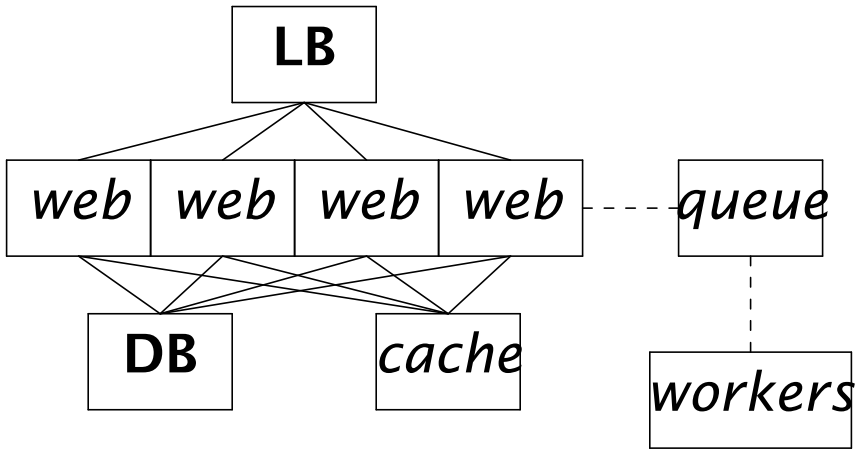
Hints for Service Oriented Architectures

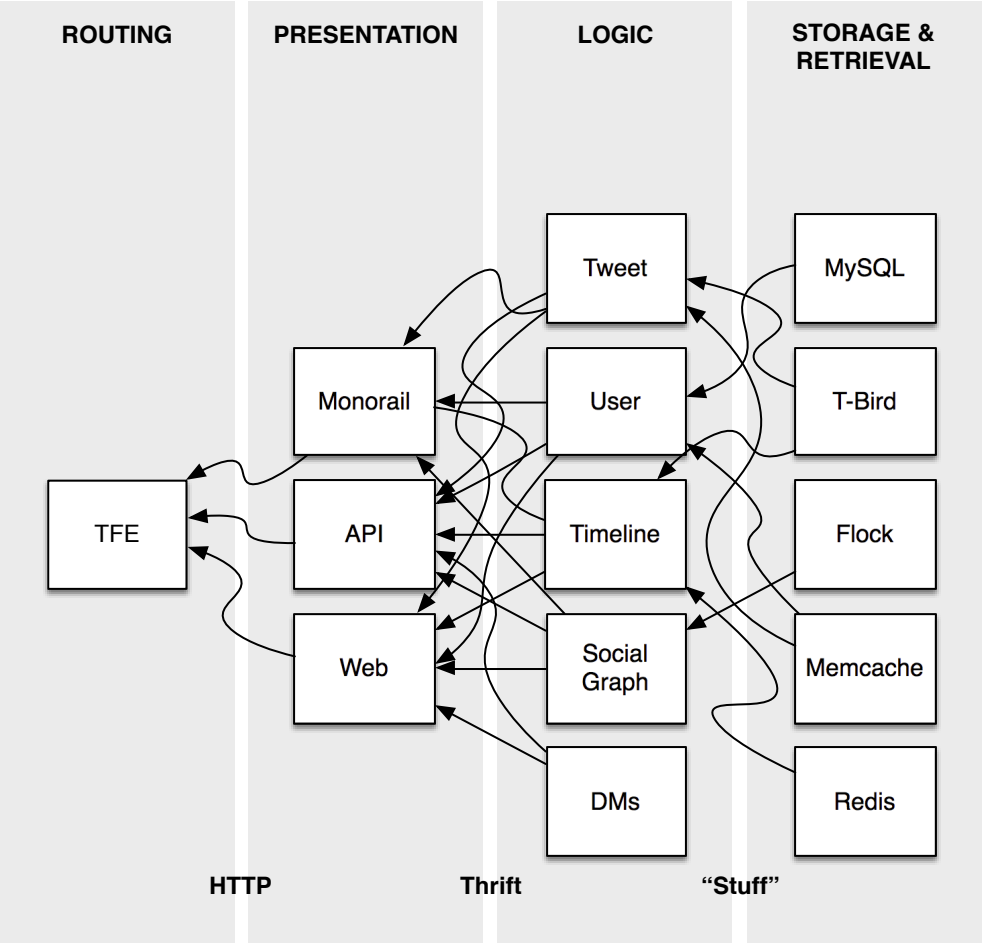
Marius Eriksen — @marius
Twitter Inc.

We went from this (circa 2010) …



(We can talk for a long time about what is wrong with this architecture; but also about what is right.)

… to this (circa 2015)



Problems, circa 2010

- Organizational growth made it difficult to share the development environment. *Deploys* became a major bottleneck.
- Ruby/RoR isn't well suited for development by large teams.
- The application was getting prohibitively expensive to run.

We invested in infrastructure:

- We improved the Ruby runtime: *Kiji* introduced a new garbage collector.
- We put a lot of effort into making deploys scale well; weed out interfering changes, etc.

But ultimately, we were working against the grain; we needed something more drastic.

Why SOAs in the first place?

- *Decoupling*. They enable independent development, deployment, operations. Individual teams can move faster; teams can alter their implementation(s) without massive coordination (c.f., library upgrades).

- *Independent scaling and distribution*. Expensive parts of the system can be scaled independently; services can occupy different failure domains; resources can be differentially allocated.

- *Isolation*. Systems are made more robust for the same reason protected memory makes systems more robust.

… **why not?**

It makes everything else more complicated.

- A new model of development and operations (e.g., try to debug an SOA);
- more, new infrastructure;
- a new failure model.

Make sure you're solving a real problem before going down this path.

Setting

A datacenter is a really crappy computer; they:

- have deep memory hierarchies,
- exhibit partial failures,
- have dynamic topologies,
- are heterogeneous,
- are connected via asynchronous networks,
- make lots of room for operator error,
- and are very complex.

But, they're what we've got. We need to gain reliability, safety, and efficiency through software.

Hints

1. The datacenter is your computer
2. Embrace RPC
3. Compute concurrently
4. Multiplex HTTP
5. Define and use SLOs
6. Abstract destinations
7. Measure liberally
8. Profile and trace systems in situ
9. Use and evolve data schemas

10. Delay work
11. Shunt state
12. Hide caches
13. Push complexity down
14. Think end-to-end
15. Expect failure, and fail well
16. Demand less
17. Beware hot shards and thundering herds

18. Embrace Conway's law
19. Deploy changes gradually
20. Keep a tab
21. Plan to deprecate

1. The datacenter is your computer

… and it's a *crappy* one.

No more *machines!* Your unit of deployment is, e.g.,

- a VM;
- a Docker container;
- a statically linked binary (or, in Java, a JAR file).

The important thing is that the abstraction does not *tie the hands of the implementor*—in this case, the system's operator.

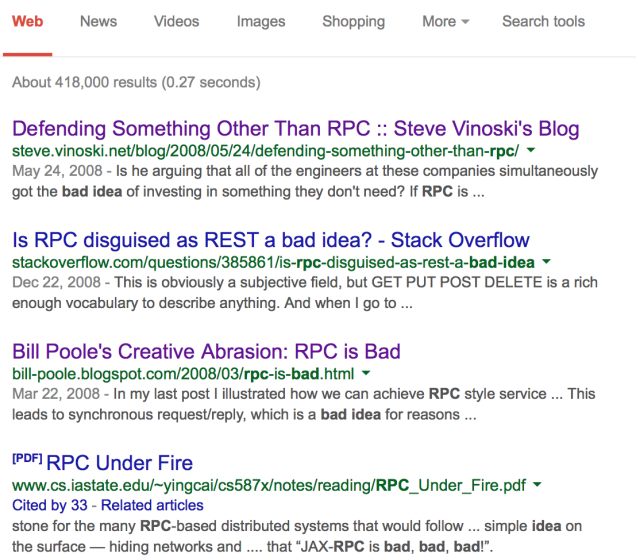
Functional units—the computational basis—should be:

- location independent;
- self-contained;
- replicable.

Use a cluster scheduler (Mesos/Aurora, Kubernetes, ..)

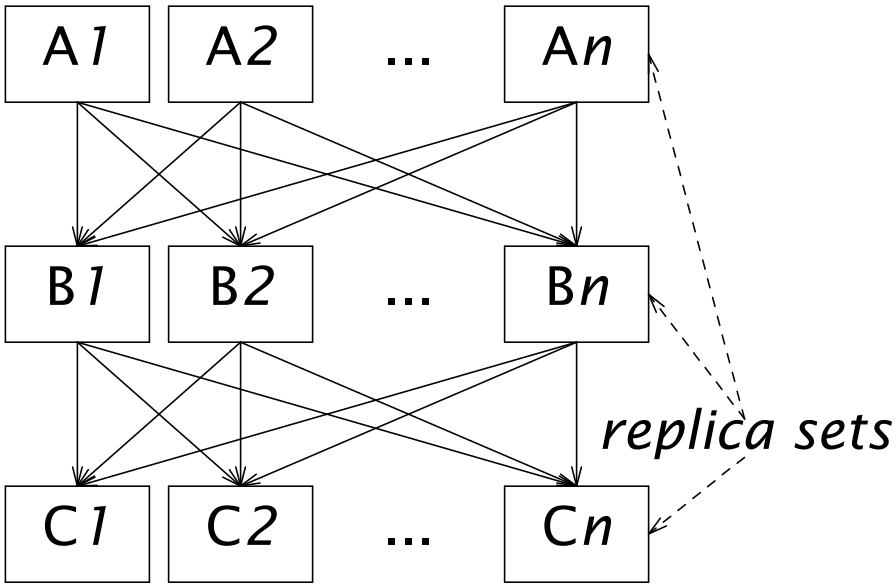
2. Embrace RPC

RPC has gotten a bad name, for the wrong reasons.



These articles attack an *outdated* idea of RPC, without acknowledging the benefits. (Of which there are many.)

Most SOAs end up looking like this; it fits the RPC model very well.



- load balancing
- retry, timeout policies
- higher-level policies (e.g., retrying, time-outs)
- failure accrual
- concurrent programming model

(See Finagle.)

3. Compute concurrently

> Concurrent programs wait faster.
> —Tony Hoare

Concurrency is the natural way of things in distributed systems. Wait only where there is a data dependency.

Your programming model can help.

```
def querySegment(id: Int, query: String)
  : Future[Result]
def search(query: String): Future[Set[Result]] = {
  val queries: Seq[Future[Result]] =
    for (id <- 0 until NumSegments) yield {
      querySegment(id, query)
    }
  Future.collect(queries) flatMap {
    results: Seq[Set[Result]] =>
    Future.value(results.flatten.toSet)
  }
}
```

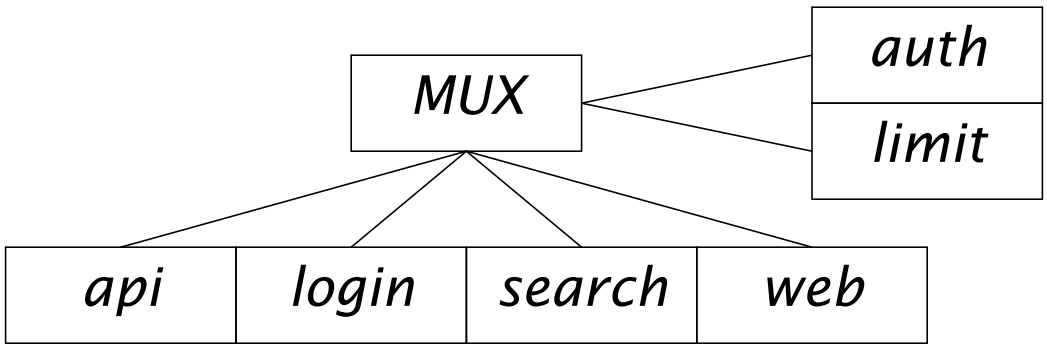
Taking it further with *queries*.

```
for {
  tweetIds <- timelineIds(User, id)
  tweets <- traverse(tweetIds) { id =>
    for {
      (tweet, sourceTweet) <- TweetyPie.getById(id)
      (user, sourceUser) <- Stitch.join(
        getByUserId(tweet.userId),
        traverse(sourceTweet) { t =>
          getByUserId(t.userId) } )
    } yield (tweet, user, sourceTweet, sourceUser)
  }
} yield (tweets)
```

4. Multiplex HTTP

An *HTTP reverse proxy* enables multiplexing a single namespace; also:

- Authenticates traffic;
- performs rate limiting;
- sanitizes HTTP;
- maintains traffic stats; etc.



(This was also a crucial piece of our migration puzzle.)

5. Define and use SLOs

Service level objectives define the target level of service through:

- Latency expectations, e.g., *p50*, *p90*, *p99*;
- success rates, e.g., ‘‘five nines’’;
- throughput, e.g., 10kQPS.

SLOs are part of your *API*. They also turn out to be a remarkably good way to parameterize your systems, and a way to introduce dynamic control.

6. Abstract destinations

Naming and service discovery takes a front seat in SOAs. They provide the means to glue together components. (Just as names and linkers do in a local model.)

e.g., use ZooKeeper to register host lists; clients listen to changes, and reconfigure themselves dynamically.

Must be highly resilient; service discovery is an *end to end problem*.

One step further: logical destinations

Wily is a logical naming system that allows *late binding*. E.g.,

```
/s/user
```

Can bind to whatever makes sense in the environment.

7. Measure liberally

Metrics are usually your only convenient view into what's going on in the world. Without them you are *blind*.

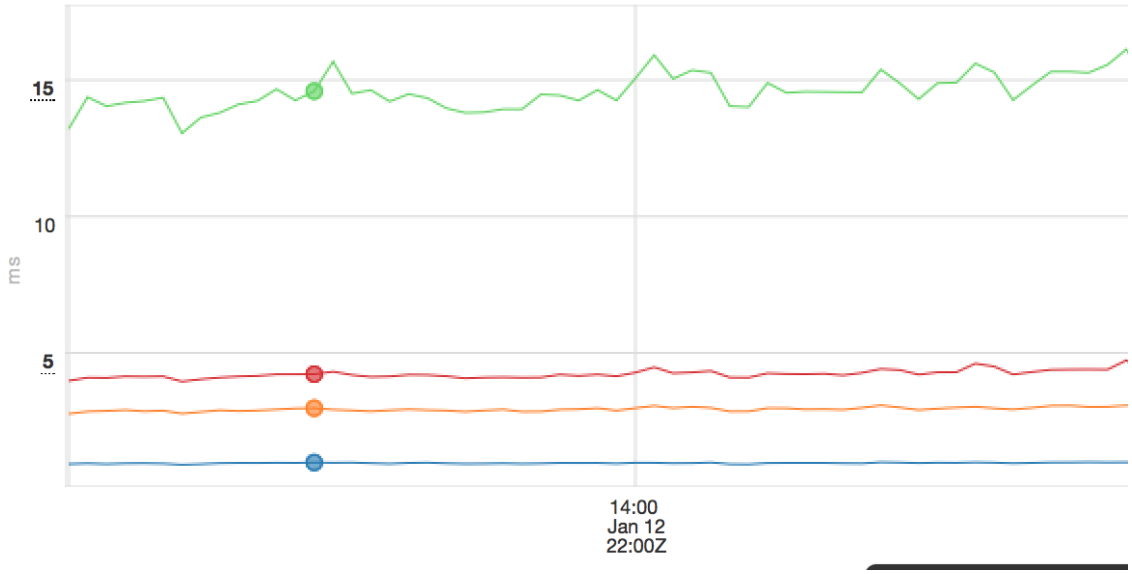
```
Stats.add("request_latency_ms", duration)
```

```
% curl http://.../admin/metrics.json
    "request_latency_ms": {
      "average": 1,
      "count": 124909591,
      "maximum": 950,
      "minimum": 0,
      "p50": 1,
      "p90": 3,
      "p95": 5,
      "p99": 19,
      "p999": 105,
      "p9999": 212,
      "sum": 222202958
    },
```

Stats are *aggregated* across:

- Clusters;
- datacenters; and also
- globally.

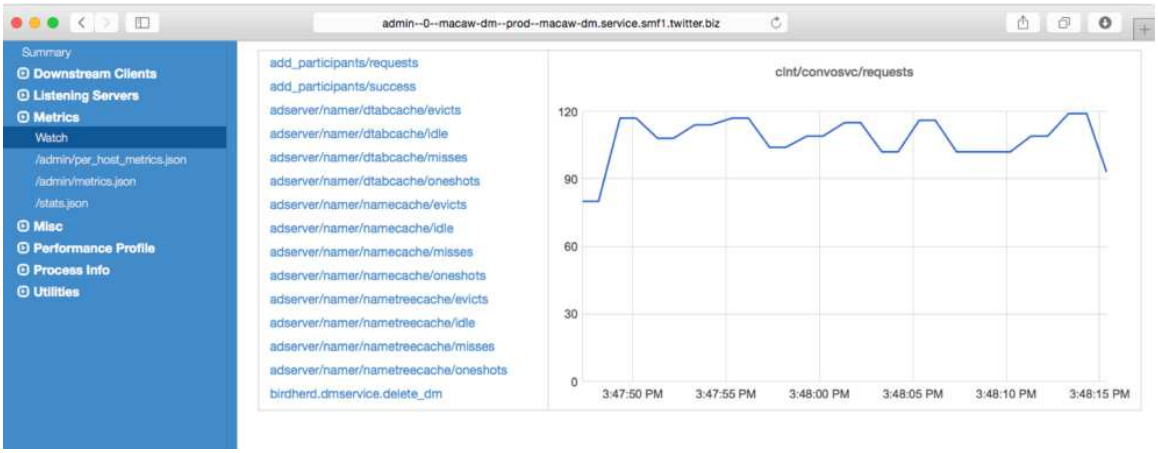
```
avg(ts(AVG, Gizmoduck,
  Gizmoduck/request_latency_ms.p{50,90,
    99,999}))
```



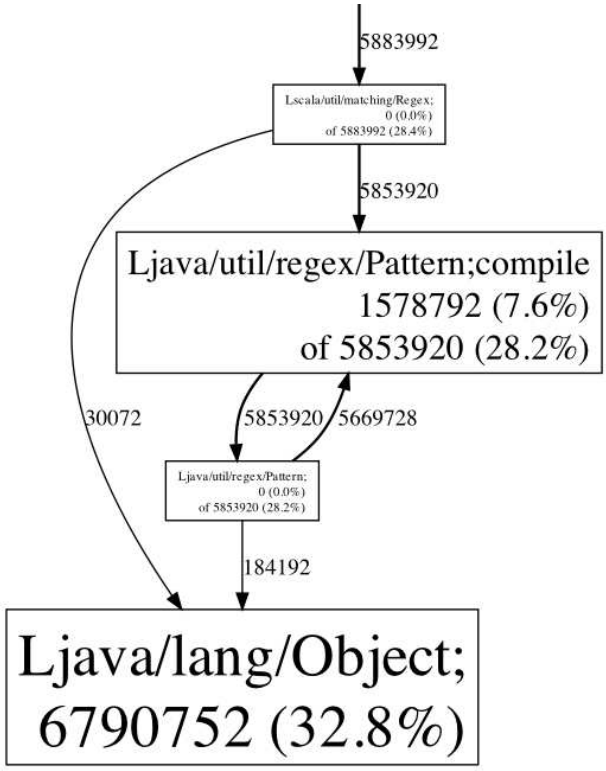
Pay special attention to *outliers*, and look at *distributions*.

8. Profile and trace systems in situ

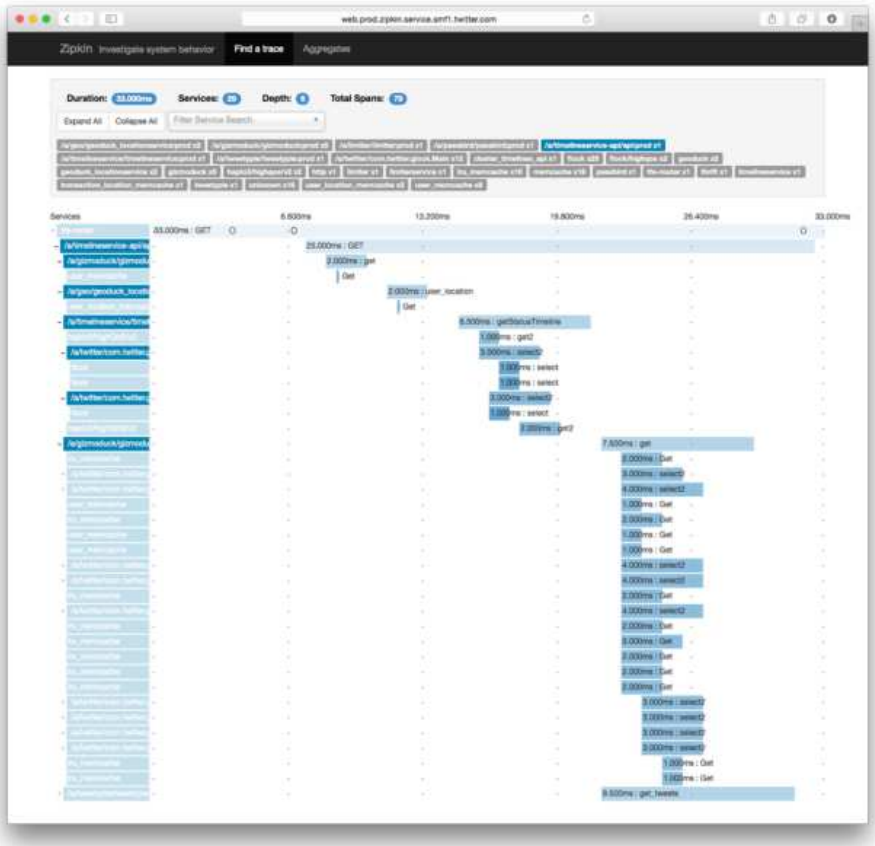
```
% open http://admin.0.userservice.smf1/
```



```
% curl -O http://.../pprof/heap
% pprof heap
```



```
% curl -H 'X-Twitter-Trace: 1' -D - \
       http://twitter.com/...
HTTP/1.1 ...
X-Trace-Id: f5e0399fa51b
...
% open http://go/trace/f5e0399fa51b
```



9. Use and evolve data schemas

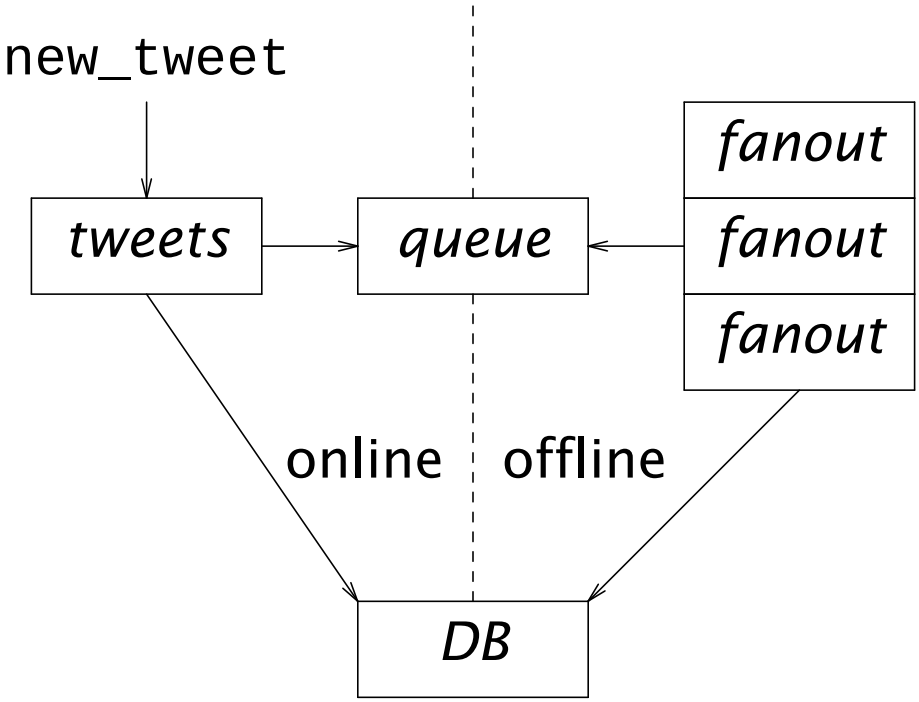
By definition, SOAs imply that data crosses process boundaries. We now need language agnostic means to serialize data. Examples include:

- Apache/Facebook Thrift;
- Google's protocol buffers;
- Apache Avro;
- Microsoft Bond;
- JSON.

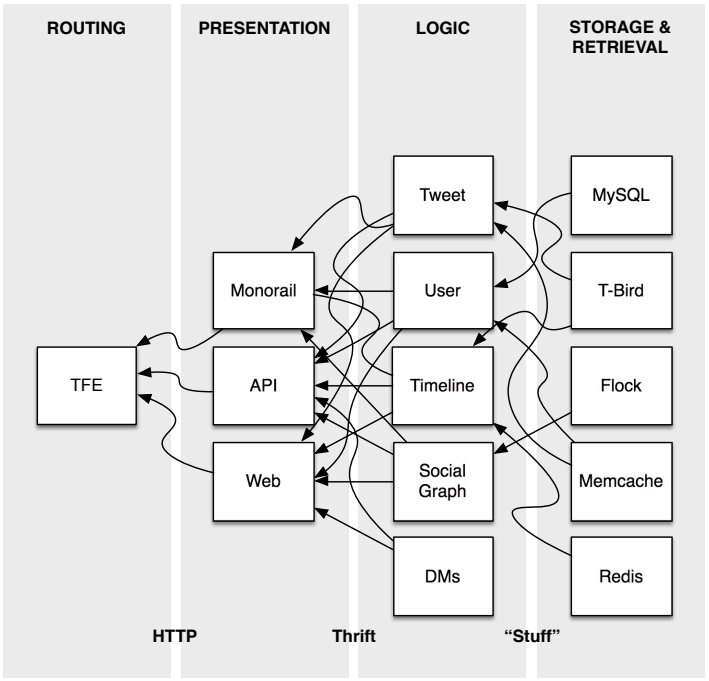
Of these, only JSON does not employ schemas. This makes it relatively difficult to evolve, and also foregoes a number of potential optimizations.

10. Delay work

Much work is delayable; decouple with queues.



11. Shunt state



Only the bottom-most layer here is stateful.

Beware caching; it is an excellent way to introduce hidden state to your system.

12. Hide caches

Caching is a necessary evil in most systems. But they are not *magical scaling sprinkles* which allow you to easily increase capacity.

They introduce a lot of complexity:

- What's your invalidation policy?
- Do you know where your writes are?
- What cache-hit rate does your system require to stay up?

If we instead reframe caching as *in-memory serving frontends* then we are forced to confront these issues, e.g.,

- Should this be integrated with our data-base so that we can extend consistency guarantees?
- Do we need to replicate the cache also?

At the very least, be very clear about the operational implications of caching.

13. Push complexity down

Maintain a bottom heavy complexity distribution. There is more leverage this way, and it usually makes for simpler systems on top.

Examples:

- Storage systems with strong(er) guarantees (e.g., HBase vs. Cassandra.);
- ordered, durable queues;
- JVM: JIT, GC.

These help systems *compose*.

14. Think end-to-end

It's often the case that an *end-to-end* solution yields a simpler system. Examples include:

- Top-level *retries*;
- unreliable control plane;
- quorum reads/reconciliation.

15. Expect failure, and fail well

resilience, n. The act of resiling, springing back, or rebounding; as, the resilience of a ball or of sound.

The systems environment is a hazardous one; among the things we must tolerate are:

- Process failure;
- overload;
- operator error;
- poor network conditions;
- network partitions.

Systems must be *designed* for this. They must *fail well*.

Balance of MTTF vs. MTTR.

Develop a common toolkit, vocabulary, *patterns*.

16. Demand less

Exploit application semantics to demand less of your systems.

This makes everything simpler, and also informs how you might degrade gracefully.

e.g., Twitter/Facebook timelines.

17. Beware hot shards and thundering herds

With scale, it's very easy to accidentally overload your own system, often in surprising ways!

Must adopt systems thinking; ''butterfly effects'' abound.

Admission control, load shedding, and hot-key caching can help.

18. Embrace Conway's law

> organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations
> —M. Conway

This seems to be a universal law. Reverse it: It's wise to structure organizations the way you'd like to structure your software architecture.

19. Deploy changes gradually

The ways in which your systems interact will continue to surprise you.

Production changes have a way of teasing these out, and to surprise you.

Roll things out slowly, and keep things *isolated*.

- process canary;
- datacenter canary;
- global.

Use *feature flags* to do the same at a finer granularity. These are also very handy to keep around for emergencies.

20. Keep a tab

In large systems, even simple parts interact in complex, often unforeseen ways.

It's important to keep a tab of what's going on globally.

- Change management;
- chat rooms;
- consolidated dashboards (spot the correlations!).

21. Plan to deprecate

Most systems have a useful lifetime of a few years. (Maybe 3-5 if you're lucky.)

How can you deprecate systems? Strict API boundaries help, but the organization must also be set up to deal with it.

A few questions to consider

These are a set of common questions which arise when designing systems in a distributed environment. This is not a complete list by any means, but they may be a useful set of *prompts* for issues to consider.

Fault tolerance

- What happens when a dependency starts failing? What if it begins failing *slowly*?
- How can the system *degrade* in a graceful manner?
- How does the system react to *overload*? Is it ‘‘well conditioned?’’
- What’s the worst-case scenario for total failure?
- How quickly can the system recover?
- Is delayable work delayed?
- Is the system as *simple* as possible?
- How can the system *shed load*?
- Which failures can be mitigated, and how?
- Which operations may be *retried*? Are they?

Scalability

- How does the system *grow*? What is the chief metric with which the system scales?
- How does the system scale to multiple datacenters?
- How does demand *vary*? How do you ensure the system is always able to handle peak loads?
- How much query processing is done? Is it possible to shape data into queries?
- Is the system replicated?

Operability

- How can features be turned on or off?
- How do you monitor the system? How do you detect anomalies?
- Does the system have operational needs specific to the application?
- How do you deploy the system? How do you deploy in an emergency?
- What are the capacity needs? How does the system grow?
- How do you configure the system? How do you configure the system *quickly*?
- Does the system behave in a *predictable* manner? Where are there nonlinearities in load or failure responses?

Efficiency

- Is it possible to precompute data?
- Are you doing *as little work as possible*?
- Is the program as concurrent as possible? (“Concurrent programs wait faster.”)
- Does the system make use of work batching?
- Have you profiled the system? Is it possible to profile in situ?
- Are there opportunities for parallelization?
- Can you load test the system? How do you catch performance regressions?

Thanks.

Eriksen, Marius. *Your server as a function.* In Proceedings of the Seventh Workshop on Programming Languages and Operating Systems, p. 5. ACM, 2013.

Hanmer, Robert. *Patterns for fault tolerant software.* John Wiley & Sons, 2013.

twitter.github.io/finagle

monkey.org/~marius/redux.html