Apache Thrift & Finagle

REST Sample

- Shared library + serialization *
- Implement HTTP Routing on Server Side
- Use Request library for communication

* - possible

Shared library + serialization

```
         case class Person(id: Int, name: String)

     object PersonJsonProtocol
            extends DefaultJsonProtocol {

      implicit val personFormat = jsonFormat2(Person)

}
```

HTTP Routing

```
// spray routing
pathPrefix("persons" / IntNumber) { id =>
  respondWithMediaType(json) {
    persons.get(id) match {
      case Some(person) =>
        complete(person)

      case None =>
        complete(NotFound)
    }
  }
}
```

HTTP Client

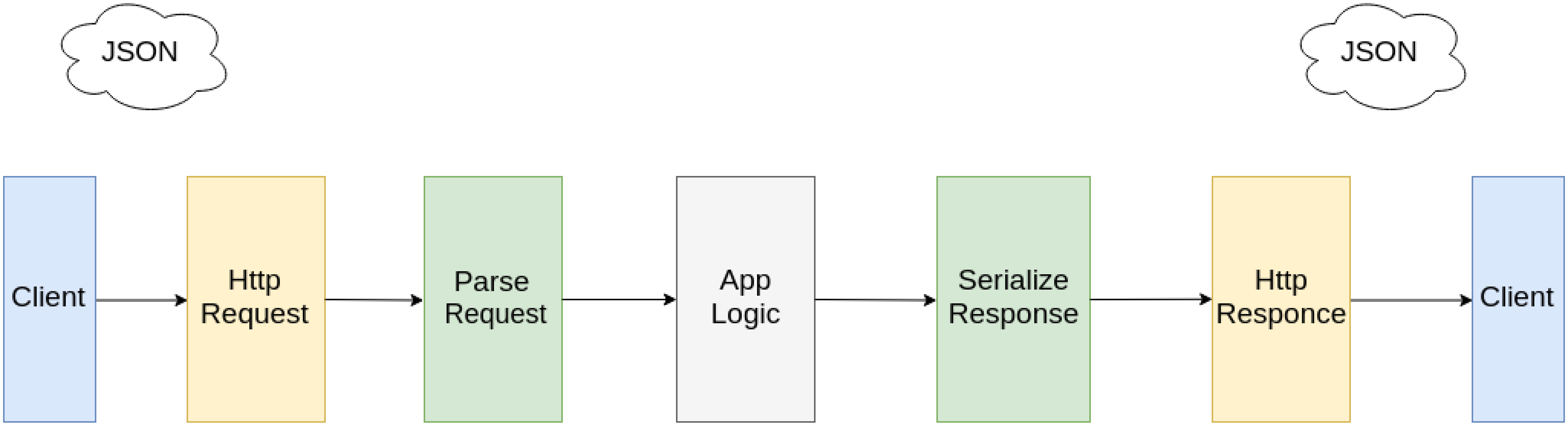
```
val pipeline: HttpRequest => Future[HttpResponse] =
    sendReceive


val response: Future[HttpResponse] =
    pipeline(Get(personsUrl))

// Usage
response.foreach {
  case HttpResponse(status, entity, _, _) if status == OK =>
    val person = entity.asString.parseJson.convertTo[Person]
    println(s"done: $person")

  case HttpResponse(status, _, _, _) =>
    println(s"error with status: $status")
}
```

Client-Server



disadvantages

(at first glance)

1. Shared libraries only for __one__ platform/language
2. Need implement http routing/http clients every time
3. Maybe text protocol is slow (serialization)

Thrift to the rescue

The Apache Thrift software framework, for scalable cross-language services development (from https://thrift.apache.org/)

How to create x-language application?

1. thousands of meetings :)
2. implement models #1
3. implement formats (xml, json) #1
4. implement restful api #1
5. implement models #2
6. implement formats (xml, json) #2
7. implement client #2
8. testing
9. production

How to create x-language application?

1. thousands of meetings :)
2. implement models #1
3. implement formats (xml, json) #1
4. implement restful api #1
5. implement models #2
6. implement formats (xml, json) #2
7. implement client #2
8. testing
9. production

Common parts

server = models + (de)serialization + communication api

client = models + (de)serialization + communication api

IDL (interface description language)

IDL

```
   Java Types
+  C# Types
+  Ruby Types
+  C++ Types
+  Javascript Types
+  C Types
=  IDL
```

IDL – base types

bool

byte/i16/i32/i64

binary

double

string

```
bool flag;
i64  id;
double weight;
string name;
binary logo;
```

IDL - containers

List

Set

Map

```
list<string> names;
set<i64> ids;
map<i64, string> id2name;
```

IDL - typedef

Like alias

```
typedef i64 age
```

IDL - structs

```
struct Person {
  i64  id;
  string name;
}
```

IDL - fields

Optional or Required and Default

```
required string name;

optional string nickname =
"anonymous";
```

IDL – fields order

```
struct Person {
  1: i64  id;
  2: required string name;
  3: string nickname;
}
```

IDL - enum

```
enum Levels {
  A;
  B;
  C;
}
```

IDL

const

```
const double PI = 3.14;
```

include

```
include "example.thrift"
```

namespace

```
namespace java com.example.thriftify
```

IDL - exceptions

Define like struct

```
exception MyException {
  1: i16 code;
  2: string message;
}
```

IDL - services

```
service TestService {
  string getTestData(1: string sample) throws (1:
MyException ex,
   NotFoundException nfex);

  void putTestData(1: string key, 2: string value);

  Statuses status();
}
```

IDL - generate

```
thrift --help <- usage
thrift --gen java src/main/thrift/example.thrift
```

output:

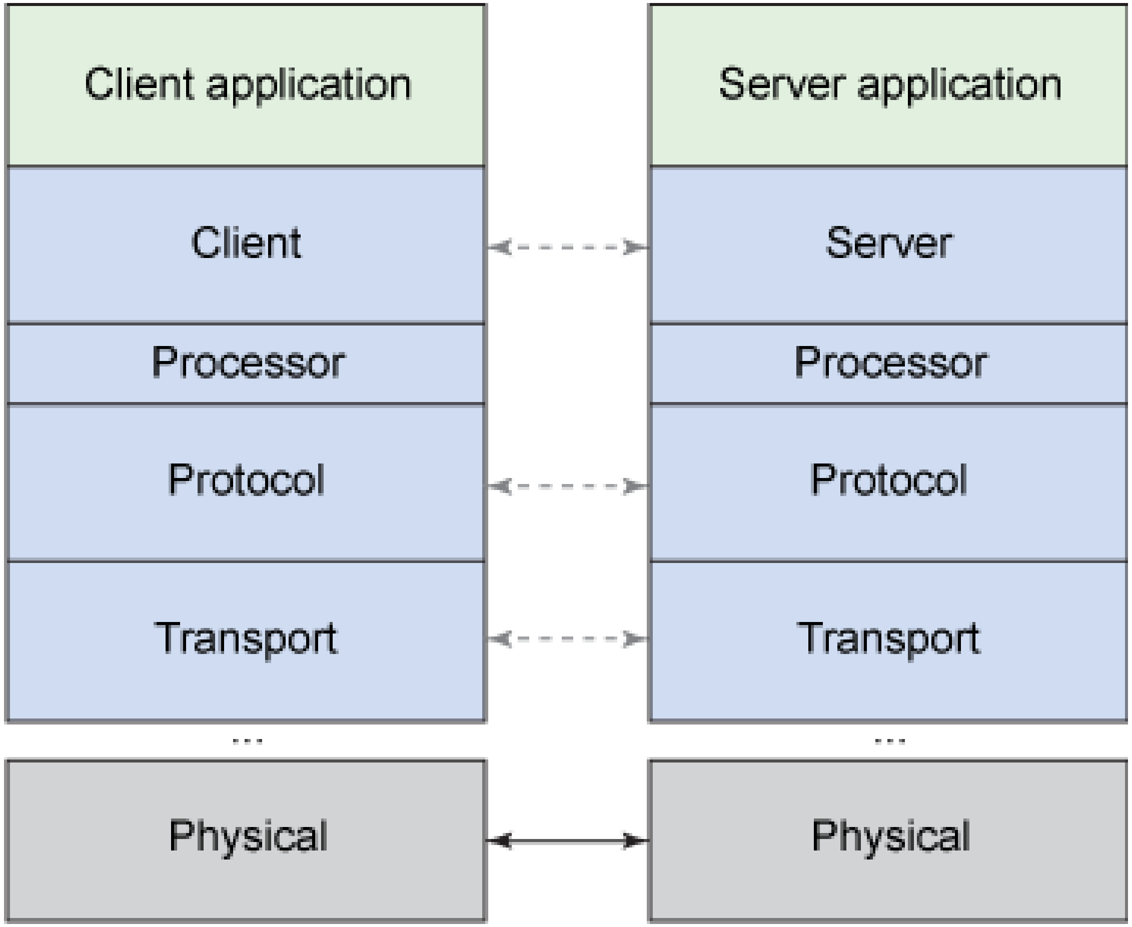
Iface – interface for service

Client – base client for service

Processor – serialization layer

For one line service ~ around 1kLOC

Thrift overview



Protocol

- Just serialization – convert thrift structures to a format
- Types:
- binary
- compact
- json

Transport

- Read and Write over network
- Types:
- TSocket
- TFramedTransport
- TFileTransport
- TMemoryTransport
- TZlibTransport

Processor

- around blocks/functions for serialization
- input_function(args) -> YourServiceMethod() -> output_function(result)

  =

  parse request -> business logic -> complete

Server

- Create transport
- Create protocols for transport
- Create processor based on protocols
- ...Wait
- Types (for java)
- TSimpleServer
- THsHaServer
- TNonblockingServer
- TThreadPoolServer
- TThreadedSelectorServer

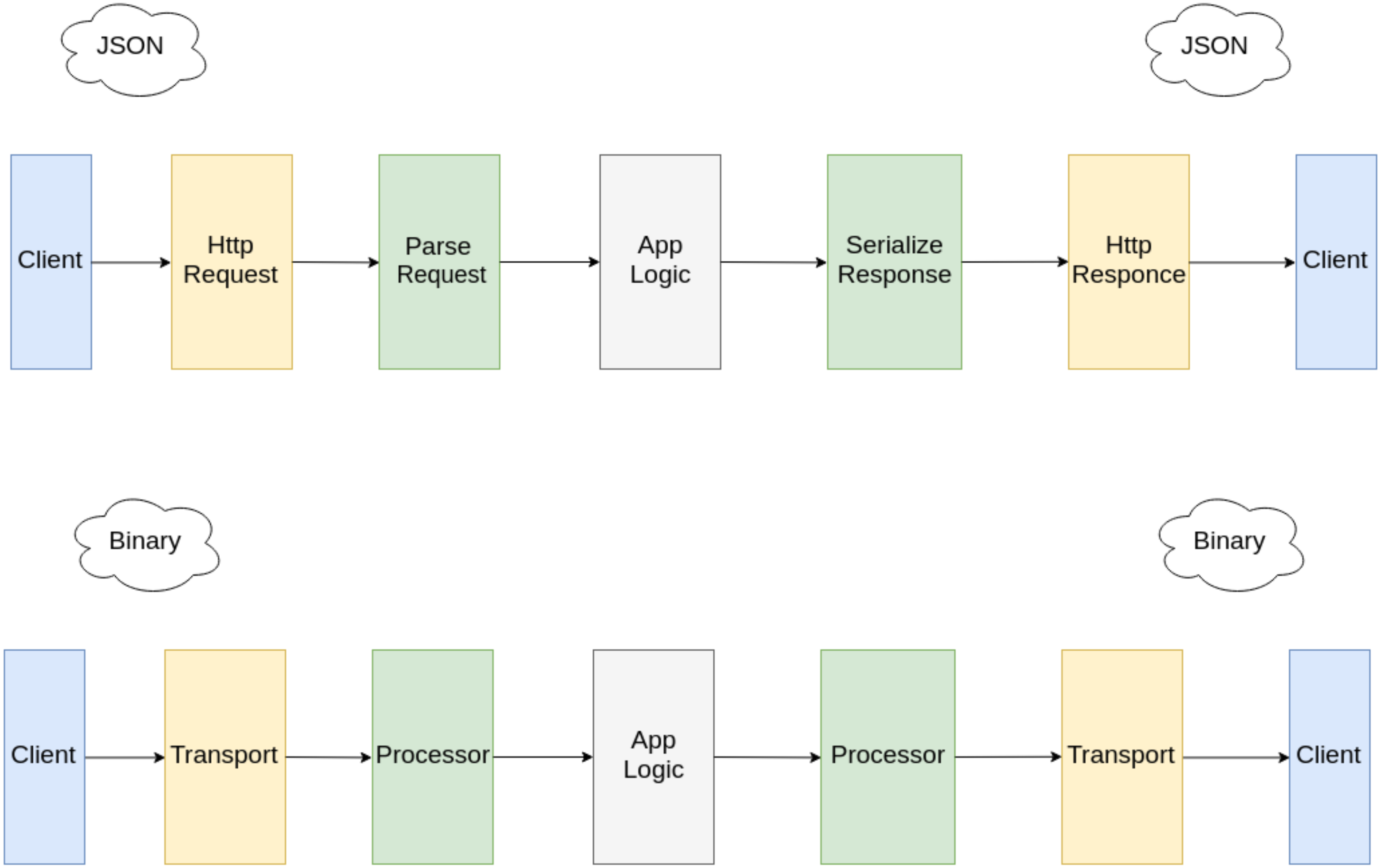
- How to choose server?
- Read: https://github.com/m1ch1/mapkeeper/wiki/Thrift-Java-Servers-Compared

Example/Server

```
     class MyTestServiceImpl implements
         TestService.Iface {          //! interface

         public String getTestData() throws Texception
{
          return "{'result': " +
             UUID.randomUUID().toString() +
          "}";
        }
}
```

Client-Server



Example/Server

```
service = new MyTestServiceImpl();
processor =
   new TestService.Processor<>(service);
transport = new
   TNonblockingServerSocket(port);
```

Example/Server

```
args =
   new TnonblockingServer.Args(transport)
  .processor(processor);

  TServer server = new TNonblockingServer(args);

  server.serve();
```

Example/Client

```java
transport = new TFramedTransport(new TSocket(host, port));
protocol = new TBinaryProtocol(transport);

client = new TestService.Client(protocol);

transport.open();


String result = client.getTestData();
System.out.println("Result = " + result); //
// => Result = {'result': c2850122-e24f-40e9-9512-49122a5cde9c}

transport.close();
```

Compatibility

- Numeric tags are immutable
- Struct fields, method types are immutable
- New fields should be optional
- Optional fields can be removed

Versioning

	old client/ new server	new client/ old server
add field	🟩	🟩
remove field	🟩	🟥

Debug

- Use thrift-tools (https://github.com/pinterest/thrift-tools)

```
sudo thrift-tool --iface lo --port 9000 dump --show-all --pretty
```

```
// client request
[16:11:36:931287] 127.0.0.1:59256 -> 127.0.0.1:9000: method=getTestData, type=call, seqid=1
header: None
fields: fields=[]
// server response
------>[16:11:36:931831] 127.0.0.1:9000 -> 127.0.0.1:59256: method=getTestData, type=reply, seqid=1
        header: None
        fields: fields=[(string, 0, {'result': 064bfeea-2a6d-4fdb-a51d-322945eddd53})]
```

disadvantages

Some IDL limitations (polymorphism/overloading, etc: https://thrift.apache.org/docs/features)

One service by one port

Hard to debug

Summary

IDL for clients and services definition

Transport – read/write over network (IO wrapper)

Protocol – serialization

Processor – around block

Server – choose by latency/throughput

Versioning/Compatibility

Debug

References

https://thrift.apache.org/static/files/thrift-20070401.pdf

http://thrift-tutorial.readthedocs.io/en/latest/thrift-stack.html

https://diwakergupta.github.io/thrift-missing-guide/#_language_reference

https://thrift.apache.org/docs/concepts

https://thrift.apache.org/docs/features

https://github.com/m1ch1/mapkeeper/wiki/Thrift-Java-Servers-Compared

https://github.com/pinterest/thrift-tools

Finagle

Finagle

Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers. (from https://twitter.github.io/finagle/)

Finagle implements uniform client and server APIs for several protocols, and is designed for high performance and concurrency.

Finagle

Server is just a Function

```
type Service[Req, Res] =
  Req => Future[Res]
type Function[In, Out] =
  In => Out
```

Finagle

Finagle = Future + Service + Filter

Future – the result of an asynchronous operation

Service – function, represent client and server

Filter – also function (modify input/outpu), block around service (not dependent on application logic), applicable for client and server

Finagle/IDL

```
namespace java com.twitter.finagle.example.thriftjava
#@namespace scala com.twitter.finagle.example.thriftscala


service MyService {
  string hi(string name);
}
```

Finagle/Server

```
class MyServiceImpl extends MyService[Future] {
  override def hi(name: String): Future[String] = {
    Future.value(s"hi, $name")
  }
}


val impl = new MyServiceImpl

val service = Thrift.server.serveIface(addr, impl)

Await.ready(service)
```

Finagle/Client

```
val client = Thrift.client
  .newIface[MyService.FutureIface](s"$addr")


client.hi("foo")

  .foreach(result => println(s"$result"))
```

Finagle/Filters

```
val whoopFilter = new SimpleFilter[String, String] {
  override def apply(name: String, service:
Service[String, String]): Future[String] = {
    service(s"$name!")

  }
}


val hiw = (s: String) => client.hi(s)

val f = whoopFilter andThen hiw

f("foo").foreach(result => println(result))
```

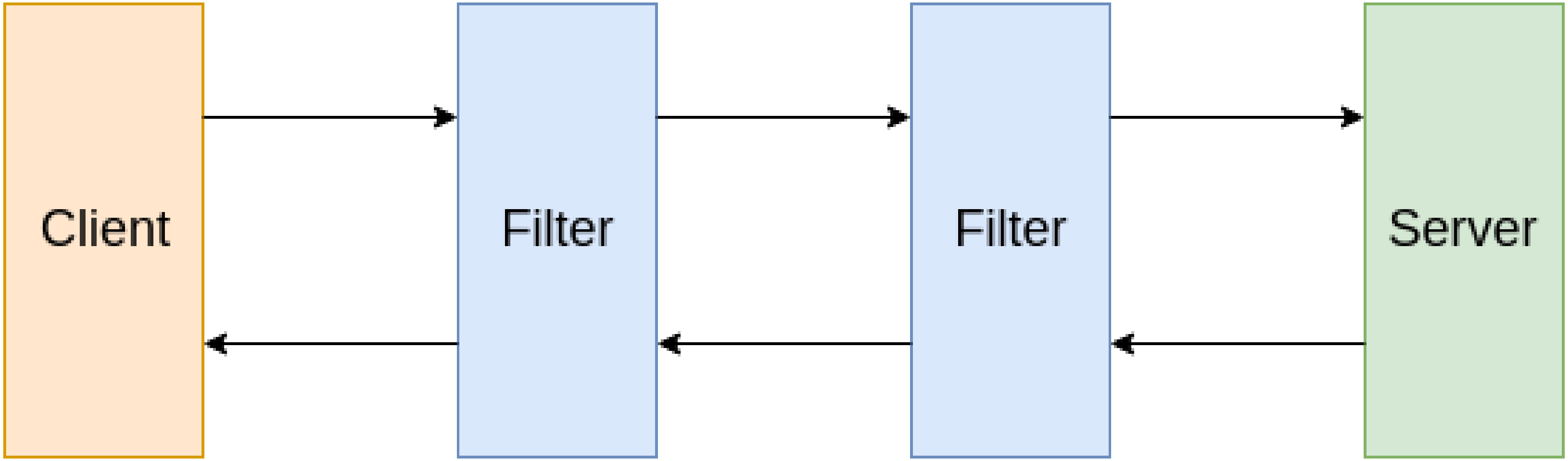
Finagle/Server Requirements

- Monitoring
- Tracing
- Stats
- Logs
- Concurrency Limit
- Rejecting Requests
- Request Timeout
- Session Expiration

Finagle/Client Requirements

- Monitoring
- Tracing
- Stats
- Logs
- Retries
- Timeouts & Expiration
- Load Balancer

Finagle/Filters



Finagle/Load Balancing example

```
val client = Thrift.client
  .newIface[MyService.FutureIface](s"$addr,
$addr1")


// output:

// service: 8000

// service: 8000

// service: 8001

// service: 8001
```

Finagle/Load Balancing example

Service discovery, service announcement

```
val service =  Thrift.server.serveIface(
       s":$port", impl
    )
```

```
service.announce(
  s"zk!$host:$zkPort!/service/impl!0"
)
```

Finagle/Load Balancing example

```
val client = Thrift.client
         .newIface[MyService.FutureIface](
  dest = s"zk!$host:$zkPort!/service/impl",
  label = "my-service"
)
```

Finagle/Load Balancing example

```
// in zookeeper cli

[zk: localhost:2181(CONNECTED) 6] ls /

   [zookeeper, service]

[zk: localhost:2181(CONNECTED) 9] ls /service

   [impl]

[zk: localhost:2181(CONNECTED) 10] ls /service/impl

   [member_0000000000, member_0000000001]
```

Finagle/Tracing

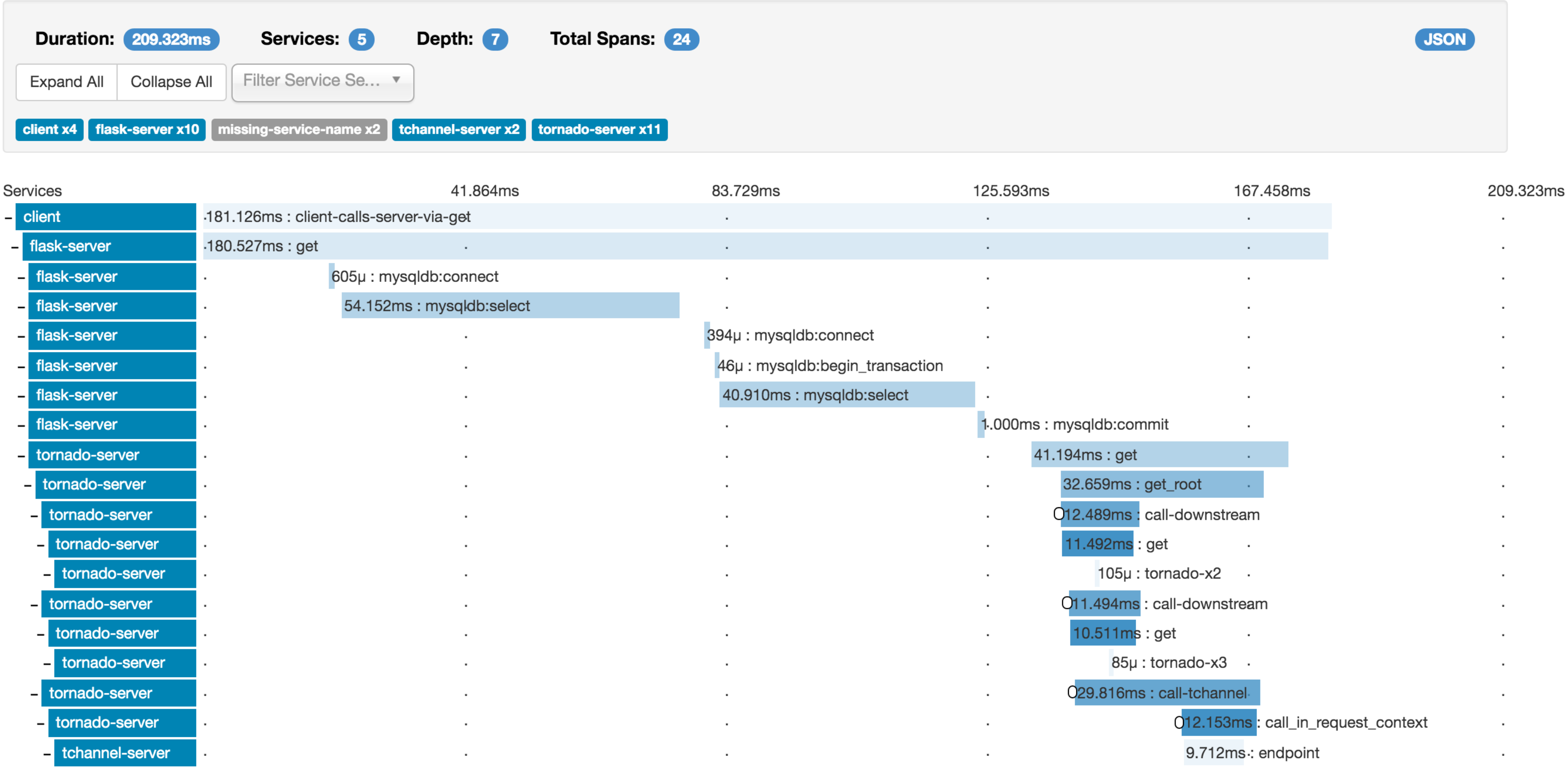
Zipkin project (http://zipkin.io/)

```
val service =  Thrift.server
  .withTracer(new HttpZipkinTracer())
  .withLabel("thrift-impln-service")
  .serveIface(addr, impl)
```

Finagle/Tracing

```
val client = Thrift.client
  .withTracer(new HttpZipkinTracer())
  .withLabel("thrift-impl-client")
  .newIface[MyService.FutureIface](addr)
```

Finagle/Tracing



Duration: 209.323ms **Services:** 5 **Depth:** 7 **Total Spans:** 24 JSON

Expand All | Collapse All | Filter Service Se... ▾

client x4 | flask-server x10 | missing-service-name x2 | tchannel-server x2 | tornado-server x11

Services 41.864ms 83.729ms 125.593ms 167.458ms 209.323ms

Services	
– client	181.126ms : client-calls-server-via-get
– flask-server	180.527ms : get
– flask-server	605μ : mysqldb:connect
– flask-server	54.152ms : mysqldb:select
– flask-server	394μ : mysqldb:connect
– flask-server	46μ : mysqldb:begin_transaction
– flask-server	40.910ms : mysqldb:select
– flask-server	1.000ms : mysqldb:commit
– tornado-server	41.194ms : get
– tornado-server	32.659ms : get_root
– tornado-server	O12.489ms : call-downstream
– tornado-server	11.492ms : get
– tornado-server	105μ : tornado-x2
– tornado-server	O11.494ms : call-downstream
– tornado-server	10.511ms : get
– tornado-server	85μ : tornado-x3
– tornado-server	O29.816ms : call-tchannel
– tornado-server	O12.153ms : call_in_request_context
– tchannel-server	9.712ms : endpoint

Summary

Future

Service

Filter

References

- https://blog.twitter.com/2011/finagle-a-protocol-agnostic-rpc-system
- https://monkey.org/~marius/funsrv.pdf
- https://twitter.github.io/finagle/
- http://twitter.github.io/finagle/guide/
- https://zookeeper.apache.org/
- http://zipkin.io/
-

Conclusion

Monitoring, Stats

Tracing, Logging

Provide dependencies, building client/format libraries

Testing

Release process