# SRE your gRPC

## Building Reliable Distributed Systems

Introduction:

- Hi, I'm Gráinne, and that's Gabe
- We run distributed Ads systems for Google. We like failin'… when nobody notices =)

- We're going to share the questions we ask to build better reliable systems.
- And we'll give you actual examples using gRPC (our open source remote procedure call) in C++

# Our Rules

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. Fail early.
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. Best effort isn't.

Image: (modified from) Jenga
by Silver Blue (flickr: cblue98), CC-BY 2.0

# Our ~~Rules~~ *Guidelines*

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. Fail early.
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. "Best" effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

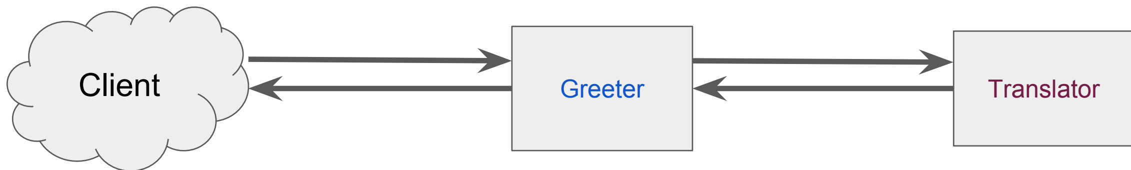
# Our ~~Rules~~ *Guidelines*

1. **Everything is (secretly) a stack.**
2. Sharing is caring.
3. Fail early.
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. "Best" effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

Allow me to introduce you to our guidelines using our service stack and our c++ code.

# Everything is a stack ?




```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    std::string prefix("வணக்கம்");
    reply->set_message(prefix + " " + request->name());
    return Status::OK;
  }
};
```

http://grpc.io/

This hello greeter is taken almost verbatim from the gRPC example code, available from github,
https://github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_server.cc
This is a c++ example, and it's easy to put together.

We're going to add a translator backend to build our stack. And show you how we as SREs would improve it.
First thing we need to know is the protocol buffer for the Translator.
(I'm informed that prefix is pronounced "Vaṇakkam" and means "hello" in Tamil.)

# Everything is a stack ? the translator.proto file

```
enum Type {
  FORMAL = 0;
  RUDE = 1;
}

message TranslationRequest {
  string message = 1;
  string language = 2;  // e.g. "en"
  Type type = 3;
}
```

```
enum Type {
  FORMAL = 0;
  RUDE = 1;   [deprecated = true];
  SLANG = 2;
}
message TranslationRequest {
  string message = 1;
  reserved 2;  // prevent future badness from reuse.
  Type type = 3;
  string locale = 4;   // e.g. "en_US"
}
```

```
service Translator {
 rpc Translate (TranslationRequest) returns (TranslationReply) {}
}
```

Your services will need to change, and the protocol buffers are here to help.

This is the translator protocol definition.
https://developers.google.com/protocol-buffers/docs/proto3
They are language independent.
The numbers are called tags. These are used to generate helper code, and create the "stub" to call the service.

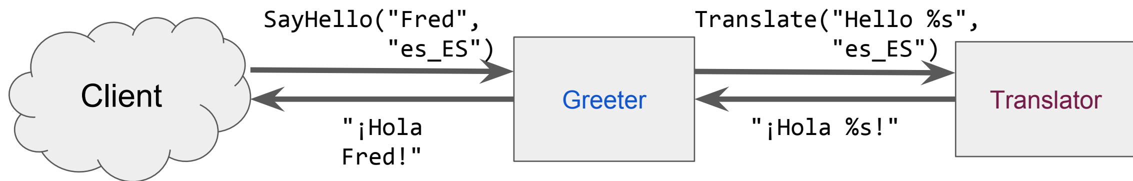
This is it now and in the near future.

Guideline : Never change types of fields or reuse them. Deprecate them or reserve them.

If you update a message type by entirely removing a field, or commenting it out, future users can reuse the tag number when making their own updates to the type. This can cause severe issues if they later load old versions of the same .proto, including data corruption, privacy bugs, and so on. Specify that the field tags (and/or names, which can also cause issues for JSON serialization) of your deleted fields are reserved. The protocol buffer compiler will complain if any future users try to re-use these field identifiers.

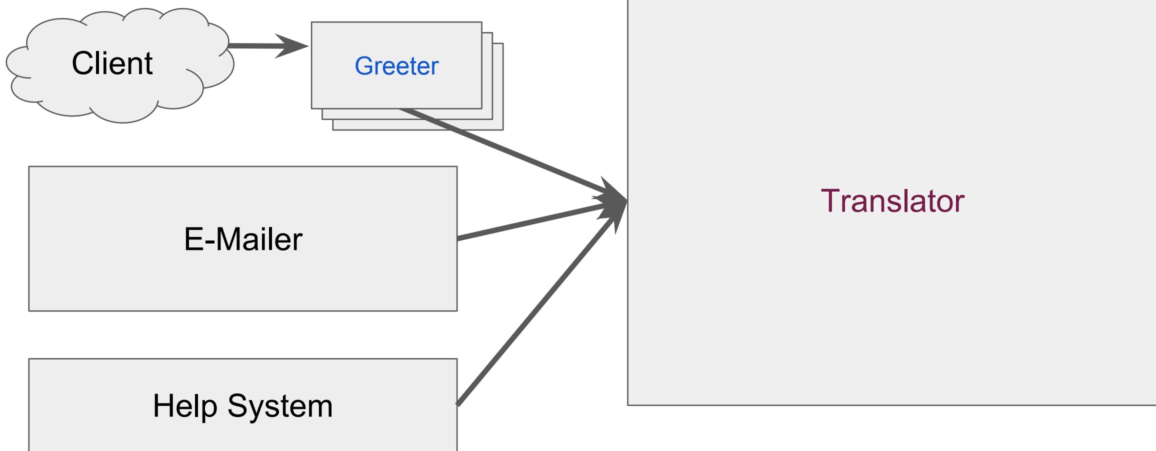
We set the message, an enum and rpc service interface in a .proto file.
The protocol buffer compiler will generate service interface code and stubs in your chosen language.

# Everything is a stack ?



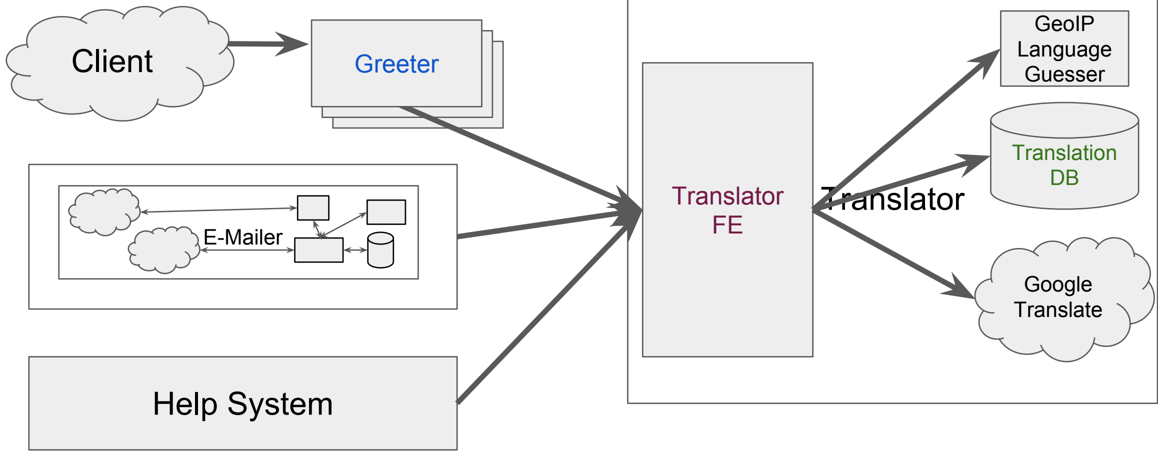

The client asks it to say hello and gives it a name and a locale, the server asks the translator backend for the localized string, and returns the hello string.

By default we'll get the first ENUM. Consider carefully what you want the defaults to be.




The Translator here has multiple "clients".

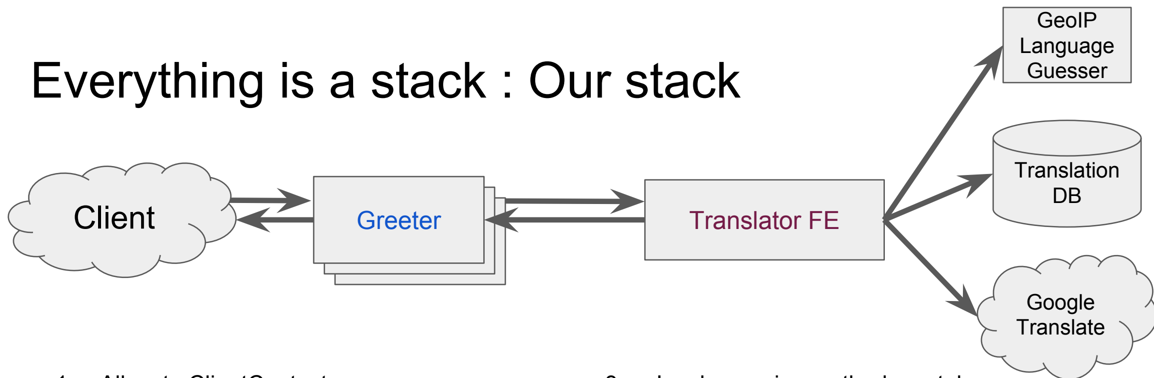
All of the clients and all of the backends are probably complex systems themselves.




Not all backends are perhaps used by all of the Translator's clients.

The translator is important to many systems and must be resilient:
If one of the backends goes bad, it shouldn't cause greeter, e-mailer or help systems to go down or crash.

So let's check out the code for simply adding the translator to the greeter.




1. Allocate ClientContext

```
ClientContext greeter_context;
```

2. Allocate request/response and fill in request

```
TranslationRequest translation_request;
TranslationReply translation_reply;
translation_request.set_message("Hello");
translation_request.set_locale(
   request->locale());
```

3. Invoke service method on stub –
   gRPC forwards call to server.

```
Status status = stub_->Translate(&context,
    Translation_request, &translation_reply);
```

4. Check call status, and handle the response

```
if (status.ok()) {
  prefix = translation_reply.translation();
}
```

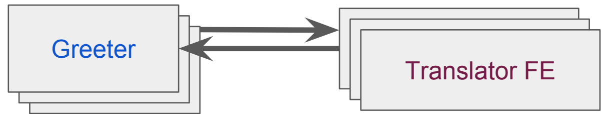
https://github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_client.cc

This adds a Translator to the basic Greeter example.
The context object is the entire RPC state.

The gRPC library takes care of the communication, marshalling, unmarshalling.

# Everything is a stack : Our code

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    std::string prefix("வணக்கம்");
    ClientContext greeter_context;  // 1
    TranslationRequest translation_request;  // 2
    TranslationReply translation_reply;
    translation_request.set_message("Hello");
    translation_request.set_locale(request->locale());
    Status status = stub_->Translate(&context, translation_request, &translation_reply);  // 3
    if (status.ok()) {  // 4
      prefix = translation_reply.translation();
    }
    reply->set_message(prefix + " " + request->name());
    return Status::OK;
  }
};
```




https://github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_client.cc

In context of the Greeter example… it's that easy!

But how does a Greeter process pick a translator process to send the rpc to?

# Our ~~Rules~~ *Guidelines*

1. Everything is (secretly) a stack.
2. **Sharing is caring.**
3. Fail early.
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. "Best" effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

# Sharing Is Caring: Load Balancing?

Elephant: Chris Hills (flickr: justcallmehillsy), CC-BY-SA 2.0
Ball: Timothy Krause (flickr: timothykrause), CC-BY 2.0

## Sharing Is Caring: Load Spreading!

Image: pixabay, CC0

vs.

Image: pixabay, CC0

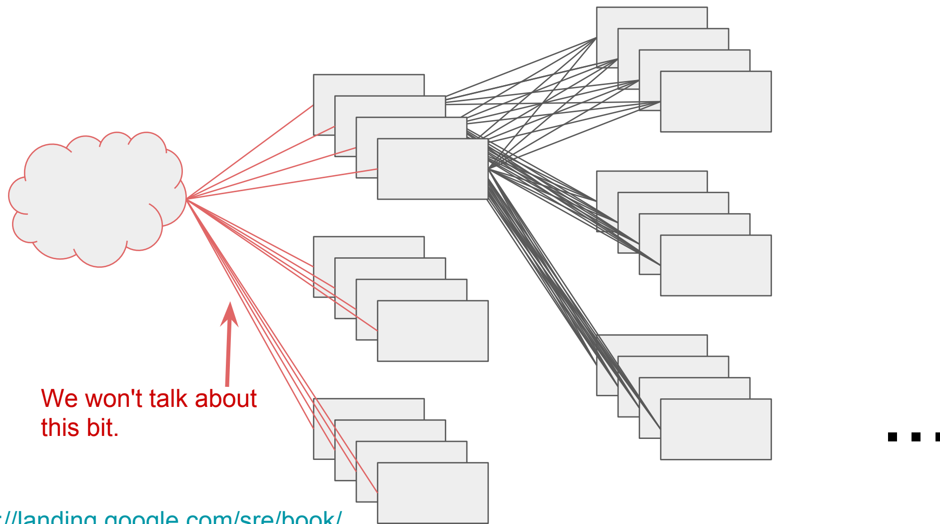
Elephant: Chris Hills (flickr: justcallmehillsy), CC-BY-SA 2.0
Ball: Timothy Krause (flickr: timothykrause), CC-BY 2.0

We really mean load spreading. But, we call it Load Balancing anyway, because that's what everybody else calls it.

Generally, there are three things we achieve:

- Redundancy: Failover
- Utilization: Hot/hot instead of hot/cold
- Capacity: More servers, maybe even more datacentres.
  ("Latency" is just another part of "capacity" here - it's in your SLA, right?)

These can be traded against each other to a degree.



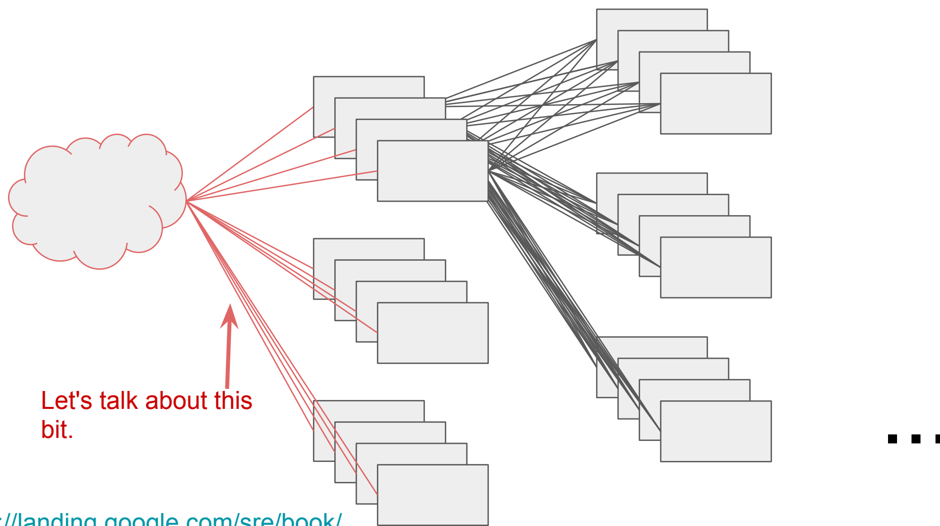

We won't talk about global load balancing for the web.
That's DNS based, and to do it your DNS service needs to have decent assumptions, given a query:

- Where is the actual client, as opposed to their ISP's recursive name server?
- How many clients will this ISP send to you before the TTL expires
- Source location of the query based on query source (location, and size - ISPs vary greatly, 8.8.8.8 doesn't make this easier for you, there are EDNS mechanisms to help, but all of this is outside the service).

See https://landing.google.com/sre/book/chapters/load-balancing-frontend.html
You have TCP connection balancing, and then everything comes in for "the HTTP service!" anyway, so you'll need some kind of proxy to forward the actual request to the correct serving backend.

Zoom in! Enhance!

## Sharing Is Caring: Load Balancing




Remember how everything is a stack? Your clients are now those proxies.
See https://landing.google.com/sre/book/chapters/load-balancing-datacenter.html

So **what balancing strategy** should you use?

When you want to get started asap, use Round Robin. If in doubt, use Round Robin.
From a **random initial choice**, finds the first available client channels and works through them in turn.
**Worst case**, it fails all traffic being served by bad tasks. X bad tasks out of a total of N tasks, will fail X/N of the total traffic.

It's easy to reason about but not optimised for resource usage.

# Sharing Is Caring: Load Balancing

Load Balancing Strategies?

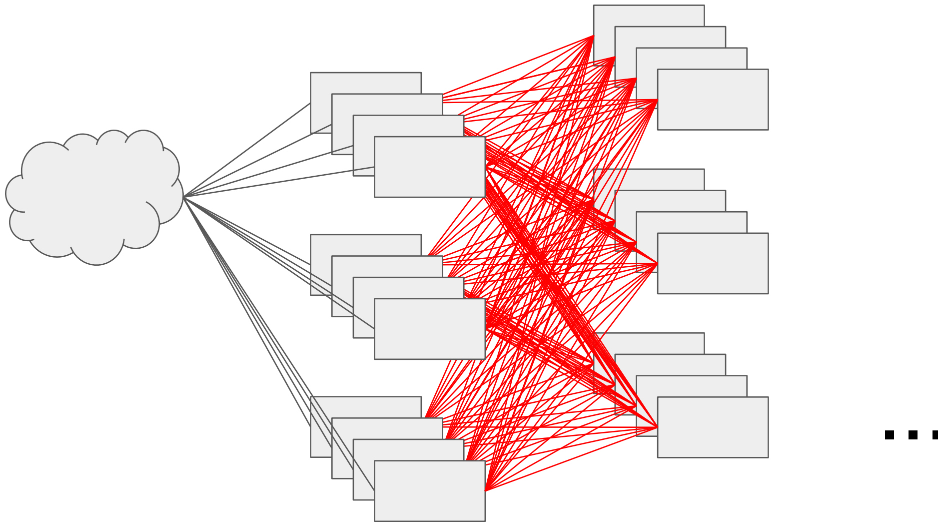
- Round Robin
  - Easy to reason about.
  - Spreads requests reasonably evenly.
  - Ignores differences in query cost and backend capacity.
- Least Loaded
  - Assigns work where there is capacity.
  - May lead to thundering herds, if there is a central coordinator, overload if there is none (other clients are ignored).
  - Errors are (should be!) cheap and fast to serve!
- Weighted Round Robin
  - Needs a definition of "cost", reported by the backends.
  - Gives good results, if all components play along.

When in doubt, use round robin!

It will not give you the best results, but its failure mode is probably not catastrophic.

And you will understand it when you get paged at six a.m.

The strategies are discussed in detail in the SRE book:
https://landing.google.com/sre/book/chapters/load-balancing-datacenter.html

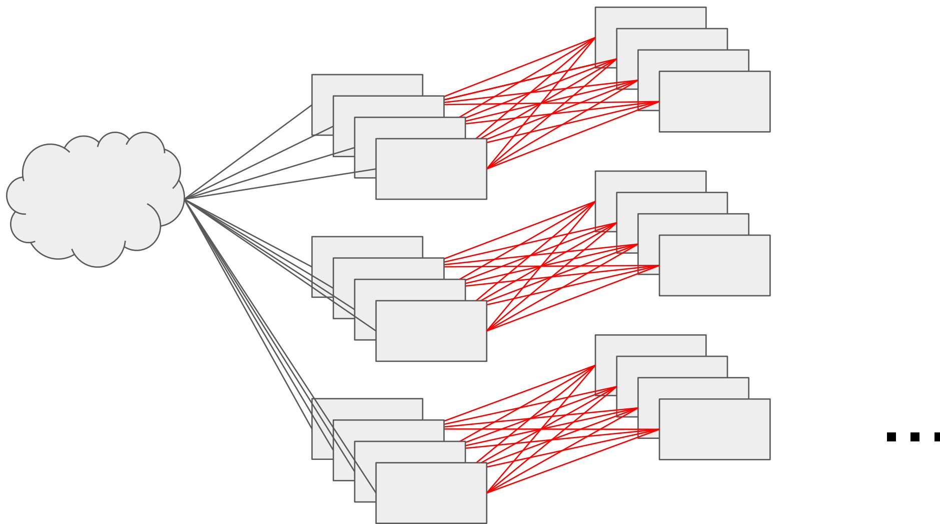



About subsets:

If every frontend maintains a connection to every backend:

- You want to avoid setup overhead and delay
- That's potentially a lot of overhead: CPU and Memory, plus a bit of network for connection health checking, and these add up
- Is it still easy to reason about?
- Will a Query of Death trample everything equally?

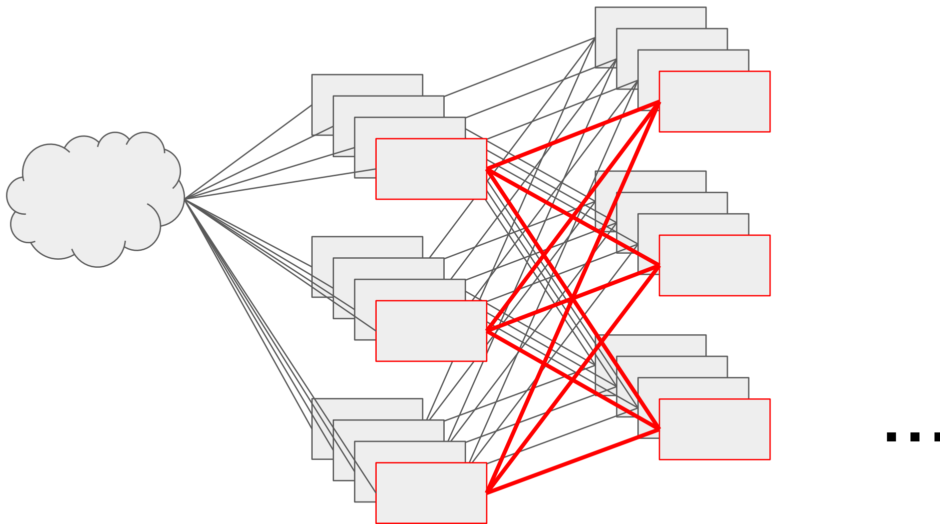
If you have a lot of backends, consider assigning subsets to clients. In this example, maybe each frontend only knows about 3 or 4 backends, not all 12.




Bad subset 1: Those servers are grouped, which implies they have some shared failure domain:

- a switch, a rack power supply, an aircon unit, a leaky roof.

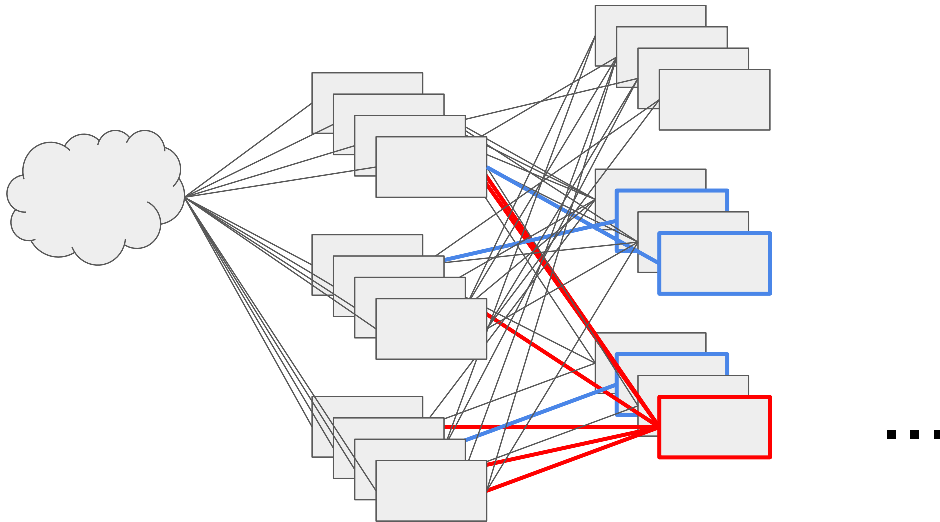
Spread things out!




Bad subset 2: Same problem, just looks different:
If a group of servers goes down, maybe because of one client sending queries of death, an entire set of clients loses all backends.

Spread things out.

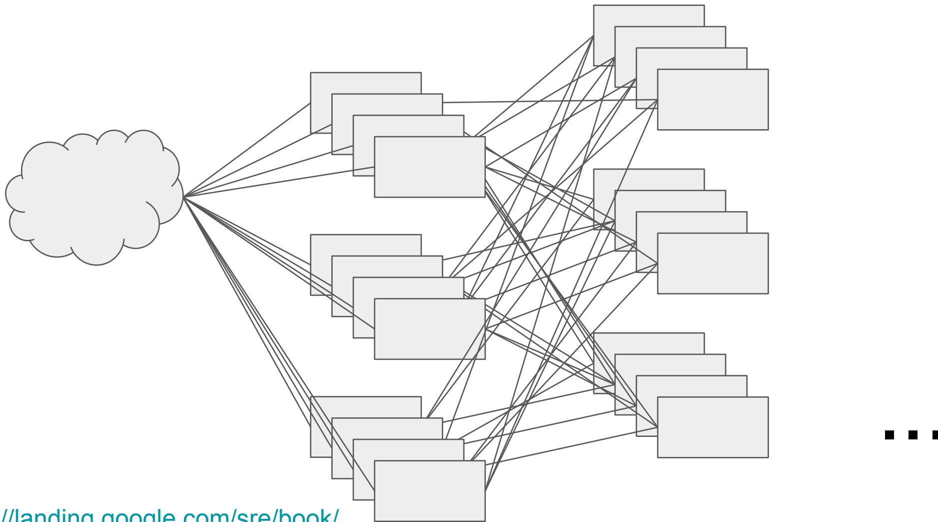
# Sharing Is Caring: Load Balancing



Bad subset 3: Random. This usually results in very uneven distribution.
In this example, one backend has 6 connections, three have just 1.

Spread things out.

## Sharing Is Caring: Load Balancing



http://landing.google.com/sre/book/

Acceptable subset: Deterministic shuffle.

Every frontend here has 3 outbound, every backend 3 inbound connections.
Things are spread out. I have nothing to highlight.

There is still pseudo-randomization!

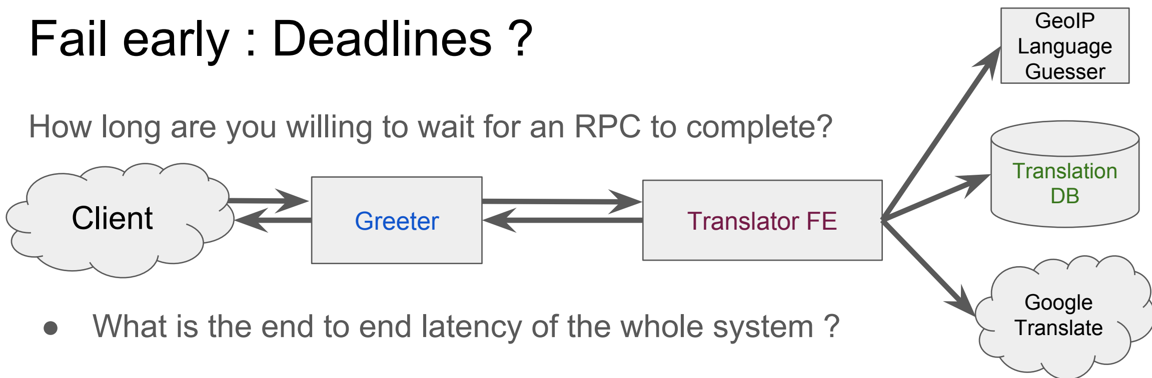
Each group of frontends shuffles the list of backends the same way, using the same PRNG seed; different group, different seed.
Then split the list evenly within the group.

## Our ~~Rules~~ *Guidelines*

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. **Fail early.**
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. "Best" effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

Now we have a greeter and it can find and send rpcs to the translator service using round-robin.
How do we make it reliable ? Fail early!




How long ? We need to set some deadlines. In order to do that, we need to ask ourselves some questions.

- What is the end to end latency of the whole system ?

Higher than your SLO. Lower than your client's timeout. Even if it's rough calculations, we need to be able to put numbers to it.
https://landing.google.com/sre/book/chapters/embracing-risk.html

- Which RPC calls are serial and which can be made in parallel ?

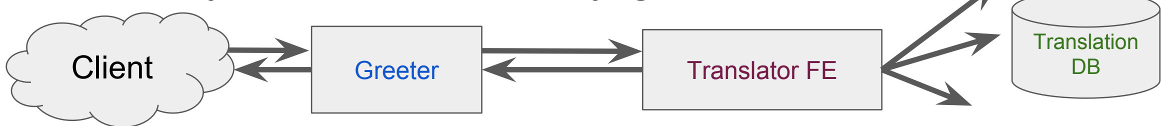
The translator has many backends, it can probably call some of them at the same time and reply to the greeter even if some of the backends fail to respond.

- Is it useful to wait for a backend response after the client deadline ?

Translator FE  might even continue to wait on any of the backend results, even after replying to the Greeter.*
Not all the work load is during the rpc call. Be mindful.

*This might be so the result can be cached, particularly if it's resource heavy request, and save time and money on the next request.

## Fail early : Deadline set by gRPC client API




1. Allocate ClientContext, set options (consider the deadline).

```
ClientContext greeter_context;
time_point deadline = system_clock::now() +
    seconds(FLAGS_translator_deadline_sec);
greeter_context.set_deadline(deadline);
greeter_context.set_wait_for_ready(false);
```

AND propagate the remaining deadline.

```
std::unique_ptr<ClientContext> trdb_context =
    FromServerContext(greeter_context);
context->set_wait_for_ready(false);
```

2. Allocate request/response and fill in request.
3. Invoke service method on stub -- gRPC forwards call to server.

```
Status status =
    stub_->Translate(&greeter_context,
                     request, &response);
```

```
Status status =
    stub_->TransDB(trdb_context.get(),
                   request, &response);
```

4. Check call status, and handle the response.

The Greeter service

- creates the client context to call the backend, but this time we set an explicit deadline.

We use a flag to set and happily adjust the deadline. If a release of a new backend causes a regression and slower requests fail the deadline.
I can adjust my Greeter using the flag, to wait longer to avoid failing, without a cherry-pick. This mitigates the issue for our users, until the backend issue can be debugged, and resolved.

We set wait for ready false, because, we have a default greetings to use if the translator is unavailable (for any reasons, overload, dead etc)

The Translator service (has it's own group of backends.)

- It creates its own client context, but copies from the "server context" passed via the gRPC system.
- By default, this will copy the deadline, as well as most other attributes (except wait_for_ready).
  (the factory method has an optional second parameter to change the behaviour.)

# Fail early : Deadlines

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    std::string prefix("வணக்கம்");
    ClientContext greeter_context;
    TranslationRequest translation_request;
    TranslationReply translation_reply;
    translation_request.set_message("Hello");
    translation_request.set_locale(request->locale());
    Status status = stub_->Translate(
        &context, translation_request, &translation_reply);
    if (status.ok()) {
      prefix = translation_reply.translation();
    }
    reply->set_message(prefix + " " + request->name());
    return Status::OK;
  }
};
```

```
ClientContext greeter_context;
time_point deadline = system_clock::now() +
    seconds(FLAGS_translator_deadline_sec);
greeter_context.set_deadline(deadline);
greeter_context.set_wait_for_ready(false);
```

Add a deadline and don't wait for it, if the backend is unavailable.

The gRPC library takes care of the communication, marshalling, unmarshalling, and deadline enforcement.

# Fail early : Deadlines

```
class TranslatorServiceImpl final : public Translator::Service {
  Status Translate(ServerContext* greeter_context,
      const TranslationRequest* request,
      TranslationReply* reply) override {
    ClientContext trdb_context;
    TranslationDBRequest trdb_request;
    TranslationDBReply trdb_reply;
    trdb_request.set_message(request->message());
    trdb_request.set_locale(request->locale());
    Status status = trdb_stub_->TransDB(
         &trdb_context, trdb_request, &trdb_reply);
    if (status.ok()) {
      reply->set_translation(trdb_reply.text());
    } else if (status.error_code() == GRPC_STATUS_NOT_FOUND) {
      reply->set_translation(request->message());
    } else {
      return status;
    }
    return Status::OK;
  }
};
```

```
std::unique_ptr<ClientContext> trdb_context =
       FromServerContext(greeter_context);
context->set_wait_for_ready(false);
```

```
Status status = trdb_stub_->TransDB(
     trdb_context.get(),
     trdb_request, &trdb_reply);
```

Only the ClientContext creation is a semantic change to propagate the deadline to the backends.

(The other edit is only to make it compile (different type).)

# Fail early : Deadlines : gRPC configuration

// The Translator service .proto file

```
enum Type {
  FORMAL = 0;
  RUDE = 1;   [deprecated = true];
  SLANG = 2;
}
message TranslationRequest {
  string message = 1;
  reserved 2;
  Type type = 3;
  string locale = 4;
}
service Translator { ... }
```

// The service_config file

```
Functionality coming soon...
     Watch this space.
```

Deadlines are set in the client API. Service owners can publish parameters in a service config.

Reminder : If you don't set a deadline, your requests can wait forever.

There are other configuration files, .proto and service_config.
https://github.com/grpc/grpc/blob/master/doc/service_config.md

# Failing : Deadlines !

Image: http.cat/504 (HTTP Status Cats, © 2011 Tomomi Imura)

- Make an informed choice of deadline.
- Client: wait until the result is no longer useful.
- Service: specify the longest default deadline you can technically support.
- If the backend is known to be unavailable, don't wait for the deadline to pass.

What do we do instead?

- Make an informed choice of deadline.

You have to understand the system to know what deadlines makes sense.
Particularly if you have a wide range of behaviour, for example  processing time required for different requests.
Reminder, edge case, Streaming RPCs we've not covered don't have deadlines.

- Clients: wait until result is no longer useful.

When under load everything will take longer, choose wisely between taking more time or not answering at all. Remember that retries probably make everything worse.

- Service: specify the longest default deadline you can technically support.

If you're a db, and your client tells you to drop a table, how long do they wait to hear back it worked?

- If the backend is known to be unavailable, don't wait for the deadline to pass.

If the backend is unavailable, you're using latency. Don't wait, use the fallback default.
Fail early!

## Our ~~Rules~~ *Guidelines*

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. Fail early.
4. **Degrade gracefully, don't guess.**
5. Measure all the things!
6. "Best" effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

Now we have a Greeter and it can find and send rpcs to the translator with round-robin.
It sets a suitable deadline.

It can tolerate some translator failures, by setting a deadline, and can return a useful default response fast if the translator is hard down.

But how reliable is the Greeter?

# Degrade gracefully ?

- How many in-flight requests – i.e. those sent but waiting on the RPC response – can the server support at a time ?
- How much work is needed for failing a request ?
- How much "health" does your service need to do something useful ?
- Can users still get a useful result if some|all of the system's backends fail?
- How does the server behave under heavy load ?

- How many in-flight requests – i.e. those sent but waiting on the RPC response – can the server support at a time ?

What happens if you get more requests?
Have a degraded mode; lower-quality & cheaper processing responses, 503's.
Measure it, (You may remember this from the opening session with Bruno from LinkedIn.
'Measured get fixed', and we'll talk about that later.)

- How much work is needed for failing a request ?

If an error is as costly to process as a success, the system is going to fail horribly.
When everything returns errors, fail fast and be cheap to run!

- How much "health" does your service needed to do something useful ?

Do all the backends need to be available, and able to reply < 200ms to be "healthy".
We'd never be 100% healthy if that were the case.

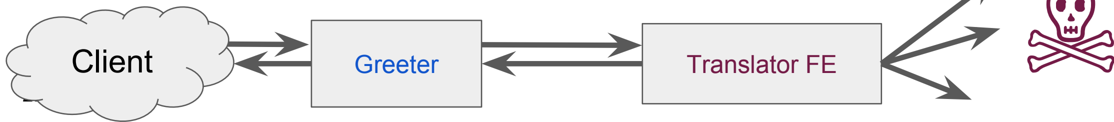
- Can users still get a useful result if some|all of the system's backends fail?

The answer you want is YES. Mitigate critical dependencies. Change the system design or the code to allow for failures, including global outages of backends.

- How does the server behave under heavy load ?

Figure out what resources limits you hit first, it could even be file descriptors. Until you run it and gather data, it's all guessing.

# Degrade gracefully ?




3. Set deadlines etc and then invoke service method on stub.

```
Status status = stub_->Translate(&context,
    translation_request, &translation_reply);
```

4. Check call status, and handle the response.

```
if (status.ok())  {
  prefix = translation_reply.translation();
} else {
  // do something useful but cheap.
  LOG_EVERY_N(ERROR, 10)
    << "Error  " << google::COUNTER << " with status "
    << status.error_code() << status.error_message();
}
reply->set_message(prefix + " " + request->name());
```

http://github.com/google/glog/

Early expiry … before you do a lot of heavy processing work.

```
if (!greeter_context->IsCancelled()) {...}
```

Especially for extremely short remaining deadlines ...return partial results from the available responses.

```
gpr_timespec time_left = gpr_time_sub(
  gpr_now(greeter_context
            ->raw_deadline().clock_type),
  greeter_context->raw_deadline()
);
if (time_left < some_threshold) {...}
```

As before, the gRPC library takes care of the communication, marshalling, unmarshalling, and deadline enforcement.
IsCancelled is true for an expired request as well as actively cancelled (by the greeter).

What if we get a query of death and all the Translator backends are dead?
Control how many in-flight request you have, and drop the ones which are already past their deadline.

What if the Translator FE doesn't respond ? (low capacity, loss of a datacenter etc).
Step 1,2 and 3 are unchanged.
Do something useful at the Greeter using a default, and log the error cheaply! Be cheap ! consider your broken behaviour, are you logging every error, synchronously ?
Recommended reading for better logging
https://github.com/google/glog/blob/master/doc/glog.html

Use the codes to understand how you want to handle the failures.
https://github.com/grpc/grpc/blob/master/doc/statuscodes.md
https://github.com/grpc/grpc/blob/master/include/grpc/impl/codegen/status.h

# Degrade gracefully ?

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    std::string prefix("வணக்கம்");
    ClientContext greeter_context;
    time_point deadline = system_clock::now() +
      seconds(FLAGS_translator_deadline_sec);
    greeter_context.set_deadline(deadline);
    greeter_context.set_wait_for_ready(false);
    TranslationRequest translation_request;
    TranslationReply translation_reply;
    translation_request.set_message("Hello");
    translation_request.set_locale(request->locale());
    Status status = stub_->Translate(
        &context, translation_request, &translation_reply);
    if (status.ok()) {
      prefix = translation_reply.translation();
    }
    reply->set_message(prefix + " " + request->name());
    return Status::OK;
  }
};
```

```
 if (status.ok())  {
  prefix = translation_reply.translation();
} else {
  // do something useful but cheap.
  LOG_EVERY_N(ERROR)
    << "Error  " << google::COUNTER << " with status "
    << status.error_code() << status.error_message();
}
reply->set_message(prefix + " " + request->name());
```

# Degrade gracefully ? Healthy ?

| | |
|---|---|
| *Gabe: Hi* | `GreeterClient greeter(grpc::CreateChannel(...` |
| *Gráinne: Hi* | |
| *Gabe: Can I ask you a question ?* | `Check(HealthCheckRequest)` |
| *Gráinne: Yes* | |
| *Gabe: Hi, what's 'Hello Gráinne in Irish?* | `Translate(&context, request, &response)` |
| *Gráinne: …* | |

"Health" is a misnomer and usually under specified.

```
greeter_context.set_wait_for_ready(false); OR greeter_context.set_wait_for_ready(true);
```

gRPC let's you set up a concept of "heatlhy" for rpc requests.
BUT Global services are never 100% "healthy".

*/walk thru*

If you need to send an RPC to know if you can send an RPC, you might as well just send the request and make one call. (Edge cases apply.)

What might care about "health"?
System components which need global state of the service, or availability..
In-band health checks usually don't help, and getting them wrong leads to avoidable system outages.

Know your "critical" dependencies i.e. backends you _need_ to respond at all.
The more of them you have, the more hard-down failures in your service.

A global load balancer, picking between cluster will care about how much capacity / available backends there are in a region.
I should send more of India's traffic to Singapore instead of Taiwan if Singapore has more capacity to serve the clients.

https://github.com/grpc/grpc/blob/master/doc/health-checking.md

## Degrade gracefully ?

Startup. Only accept connections once you can usefully respond.

Serving. Don't die. Assertions only on startup.

Shutdown. Close your port to new requests, finish inflight requests.

Remember you process will change state, and you want to handle that. Think about the life cycle.

Startup.

- Don't accept connections if critical dependencies are unavailable, or configuration is wrong.
- Situations when it's actively dangerous to serve.

Serving.

- Removing capacity from a system under load is a fast way to bring it down hard.
- Accept 2x your normal requests, in a degraded way if possible, don't die.
- After that, shed load, expire early, etc.
- A greeter component might not be "healthy" but so too could all the other greeters, an individual greeter server doesn't know the global state.

Shutdown.

- Finish accepted requests. Wait at least the longest default deadline for your services before terminating.

# Degrade gracefully don't guess !

Image: "We Can Do It" poster, 1943 (Public Domain)

- Load test; push it 2x, 5x,10x.
- Have a degraded mode; lower-quality & cheaper processing responses, 503's.
- When everything returns errors, fail fast and be cheap!
- Know your "critical" dependencies, i.e. backends needed so you can respond at all.
- Mitigate critical dependencies.

context.set_wait_for_ready(true);  // only for critical backends, when you can do nothing useful for the client.

Anything left is a functional critical dependency, for example getting a decryption key. Have a plan for what to do if your service can't mitigate a critical dependency by itself.

https://landing.google.com/sre/book/chapters/addressing-cascading-failures.html and in particular #xref_cascading-failure_load-shed-graceful-degredation
https://landing.google.com/sre/book/chapters/handling-overload.html

Image : https://en.wikipedia.org/wiki/We_Can_Do_It!#/media/File:We_Can_Do_It!.jpg

# Our ~~Rules~~ *Guidelines*

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. Fail early.
4. Degrade gracefully, don't guess.
5. **Measure all the things!**
6. "Best" effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

## Measure all the things!

- What's happening?
- Why is it happening?
- Who made it happen?
- Will it keep happening?

"What gets measured, gets fixed"
(Bruno from LinkedIn, Monday morning)

We want to answer these questions.

## Measure all the things! what's happening?

### Server things!

- CPU rate
- Memory usage
- Network IO
- Disk IO (ops & MB)
- Startup time

### Service things!

- Request rate
- Success rate
- Error rate by type
- Latency
- Logging behaviour

Server things: These are coarse, but absolute: If they're bad, they're very bad.
Server things feed capacity plans and alerting.

CPU, Memory, Network and Disk: These are the total cost to run your service.
All of them have limits.
Know these, and watch them over time.

Your startup time is your service recovery time *after* you have fixed everything.
Know this. Watch this over time.

Service things: These can tell you what your service is doing, and how well it is doing it.
This is where you compute your performance to compare to your SLO.

Your request rate is the denominator for your capacity planning. How many queries are you actually getting?

Of course, you need to know whether you're replying successfully.
One way to succeed, many ways to fail - 500 vs 404.

Latency is probably important - fast is better than slow. Your SLA tells you when it's fast enough - but only for success.

Logging behaviour is often forgotten, but worth noting: I/O operations are expensive, so is space (eventually).
Mentioned earlier.

https://landing.google.com/sre/book/chapters/monitoring-distributed-systems.html

# Measure all the things! What's happening?

Measure the server.

```
import "github.com/grpc-ecosystem/go-grpc-prometheus"
...
    myServer := grpc.NewServer(
        grpc.StreamInterceptor(grpc_prometheus.StreamServerInterceptor),
        grpc.UnaryInterceptor(grpc_prometheus.UnaryServerInterceptor),
    )
    myservice.RegisterMyServiceServer(s.server, &myServiceImpl{})
    grpc_prometheus.Register(myServer)
    http.Handle("/metrics", prometheus.Handler())
...
```

http://github.com/grpc-ecosystem/go-grpc-prometheus

This is probably the easiest example: using Go, and Prometheus. The contributed library uses the RPC intercept capability to gather metrics, and they are then available via HTTP for prometheus to collect.

https://github.com/grpc-ecosystem/go-grpc-prometheus

But! If a remote procedure call is sent, and no timely reply is received, do the server metrics matter?

# Measure all the things! What's happening?

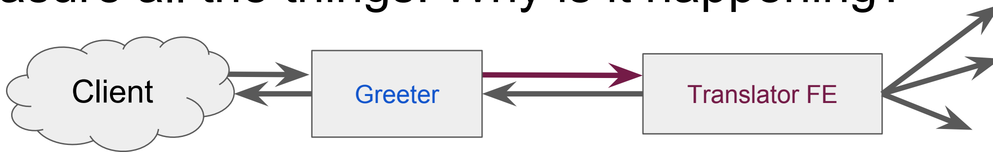
Measure the client!

```
import "github.com/grpc-ecosystem/go-grpc-prometheus"
...
   clientConn, err = grpc.Dial(
       address,
           grpc.WithUnaryInterceptor(UnaryClientInterceptor),
           grpc.WithStreamInterceptor(StreamClientInterceptor)
   )
   client = pb_testproto.NewTestServiceClient(clientConn)
   resp, err := client.PingEmpty(s.ctx, &myservice.Request{Msg: "hello"})
...
```

http://github.com/grpc-ecosystem/go-grpc-prometheus

"In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match. This means that, for example, you could have an RPC that finishes successfully on the server side ("I have sent all my responses!") but fails on the client side ("The responses arrived after my deadline!"). "
http://www.grpc.io/docs/guides/concepts.html#rpc-termination

# Measure all the things! Why is it happening?




1. Save the "before" state.
```
struct timespec start;
clock_gettime(CLOCK_REALTIME, &start);
```
2. Create Request and Response, **call the backend**, check the status, prepare your reply.

3. Report the delta.
```
struct timespec finish;
clock_gettime(CLOCK_REALTIME, &finish);
double diff_in_s =
  (finish.tv_sec + finish.tv_nsec / 1000000000.0) -
  (start.tv_sec + start.tv_nsec / 1000000000.0);
RecordTimeTaken("SayHello", diff_in_s);
```

1. Save the "before" state.
```
struct timespec start_ts;     // translator service
clock_gettime(CLOCK_REALTIME, &start_ts);
```
2. Create Request and Response, **call the backend**... check the status, prepare your reply.

3. Report the delta.
```
struct timespec finish_ts;    // translator service
clock_gettime(CLOCK_REALTIME, &finish_ts);
double ts_diff_in_s =
  (finish_ts.tv_sec + finish_ts.tv_nsec / 1000000000.0) -
  (start_ts.tv_sec + start_ts.tv_nsec / 1000000000.0);
RecordTimeTaken("SayHello_Translate", ts_diff_in_s);
```

Example: Measure latency: End time minus start time.

Remember how everything is a stack?

You took as long as you took because you did what you did.

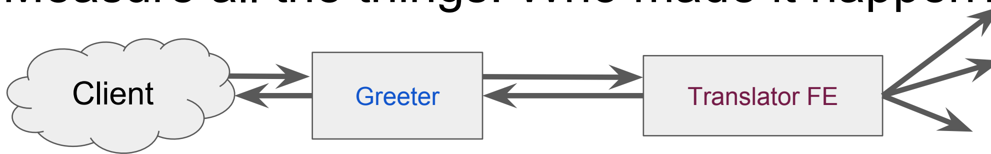
# Measure all the things! Why is it happening?

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    struct timespec start;
    clock_gettime(CLOCK_REALTIME, &start);
    std::string prefix("வணக்கம்");
    ClientContext greeter_context;
    // ...
    translation_request.set_message("Hello");
    translation_request.set_locale(request->locale());
    struct timespec start_ts;
    clock_gettime(CLOCK_REALTIME, &start_ts);
    Status status = stub_->Translate(
        &greeter_context, translation_request, &translation_reply);
    struct timespec finish_ts;
    clock_gettime(CLOCK_REALTIME, &finish_ts);
    double ts_diff_in_s = MyTimeDiff(finish_ts, start_ts);
    RecordTimeTaken("SayHello_Translate", ts_diff_in_s);
    if (status.ok()) {
      prefix = translation_reply.translation();
    } else {
      // do something useful but cheap.
      LOG_EVERY_N(ERROR)
        << "Error  " << google::COUNTER << " with status "
        << status.error_code() << status.error_message();
    }
    reply->set_message(prefix + " " + request->name());
    struct timespec finish;
    clock_gettime(CLOCK_REALTIME, &finish);
    double diff_in_s = MyTimeDiff(finish_ts, start_ts);
    RecordTimeTaken("SayHello", diff_in_s);
    return Status::OK;
  }
};
```

# Measure all the things! Why is it happening?

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    struct timespec start;
    clock_gettime(CLOCK_REALTIME, &start);
    std::string prefix("வணக்கம்");
    ClientContext greeter_context;
    // ...
    translation_request.set_message("Hello");
    translation_request.set_locale(request->locale());
    struct timespec start_ts;
    clock_gettime(CLOCK_REALTIME, &start_ts);
    Status status = stub_->Translate(
        &greeter_context, translation_request, &translation_reply);
    struct timespec finish_ts;
    clock_gettime(CLOCK_REALTIME, &finish_ts);
    double ts_diff_in_s = MyTimeDiff(finish_ts, start_ts);
    RecordTimeTaken("SayHello_Translate", ts_diff_in_s);
    if (status.ok()) {
      prefix = translation_reply.translation();
    } else {
      // do something useful but cheap.
      LOG_EVERY_N(ERROR)
        << "Error  " << google::COUNTER << " with status "
        << status.error_code() << status.error_message();
    }
    reply->set_message(prefix + " " + request->name());
    struct timespec finish;
    clock_gettime(CLOCK_REALTIME, &finish);
    double diff_in_s = MyTimeDiff(finish_ts, start_ts);
    RecordTimeTaken("SayHello", diff_in_s);
    return Status::OK;
  }
};
```

## Measure all the things! Who made it happen?




```
ClientContext context;
context.AddMetadata(
    "call_site", "Greeter::SayHello");
Status status =
    stub_->Translate(&context, req, &resp);
...
```

```
TranslatorImpl::Translate(
    ServerContext* ctx, ...) override {
  const auto& metadata =
      ctx->client_metadata();
  auto iter = metadata.find("call_site");
  LOG(INFO) << "Translation request by: "
      << (iter == metadata.end() ?
          "unknown" : iter->second);
  ctx->AddInitialMetadata(
      "translation_mode", "offline");
```

In addition to using the authenticated user, you would like to know the calling service - if it's an experimental or QA environment, maybe you can ignore their errors.

Ideally, you would also like to know the calling site - not just your immediate client, but its client and so on up the stack. For example, attributing particularly expensive queries to some remote frontend without this facility is… hard.

In gRPC, you can add arbitrary metadata; a similar facility exists in the other direction.

Google is currently working on open-sourcing the "census" infrastructure that allows automatic tracing of distributed queries.
A lot of the low-level code is already in github, but the public APIs are not yet complete.

https://github.com/grpc/grpc/blob/master/src/core/ext/census/README.md - we hope to be expanding on this in a future talk.

## Measure all the things! Will it keep happening?

Capacity planning, SLA negotiations, Troubleshooting...

| one | | | two | | | three | | | four | |
|---|---|---|---|---|---|---|---|---|---|---|
| x | y | | x | y | | x | y | | x | y |
| 10.0 | 8.04 | | 10.0 | 9.14 | | 10.0 | 7.46 | | 8.0 | 6.58 |
| 8.0 | 6.95 | | 8.0 | 8.14 | | 8.0 | 6.77 | | 8.0 | 5.76 |
| 13.0 | 7.58 | | 13.0 | 8.74 | | 13.0 | 12.74 | | 8.0 | 7.71 |
| 9.0 | 8.81 | | 9.0 | 8.77 | | 9.0 | 7.11 | | 8.0 | 8.84 |
| 11.0 | 8.33 | | 11.0 | 9.26 | | 11.0 | 7.81 | | 8.0 | 8.47 |
| 14.0 | 9.96 | | 14.0 | 8.10 | | 14.0 | 8.84 | | 8.0 | 7.04 |
| 6.0 | 7.24 | | 6.0 | 6.13 | | 6.0 | 6.08 | | 8.0 | 5.25 |
| 4.0 | 4.26 | | 4.0 | 3.10 | | 4.0 | 5.39 | | 19.0 | 12.50 |
| 12.0 | 10.84 | | 12.0 | 9.13 | | 12.0 | 8.15 | | 8.0 | 5.56 |
| 7.0 | 4.82 | | 7.0 | 7.26 | | 7.0 | 6.42 | | 8.0 | 7.91 |
| 5.0 | 5.68 | | 5.0 | 4.74 | | 5.0 | 5.73 | | 8.0 | 6.89 |

Here we have some data sets.

Let's use our monitoring system, or favourite number library or tool, to summarise them.

# Measure all the things! Will it keep happening?

| Property | Value | Accuracy |
| --- | --- | --- |
| **Mean** of *x* | 9 | exact |
| **Sample variance** of *x* | 11 | exact |
| Mean of *y* | 7.50 | to 2 decimal places |
| Sample variance of *y* | 4.125 | plus/minus 0.003 |
| Correlation between *x* and *y* | 0.816 | to 3 decimal places |
| **Linear regression line** | $y = 3.00 + 0.500x$ | to 2 and 3 decimal places, respectively |

That's handy. They all seem to be behaving the same.
Especially that last line looks like a great growth projection.

# Measure all the things! Will it keep happening?

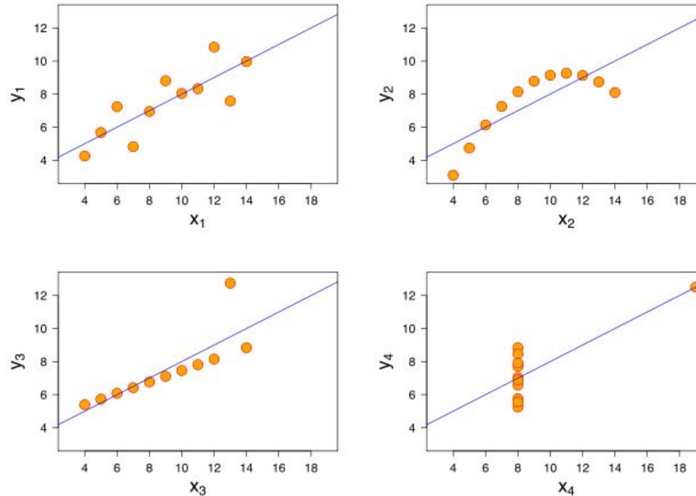



Image: Wikipedia (Anscombe.svg: Schutz Derivative works of this file:(label using subscripts): Avenue) (CC-BY-SA 3.0)

Well, maybe not.

These datasets are Anscombe's Quartet, specifically designed to show this.

Graph everything you measure.

The math doesn't lie, but that doesn't mean it's answering your question.

Measure all the *important* things!

- What's happening:
  - Gather server metrics as well as service
  - Break data out by type
- Why is it happening:
  - Group your backend metrics with your incoming metrics
- Who made it happen:
  - Group metrics by caller: User, Call site, Service, ...
- Will it keep happening:
  - Maybe.
  - Do the math, but check the graphs!

- What's happening
- Why is it happening
- Who made it happen:
  Don't try to keep a cross-product table in memory.
  Feed your logs system.
  When the data is gone, it's gone. Don't drop it until you must.
- Will it keep happening: Machine Learning, Curve Fitting, Numerology,.whatever you want to use.

Which monitoring tools you use, is up to you. But use them, and take care of your time series.
Prometheus is a third-party open-source system that has bindings for many languages. By all means, use RRDtool, if that works for you (but beware of downsampling turning one of the datasets into another!)

# Our ~~Rules~~ *Guidelines*

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. Fail early.
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. **"Best" effort isn't.**

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

Our rapid fire round.

gRPC isn't going to solve all your service problems. This isn't a silver bullet.
So here are 3 top answers given to solve all your services problems,.. Which you should question !

# "Best" effort isn't ?

Frequently given answers :

*QoS makes important things faster.*

*Caching makes it faster.*

*It definitely won't take longer than X.*

Frequently missing question :

*Oh really ?*

We like caching; it saves the trouble of re-computing a result we already know. That's good, right?

QoS here is any kind of priority tagging for requests. Maybe you have different server pools for different clients, maybe your queries contain an urgency/importance indicator, maybe you automatically process them in different queues based on the remote user's authentication. That allows your important requests to be handled faster, right?

We talked about deadlines earlier. If you set them, you know the maximum amount of time you'll be processing a request, right? That'll keep everything deterministic, right?

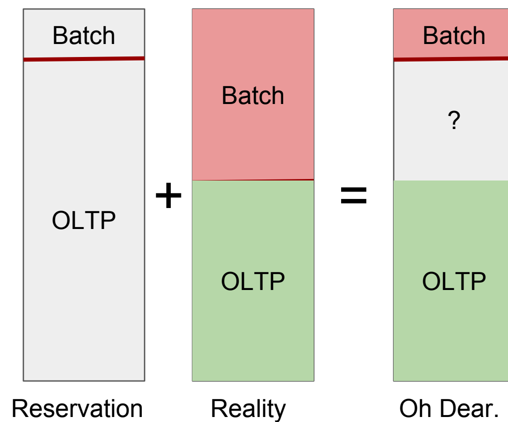
# "Best" effort isn't ? QoS makes it faster!

Image: (modified from) Untitled
by Phelyan Sanjoin (flickr: phelyan), CC-BY 2.0

Maybe. You can suffer priority inversion, or just poor utilisation. Either way, the result may or may not be what you want.

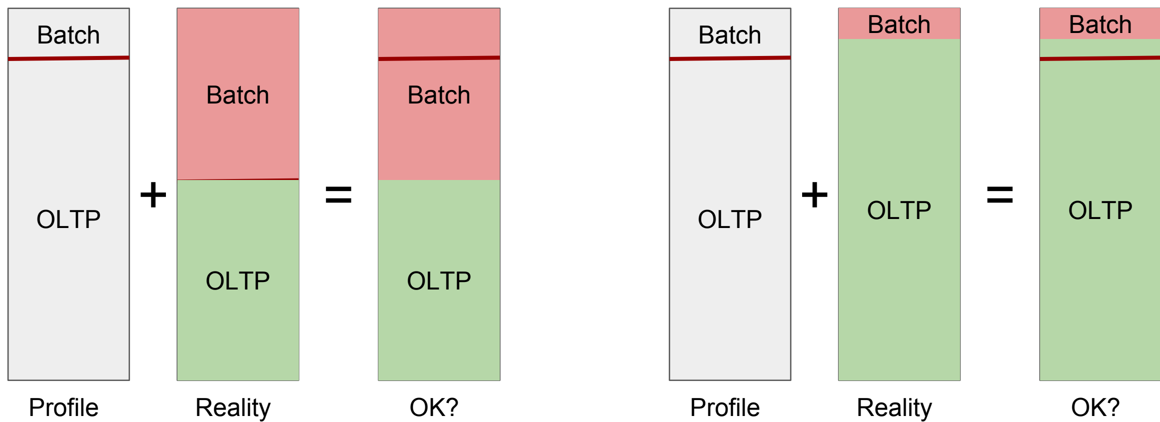
Even in the picture, the empty queue might be fine: if that request takes a long time to handle, queueing anything behind it may be a terrible choice.
The long queue is probably not good, but is it a problem? If this was a burst of quick requests, maybe not.

## "Best" effort isn't ? QoS makes it faster!




To reduce starvation and inversion, you may want to have profiles, not strict pools.

If 90% of your capacity is reserved for OLTP traffic, 10% for batch, then if your traffic mix is 60% OLTP and 40% batch, your batch requests are being throttled while your service is partly idle.




You can share the spare capacity.

Maybe now your small, low-latency request is enqueued behind a big batch query that was used to backfill.

Which brings us back to the previous topic: What is happening? Why is it happening?

You need to monitor, tune, and keep monitoring.

# "Best" effort isn't ? Caching makes it faster!

Old Problem: It's too slow.

New Problem: Cache Expiry.

Other new problem: Cache Revalidation.

Unsolved problem: Negative Caching.

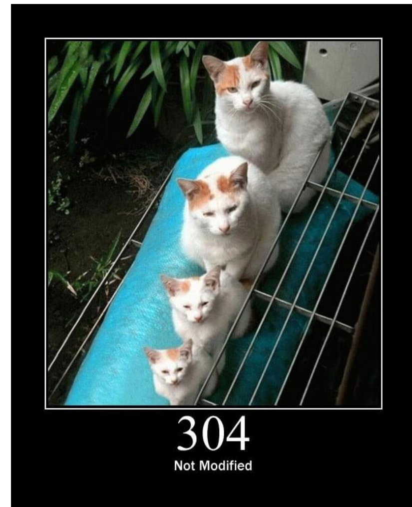



Image: http.cat/304 (HTTP Status Cats, © 2011 Tomomi Imura)

How do you manage your cache? Least-Recently-Used sounds great, but ignores whether a given result is particularly expensive to compute. Or particularly expensive to store, it could be big.

If you have some result in your cache, do you know it's still correct? Eventual Consistency may or may not be enough, especially interactively.
When you've made a payment from your bank account, you want to see the new balance immediately, not a cached one.

We often use caches to help against hotspots - the most frequently accessed data.
Maybe celebrities' avatar images on a social media site.
What if they don't have one? Will every request go to disk?

## "Best" effort isn't ? It won't take longer than X!

- Does it have an SLO?
- Does it have deadlines ?
  - All the way down the stack…
- Do you understand all your dependencies?

  No really, do you?
  - if your DNS goes away or is slow?
  - if your log buffer is full?
- What's your mitigation ? Is it automatic or human ?

DNS: synchronous queries in your code? In something your code uses?

Mitigation: Unexpected slowness? Stop the bleeding, then debug.

# "Best" effort isn't !

- QoS **may** allow **well-behaved** important things to go faster.

  It also needs careful management, and may decrease your utilisation.

- Caching **may** make it faster.

  It also makes everything more complicated, and maybe less correct.

- It **maybe** won't take longer than...

  Deadlines help, but not everything supports them.

# Our Rules *Guidelines*

## Be a good Jenga block

1. Everything is (secretly) a stack.
2. Sharing is caring.
3. Fail early.
4. Degrade gracefully, don't guess.
5. Measure all the things!
6. Best effort isn't.

Image: (modified from) Jenga
by Chris (flickr: chrish_99), CC-BY 2.0

# What next ?

Where to send your gRPC feedback ?

File features requests & issues https://github.com/grpc/grpc/issues

That's great ! but I've still got questions ?

Coming soon : more grpc.io examples, plus SREcon EMEA.

I've got a question like right now ? Shoot !