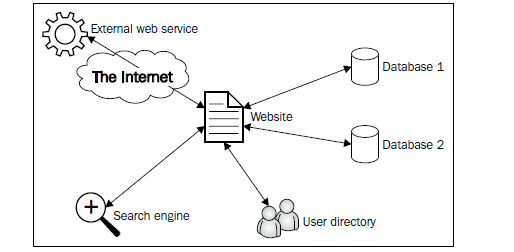
**Thrift Lecture**

**Distributed systems and their services**

Imagine typical web applications that you use every day, such as search engines, messaging platforms, or social networks. Under one web address, they deliver different services.

For example, a social network delivers people search, messaging, and users' profile pages. While you access them by one user interface—a web page written in HTML and JavaScript—what you see in your browser is only a gateway.

Your request to message a friend is being relayed by the underlying application to the messaging service—an application which is specifically designed to deal with exchange of messages between the social network's users.



**Service-Oriented Architecture**

Messaging service, which we use as an example here, may be written in a completely different programming language than web application. It is a design decision.

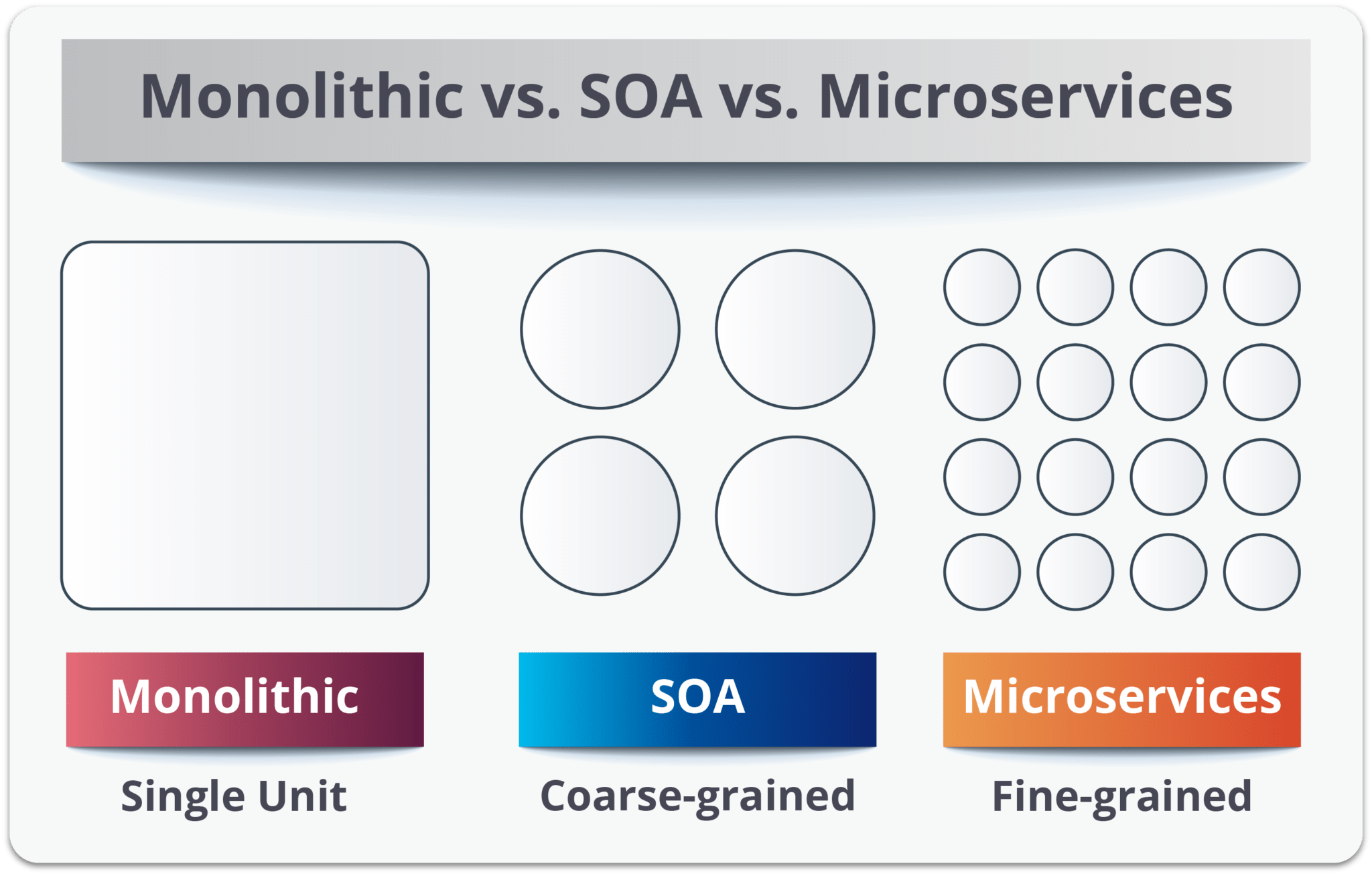
The system architect may decide that interface of your social network; the web pages that you see every time you log in will be easier to manage and maintain when they are written in, let's say, Angular or PHP or Ruby on Rails.

However, messaging systems may be written in Python as the architect may decide that this language offers better libraries for this task.

On the other hand, search engines or other tools that need superb performance are often written in C++. There may be also some internal corporate applications in Java or C#.

Those applications, of course, need to communicate with each other. But how to do that?

There is a concept in software design called **service-oriented architecture** (**SOA**). We just discussed the first part of this principle. It focuses on creating applications around distinct tasks.



(We will look at microservices later in the course. https://youtu.be/EpyPFnjue38 )

If every task is performed by a different application, there is a need for some means of communication between them.

To achieve this goal, applications expose services that are used by other applications.

Typically, they are accessible over some medium, that is, an internal network or the Internet.

They are self-contained and autonomous, which means they are independent of other services and are able to deliver complete response when queried.

They should also be well documented so that any developer can use them.

**Distributed systems**

When—as in our example of social network—we have a system that consists of many autonomous services, we call such systems **distributed systems**.

Depending on the scale, business needs, or technical constraints, the systems may be spread over lots of computers in a local network, the Internet, or just on a single machine.

Benefitting from the SOA principle, you may run and test on your desktop computer distributed system of the same logical architecture, which will be then used on hundreds of servers in the production environment.

There are many advantages of SOA in distributed systems over monolithic applications. Let's discuss some of them.

**Maintainability**

The greatest advantage of distributed systems in SOA is their maintainability, which means ease of performing all the tasks related to the caretaking of the software.

If the system consists of many applications, each dedicated to one task or type of tasks instead of one big monolith, some of the actions can be performed a lot easier:

* You can select tools (that is, programming languages, libraries, and services) that are best for a given task.
* Instead of having all the developers working on one application (that means one code base), you can split the team to work on many applications separately.
* Communication between the different components of the system is narrowed to only one specified interface.
* It is easier to respond to failures and fix bugs. System operators or developers are able to replace a buggy service with a stable version and do some tests to identify the bug or perform other actions without affecting the rest of the system.
* Introducing changes is a lot easier too. In the common workflow, if a new version of a service is to be deployed, it can be run as a separate instance with the old version simultaneously. System operators can switch the client application from the old to the new service and see whether everything performs correctly. If it does, the old service is turned off; otherwise, it is easy to switch back to the old service and fix the new one.

**Scalability**

Many systems are required to perform well under a high load.

The most obvious way, known by every computer user, is to add RAM or switch to a better CPU if applications don't run smoothly.

But there is a limit to such scaling (called **vertical scaling**).

The other type of scalability is **horizontal scaling**, which means adding more computers (called **nodes**) to the system.

In properly designed systems, operators can add or remove nodes depending on the expected load and other circumstances.

More sophisticated systems can even scale themselves, starting or stopping nodes in the cloud automatically, based on the traffic analysis.

SOA allows multiple nodes of the same function to be accessible to the clients.

In most scenarios, traffic to the services is managed by software or hardware load balancers, making it completely invisible for the client.

**Testability**

Another advantage of distributed systems is the easiness of testing them and finding and fixing bugs.

Independence of services means that they can be tested in isolation from the whole system.

Only a particular service's operation is being tested without any influence from other components.

If bugs are found, they can be evaluated and fixed without the need to consider them in the scope of whole system.

**Apache Thrift**

Facebook was started in 2004.

In its early years it faced rapid growth in terms of traffic, system, and network structure.

Their engineering culture allowed choosing any solution that was deemed optimal for a given task without any constraints

This led to a situation when they had lots of different services, but no reliable way to connect them together.

Describing Apache Thrift, Facebook's engineers stated in the white paper (https:// thrift.apache.org/static/files/thrift-20070401.pdf):

*"(...) we were presented with the challenge of building a transparent, high-performance bridge across many programming languages."*

They tested solutions available in the market and came to the conclusion that none of them fulfilled the requirements of high performance, flexibility, and simplicity.

The result of their work was Thrift—a piece of software that was later open sourced and handed over to the Apache Foundation.

Apache Thrift's simplicity comes from the fact that the code for different programming languages is generated automatically from a single file written in the **interface definition language** (**IDL**).

In other similar solutions, data has to be prepared before it is transferred to meet the limitations of the method of transport—not all structures are easily transferred.

In most cases, simple data types such as strings and integers are transferrable.

A developer has to translate every structure more complex than that to the text form in a process called **serialization**.

This has to be done on both ends (deserialization being the reverse process), which needs extra work, testing, and debugging.

In the case of Apache Thrift, the developer can use data types native to their programming language of choice using the methods dedicated to this language.

All serialization and deserialization is made by the Apache Thrift itself and is not visible to the developer.

This architecture of the solution allows programmers to focus on working on the actual services, and not having to care about how the data is going to be transferred from one application to another.

So Thrift is a framework for building services and clients that communicate via remote procedure calls (RPC).

Thrift has grown to become one of the primary means by which the back-end services at Facebook communicate with one another.

Since its development at Facebook, Thrift has been open sourced as an Apache project (http://thrift.apache.org/), where it continues to grow to fill the needs of service developers, not only at Facebook but also at other companies, including Evernote and last.fm, and on major open source projects such as Apache Cassandra and HBase.

These are the major components of Thrift:

* *Thrift Interface Definition Language (IDL)*—Used to define your services and compose

any custom types that your services will send and receive

* *Protocols*—Used to control encoding/decoding elements of data into a common binary format (such as Thrift binary protocol or JSON)
* *Transports*—Provides a common interface for reading/writing to different media (such as TCP socket, UDP etc)
* *Thrift compiler*—Parses Thrift IDL files to generate stub code for the server and client interfaces, and serialization/deserialization code for the custom types defined in IDL
* *Server implementation*—Handles accepting connections, reading requests from those connections, dispatching calls to an object that implements the interface, and sending the responses back to clients
* *Client implementation*—Translates method calls into requests and sends them to the server

Let's look at the pillars of Apache Thrift.

**Supported programming languages**

An important feature of Thrift is that it’s available for a wide variety of languages. This enables teams at Facebook to choose the right language for the job, without worrying about whether they’ll be able to find client code for interacting with other services.

Before starting any work with Apache Thrift, you should probably check whether it supports the programming language that you use. The complete list for version 0.9.3 is as follows:

1. ActionScript 3
2. C++
3. C#
4. D
5. Delphi
6. Erlang
7. Haskell
8. Java
9. JavaScript
10. Node.js
11. Objective-C/Cocoa
12. OCaml
13. Perl
14. PHP
15. Python
16. Ruby
17. Smalltalk

**Data types**

One of the basic features of every programming language is their data types.

Although the basic ones may be very similar, that is, integer or string, it may not be that easy for the rest of them.

Some of the languages (for example, C++) are statically typed.

This means that the type of the variable has to be known at the compile time.

Thus, it has to be defined in the source code when the program is written.

After that, the variable can be of only this type. For example, consider the following line from C++:

int x = 42;

It initializes the variable x, which is an integer.

This variable has to stay an integer through the execution of the program.

Let's take a look at the following example:

int main()

{

int x = 42;

// this line will produce compilation error

x = "forty two";

return 0;

}

If you try to compile this simple code, you will end up with the following compile error:

$ g++ -o example example.cpp

example.cpp: In function 'int main()':

example.cpp:4:6: error: invalid conversion from 'const char\*' to 'int' [-fpermissive]

x = "forty two";

^

Other languages are dynamically typed, that is, the type of the variable is checked in the runtime, but in the source code it might be anything, any time.

Consider this example from PHP:

if (rand(0,1) == 1) {

$x = 42;

} else {

$x = "forty two";

}

var\_dump($x); // var\_dump() function prints type of specified

// variable and its value

Depending on the random outcome of the condition, the value of the variable may be either integer or string.

As you can see, both values are permitted as PHP interpreter changes the type of the variable during the runtime.

Without Apache Thrift, developer themselves would have to serialize the variables.

It means that before the variables are transferred, they should be mapped to the most basic data types that are understood by every programming language (most probably, integers and strings of characters).

After the transmission, those serialized variables have to be translated back to the structures available in the programming language at the receiving end.

Apache Thrift does all that work for the developer.

It provides its own data types that are then mapped to the ones native to the given programming language, thereby allowing the developer to focus on creating the application, not the communication interface.

**Transports**

Transports are a part of Apache Thrift's network stack.

They allow you to transmit data over different channels, that is, HTTP, sockets, or files.

Decoupling the transport layer lets you to easily choose the transport that best fits your solution without many changes in the code.

**Protocols**

Protocols prepare data to be transmitted over transports.

The name of the process is called **serialization** (when sending data) and **deserialization** (when receiving data).

There are different protocols that can be used: JSON, binary, plain text, and so on.

It means that depending on what data you want to transfer, you can use different methods of serialization.

For example, if you expect to transmit images or other binary data, choosing the binary protocol is the best option as there would be almost zero overhead.

If you chose JSON for this purpose, binary data would be converted to text, thereby increasing the payload by a third or more.

The choice of protocol should be dictated by the data you wish to transfer using Apache Thrift.

**Versioning**

Versioning is an approach for managing changes in the service's API (and in the software in general).

As software is being developed, it changes.

Sometimes the changes are miniscule, and sometimes great.

They are often manifested by modification of the methods or parameters exposed by the API.

When developing client and server software, you shouldn't assume that clients will be updated to the newest version instantly.

It is also wise to allow the older versions of the client to work with the newer versions of the server.

Changes in the APIs, libraries, and other externally available components pose a big challenge for the developers, leading to problems often referred to as **dependency hell**—when different applications are compatible with different versions of the same library or API, leading to difficulties with managing those dependencies.

To alleviate this inconvenience, most of the software developers adopt a convention of marking the version of the application with decimal numbers, according to the template, MAJOR.MINOR.PATCH, where PATCH means miniscule changes (that is, fixing some bugs), MINOR is a larger change but backward-compatible with the previous versions, and MAJOR means a major release that might break the compatibility with the previous versions of the software.

Apache Thrift has a set of tools that allows users to easily keep backward compatibility with the new versions of the service.

It is achieved by the following properties:

* The method's arguments are numbered. You can add or remove them. As long as the same number is not reused, the new versions of methods may function without removed arguments. Those numbers shouldn't be changed for any existing argument.
* You can set default values for the arguments, so if the older version of the client has a method without a new variable, the service doesn't receive any value for such an argument and the default value is set. This is useful when you want to add some fields.
* While manipulation with fields is relatively easy, you shouldn't rename methods or services. This makes them unavailable for the older clients.

**Security**

Security is essential to every service.

Although you definitely need to take extra care when exposing services to the Internet, it is also important when they are available in private networks.

Apache Thrift allows you to use TSSLTransportFactory to utilize RSA key pairs, providing security for the connection.

**Interface description language**

Apache Thrift's core feature is its own IDL

Using IDL, you are able to define the service and all the variables that it uses in one file.

It is an unambiguous description of what the service will look similar to without going into the implementation details.

Let's consider a very simple service, which allows you to add two integers:

namespace py thrift.example1

namespace php thrift.example1

service AddService {

i32 add(1: i32 a, 2: i32 b),

}

This example code defines AddService service, which contains the add method.

This method takes two 32-bit signed integers (i32) as parameters and also returns such an integer as a result.

We will want to have the code generated for Python and PHP languages, but Apache Thrift is able to do it for a far greater number of languages.

Now the Thrift's magic begins; if you save this code to the file (let's say, example1. thrift) and run the following commands, you will get the code of client and server for this service in desired languages (Python and PHP in this example) in the newly-created folders, gen-py and gen-php:

**$ thrift --gen py example1.thrift**

**$ thrift --gen php example1.thrift**

In the simplest solution, it is enough to fill the code of the add method, and voilà, you have a fully-functional client and server.

This example shows the major advantage of Apache Thrift—the ability to define in one place and then instantly generate services and the corresponding client code without the need of writing code in every language from scratch.

To see how much work Apache Thrift just spared you, examine the generated files that are saved in the gen-py and gen-php folders when we do the lab.

**Apache Thrift and others**

Until now, you may have come to the conclusion that Apache Thrift is the best solution for all your needs when dealing with distributed systems.

Surprisingly, it is not always true.

Frequently, inventing your own custom protocol is the first idea that comes to a developer's mind when he/she needs to transfer data between two applications (e.g. your project). Other technologies include; XML-RPC, JSON-RPC , SOAP and WSDL, RESTful APIs (all covered later)

**CORBA**

No longer covered

**Common Object Request Broker Architecture** (**CORBA**), http://www.corba. org/, dates back to 1991, and is the oldest of the standards. However, its concepts are quite similar to Apache Thrift (for example, it uses its own IDL).

It is considered a bit cumbersome; instead of using a language's native code, a developer needs to use a CORBA-specific one.

Also Its based around distributing objects which the industry has moved away from

It's hard to install and heavy to run. There are different implementations and they are inconsistent.

Other distributed object technologies include DCOM, ICE and Java RMI (no longer covered)

See Fowler’s first law of Distributed Objects: <https://www.martinfowler.com/articles/distributed-objects-microservices.html>

**Apache Avro**

Apache Avro (https://avro.apache.org/) is another remote procedure call and data serialization framework developed with the support of the Apache Foundation. It was developed as a tool for the Apache Hadoop framework.

Lots of similarities to Apache Thrift include describing the interface with IDL, support for many programming languages (Java, C, C++, C#, Scala, Python, and Ruby), and a compact, fast binary format.

The main difference is that Avro's code doesn't have to be generated when the service is defined and later on, when it changes.

As a disadvantage in comparison with Apache Thrift, Apache Avro doesn't offer such a wide selection of serialization formats (protocols, in Thrift's terminology) and transports.

**gRPC and Protocol Buffers**

gRPC and Protocol Buffers are an *older brother* of Apache Thrift, and they share lots of similarities.

They were developed as an internal proprietary software in Google and are used in most of the inter-machine communication.

Since their release to open source in 2008, they have gained support not only for officially implemented languages (C++, Java, and Python), but also a lot more (JavaScript, Go, PHP, Ruby, Perl, and Scala).

Apart from IDL syntax and implementation details, Protocol Buffers differ from Apache Thrift in that they have less language support, different base types, a lack of constants and containers, and no built-in exception handling.

On the other hand, Protocol Buffers are a little bit faster than Apache Thrift and their objects are smaller.

Also the documentation and availability of tutorial is considered better and more complete.

**When to choose Apache Thrift**

When designing and developing applications that have to communicate with each other, one may go through the whole evolution process involving the solutions presented above

Many services start as a very limited tool, which works quite well with some simple custom protocol.

But the data that needs to be transferred may become more and more complicated than the need for some format, such as JSON or XML appears JSON-RPC or XML-RPC may be then used.

As the service is growing and is exposed to more external applications, the need to standardize the architecture and proper documentation arises.

In such cases, using web services based on SOAP and WSDL seems to be a proper idea.

If your application's goal is to operate on collections of elements, RESTful API may be a good solution.

But there are situations where one needs to transfer binary data and provide flexibility for changing the definition of the services along with support for different platforms and languages; all this in an environment where high performance is crucial.

In these cases, serialization and remote procedure call for frameworks such as Apache Avro, gRPC and Protocol Buffers, and Apache Thrift.

From these three, the last one offers the widest selection of serialization formats, and transports along with remote procedure call implementation.

In the lab, we will install Apache Thrift, generate and run our simple services, and discuss the features in great detail.

References:

See Netty in Action chapter 15

See Learning Apache Thrift chapter 1

See thrift.apache.org

See code examples in Java at <https://thrift.apache.org/tutorial/java>