Thrift struct contains a boolean value for each field which denotes whether or not that field is present in the struct. When a reader receives a struct, it should check for a field being set before operating directly on it.

```
class Example {
 public:
  Example() :
    number(10),
    bigNumber(0),
    decimals(0),
    name("thrifty") {}

  int32_t number;
  int64_t bigNumber;
  double decimals;
  std::string name;

  struct __isset {
    __isset() :
      number(false),
      bigNumber(false),
      decimals(false),
      name(false) {}
    bool number;
    bool bigNumber;
    bool decimals;
    bool name;
  } __isset;
...
}
```

5.3 Case Analysis

There are four cases in which version mismatches may occur.

1. *Added field, old client, new server.* In this case, the old client does not send the new field. The new server recognizes that the field is not set, and implements default behavior for out of date requests.
2. *Removed field, old client, new server.* In this case, the old client sends the removed field. The new server simply ignores it.
3. *Added field, new client, old server.* The new client sends a field that the old server does not recognize. The old server simply ignores it and processes as normal.
4. *Removed field, new client, old server.* This is the most dangerous case, as the old server is unlikely to have suitable default behavior implemented for the missing field. It is recommended that in this situation the new server be rolled out prior to the new clients.

5.4 Protocol/Transport Versioning

The `TProtocol` abstractions are also designed to give protocol implementations the freedom to version themselves in whatever manner they see fit. Specifically, any protocol implementation is free to send whatever it likes in the `writeMessageBegin()` call. It is entirely up to the implementor how to handle versioning at the protocol level. The key point is that protocol encoding changes are safely isolated from interface definition version changes.

Note that the exact same is true of the `TTransport` interface. For example, if we wished to add some new checksumming or error detection to the `TFileTransport`, we could simply add a version header into the data it writes to the file in such a way that it would still accept old logfiles without the given header.

6. RPC Implementation

6.1 TProcessor

The last core interface in the Thrift design is the `TProcessor`, perhaps the most simple of the constructs. The interface is as follows:

```
interface TProcessor {
  bool process(TProtocol in, TProtocol out)
    throws TException
}
```

The key design idea here is that the complex systems we build can fundamentally be broken down into agents or services that operate on inputs and outputs. In most cases, there is actually just one input and output (an RPC client) that needs handling.

6.2 Generated Code

When a service is defined, we generate a `TProcessor` instance capable of handling RPC requests to that service, using a few helpers. The fundamental structure (illustrated in pseudo-C++) is as follows:

```
Service.thrift
 => Service.cpp
     interface ServiceIf
     class ServiceClient : virtual ServiceIf
       TProtocol in
       TProtocol out
     class ServiceProcessor : TProcessor
       ServiceIf handler

ServiceHandler.cpp
 class ServiceHandler : virtual ServiceIf

TServer.cpp
 TServer(TProcessor processor,
         TServerTransport transport,
         TTransportFactory tfactory,
         TProtocolFactory pfactory)
 serve()
```

From the thrift definition file, we generate the virtual service interface. A client class is generated, which implements the interface and uses two `TProtocol` instances to perform the I/O operations. The generated processor implements the `TProcessor` interface. The generated code has all the logic to handle RPC invocations via the `process()` call, and takes as a parameter an instance of the service interface, implemented by the application developer.

The user provides an implementation of the application interface in their own, non-generated source file.

6.3 TServer

Finally, the Thrift core libraries provide a `TServer` abstraction. The `TServer` object generally works as follows.

- Use the `TServerTransport` to get a `TTransport`
- Use the `TTransportFactory` to optionally convert the primitive transport into a suitable application transport (typically the `TBufferedTransportFactory` is used here)
- Use the `TProtocolFactory` to create an input and output protocol for the `TTransport`
- Invoke the `process()` method of the `TProcessor` object

The layers are appropriately separated such that the server code needs to know nothing about any of the transports, encodings, or applications in play. The server encapsulates the logic around connection handling, threading, etc. while the processor deals with RPC. The only code written by the application developer lives in the definitional thrift file and the interface implementation.

Facebook has deployed multiple `TServer` implementations, including the single-threaded `TSimpleServer`, thread-per-connection `TThreadedServer`, and thread-pooling `TThreadPoolServer`.

The `TProcessor` interface is very general by design. There is no requirement that a `TServer` take a generated `TProcessor` object. Thrift allows the application developer to easily write any type of server that operates on `TProtocol` objects (for instance, a server could simply stream a certain type of object without any actual RPC method invocation).

7. Implementation Details

7.1 Target Languages

Thrift currently supports five target languages: C++, Java, Python, Ruby, and PHP. At Facebook, we have deployed servers predominantly in C++, Java, and Python. Thrift services implemented in PHP have also been embedded into the Apache web server, providing transparent backend access to many of our frontend constructs using a `THttpClient` implementation of the `TTransport` interface.

Though Thrift was explicitly designed to be much more efficient and robust than typical web technologies, as we were designing our XML-based REST web services API we noticed that Thrift could be easily used to define our service interface. Though we do not currently employ SOAP envelopes (in the author's opinion there is already far too much repetetive enterprise Java software to do that sort of thing), we were able to quickly extend Thrift to generate XML Schema Definition files for our service, as well as a framework for versioning different implementations of our web service. Though public web services are admittedly tangential to Thrift's core use case and design, Thrift facilitated rapid iteration and affords us the ability to quickly migrate our entire XML-based web service onto a higher performance system should the future need arise.

7.2 Generated Structs

We made a conscious decision to make our generated structs as transparent as possible. All fields are publicly accessible; there are no `set()` and `get()` methods. Similarly, use of the `isset` object is not enforced. We do not include any `FieldNotSetException` construct. Developers have the option to use these fields to write more robust code, but the system is robust to the developer ignoring the `isset` construct entirely and will provide suitable default behavior in all cases.

The reason for this choice was for ease of application development. Our stated goal is not to make developers learn a rich new library in their language of choice, but rather to generate code that allow them to work with the constructs that are most familiar in each language.

We also made the `read()` and `write()` methods of the generated objects public members so that the objects can be used outside of the context of RPC clients and servers. Thrift is a useful tool simply for generating objects that are easily serializable across programming languages.

7.3 RPC Method Identification

Method calls in RPC are implemented by sending the method name as a string. One issue with this approach is that longer method names require more bandwidth. We experimented with using fixed-size hashes to identify methods, but in the end concluded that the savings were not worth the headaches incurred. Reliably dealing with conflicts across versions of an interface definition file is impossible without a meta-storage system (i.e. to generate non-conflicting hashes for the current version of a file, we would have to know about all conflicts that ever existed in any previous version of the file).

We wanted to avoid too many unnecessary string comparisons upon method invocation. To deal with this, we generate maps from strings to function pointers, so that invocation is effectively accomplished via a constant-time hash lookup in the common case. This requires the use of a couple interesting code constructs. Because Java does not have function pointers, process functions are all private member classes implementing a common interface.

```
private class ping implements ProcessFunction {
  public void process(int seqid,
                      TProtocol iprot,
                      TProtocol oprot)
    throws TException
  { ...}
}

HashMap<String,ProcessFunction> processMap_ =
  new HashMap<String,ProcessFunction>();
```

In C++, we use a relatively esoteric language construct: member function pointers.

```
std::map<std::string,
  void (ExampleServiceProcessor::*)(int32_t,
  facebook::thrift::protocol::TProtocol*,
  facebook::thrift::protocol::TProtocol*)>
 processMap_;
```

Using these techniques, the cost of string processing is minimized, and we reap the benefit of being able to easily debug corrupt or misunderstood data by looking for string contents.

7.4 Servers and Multithreading

MARC TO WRITE THIS SECTION ON THE C++ concurrency PACKAGE AND BASIC TThreadPoolServer PERFORMANCE ETC. (ie. 140K req/second, that kind of thing)

7.5 Nonblocking Operation

Though the Thrift transport interfaces map more directly to a blocking I/O model, we have implemented a high performance `TNonBlockingServer` in C++ based upon `libevent` and the `TFramedTransport`. We implemented this by moving all I/O into one tight event loop using a state machine. Essentially, the event loop reads framed requests into `TMemoryBuffer` objects. Once entire requests are ready, they are dispatched to the `TProcessor` object which can read directly from the data in memory.

7.6 Compiler

The Thrift compiler is implemented in C++ using standard lex/yacc style tokenization and parsing. Though it could have been implemented with fewer lines of code in another language (i.e. Python/PLY or ocamlyacc), using C++ forces explicit definition of the language constructs. Strongly typing the parse tree elements (debatably) makes the code more approachable for new developers.

Code generation is done using two passes. The first pass looks only for include files and type definitions. Type definitions are not checked during this phase, since they may depend upon include files. All included files are sequentially scanned in a first pass. Once the include tree has been resolved, a second pass is taken over all files which inserts type definitions into the parse tree and raises an error on any undefined types. The program is then generated against the parse tree.

Due to inherent complexities and potential for circular dependencies, we explicitly disallow forward declaration. Two Thrift structs cannot each contain an instance of the other. (Since we do not allow `null` struct instances in the generated C++ code, this would actually be impossible.)

7.7 TFileTransport

The `TFileTransport` logs thrift requests/structs by framing incoming data with its length and writing it to disk. Using a framed on-disk format allows for better error checking and helps with processing a finite number of discrete events. The file is split up into chunks of a speficified size and logged messages are not allowed to cross chunk boundaries. A message that would cross a chunk boundary will cause padding to be added until the end of the chunk and the first byte of the message is aligned to the beginning of the new chunk.

Partitioning the file into chunks makes it possible to read and interpret data from a particular point in the file. The `TFileWriterTransport` uses a system of swapping in-memory buffers to ensure good performance while logging large amounts of data.

8. Conclusions

Thrift has enabled Facebook to build scalable backend services efficiently by enabling engineers to divide and conquer. Application developers can focus upon application code without worrying about the sockets layer. We avoid duplicated work by writing buffering and I/O logic in one place, rather than interspersing it in each application.

Thrift has been employed in a wide variety of applications at Facebook, including search, logging, mobile, ads, and platform. We have found that the marginal performance cost incurred by an extra layer of software abstraction is eclipsed by the gains in developer efficiency and systems reliability.

A. Similar Systems

The following are software systems similar to Thrift. Each is (very!) briefly described:

- *SOAP.* XML-based. Designed for web services via HTTP, excessive XML parsing overhead.
- *CORBA.* Relatively comprehensive, debatably overdesigned and heavyweight. Comparably cumbersome software installation.
- *COM.* Embraced mainly in Windows client softare. Not an entirely open solution.
- *Pillar.* Lightweight and high-performance, but missing versioning and abstraction.
- *Protocol Buffers.* Closed-source, owned by Google. Described in Sawzall paper.

Acknowledgments

Many thanks for feedback on Thrift (and extreme trial by fire) are due to Martin Smith, Karl Voskuil and Yishan Wong.

Thrift is a successor to Pillar, a similar system developed by Adam D'Angelo, first while at Caltech and continued later at Facebook. Thrift simply would not have happened without Adam's insights.