O'REILLY®
Velocity

How to scale a distributed system

Henry Robinson

@henryr / henry.robinson@gmail.com

velocityconf.com
#VelocityConf

What is this, and who's it for?

- Lessons learned from the trenches building distributed systems for 8+ years at Cloudera and in open source communities.

What is this, and who's it for?

- Lessons learned from the trenches building distributed systems for 8+ years at Cloudera and in open source communities.

- Not:
  - A complete course in distributed systems theory (but boy do I have references for you)
  - Always specific to distributed systems
  - Complete
  - Signed off by experts
  - A panacea (sorry)

…and you are?

- Distributed systems dilettante
- Some years in graduate school for distributed systems

  ..followed by some years in industry for the same thing.
- Some writing on my blog: http://the-paper-trail.org/
- A community: https://dist-sys-slack.herokuapp.com/ for the invite

Today

- Primitives
- Practices
- Possibility
- Papers

Today

- **Primitives** - what are the concepts, and nouns, that it's important to know?
- **Practices** - what are good habits in distributed systems design?
- **Possibility** - how should we think - if at all - about formal impossibility?
- **Papers** - you don't have time to read everything? Join the club.

[spoiler: everyone argues
about CAP, forever]

I. Primitives

Basic concepts

- Processes may fail.
- There is no particularly good way to tell that they have done so.
- Almost always better to err on the side of caution.

Basic concepts

1. Failure detectors
2. Symmetry breaking (with leader election as an example)
3. Fault models
4. Replicated state machines
5. Quorums
6. Logical time
7. Coordination: broadcast, consensus, commit protocols

2. Practices

Hints for Computer System Design

Butler W. Lampson

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Abstract

Experience with the design and implementation of a number of computer systems, and study of many other systems, has led to some general hints for system design which are described here. They are illustrated by a number of examples, ranging from hardware such as the Alto and the Dorado to applications programs such as Bravo and Star.

1. Introduction

Designing a computer system is very different from designing an algorithm:

> The *external* interface (i.e., the requirement) is more complex, less precisely defined, and more subject to change.
>
> The system has much more internal structure, and hence many *internal* interfaces.
>
> The measure of success is much less clear.

and studied many other systems, both successful and unsuccessful. From this experience come some general hints for designing successful systems. I claim no originality for them; most are part of the folk wisdom of experienced designers. Nonetheless, even the expert often forgets, and after the second system [6] comes the fourth one.

Disclaimer: These are not

> novel (with a few exceptions),
>
> foolproof recipes,
>
> laws of system design or operation,
>
> precisely formulated,
>
> consistent,
>
> always appropriate,
>
> approved by all the leading experts,
>
> or guaranteed to work;

they are just hints. Some are quite general and vague; others are specific techniques which are more widely applicable than many people know. Both the hints and the illustrative examples are necessarily oversimplified. Many

Hints for Computer System Design

Butler W. Lampson

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Abstract

Experience with the design and implementation of a number of computer systems, and study of many other systems, has led to some general hints for system design which are described here. They are illustrated by a number of examples, ranging from hardware such as the Alto and the Dorado to applications programs such as Bravo and Star.

1. Introduction

Designing a computer system is very different from designing an algorithm:

> The *external* interface (i.e., the requirement) is more complex, less precisely defined, and more subject to change.
>
> The system has much more internal structure, and hence many *internal* interfaces.
>
> The measure of success is much less clear.

and studied many other systems, both successful and unsuccessful. From this experience come some general hints for designing successful systems. I claim no originality for them; most are part of the folk wisdom of experienced designers. Nonetheless, even the expert often forgets, and after the second system [6] comes the fourth one.

Disclaimer: These are not

> novel (with a few exceptions),
>
> foolproof recipes,
>
> laws of system design or operation,
>
> precisely formulated,
>
> consistent,
>
> always appropriate,
>
> approved by all the leading experts,
>
> or guaranteed to work;

they are just hints. Some are quite general and vague; others are specific techniques which are more widely applicable than many people know. Both the hints and the illustrative examples are necessarily oversimplified. Many

Always Be sCaling

What do we talk about, when we talk about scaling?

- Scaling (up) means more. Of everything.

- “what happens to the **behavioral characteristics** of my system as the **operational parameters** increase?”

- Not just number of nodes.

Why are we scaling? Not just increased load.

- Commodity hardware revolution made incremental capacity improvements possible.
- The operational mode of the software we build has changed: **availability** is the sword by which web properties live or die.
- **Redundancy** is the basic conceptual approach to providing availability
- Adding more processing power is how we provide redundancy; i.e. we **scale our systems up**.

Scalability axes

- One rarely considered scalability axis: more failures. (and more types of failure)

Scalability axes

- One rarely considered scalability axis: more failures. (and more types of failure)

- GFS Paper (SOSP 2003)

> First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Scalability axes

- One rarely considered scalability axis: more failures. (and more types of failure)

- GFS Paper (SOSP 2003)

> First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Apache Impala has to scale with respect to…

Apache Impala has to scale with respect to…

- Query complexity
- Queries per second
- Cluster size
- Node CPU / memory
- Degree of per-node parallelism
- Number of clients per node
- Number of clients per cluster
- Number of tables
- Number of partitions per table

Apache Impala has to scale with respect to…

- Query complexity
- Queries per second
- Cluster size
- Node CPU / memory
- Degree of per-node parallelism
- Number of clients per node
- Number of clients per cluster
- Number of tables
- Number of partitions per table

- Number of columns per table
- Data size per table
- Intermediate result size
- Kerberos ticket grants

Scale is a fundamental design consideration

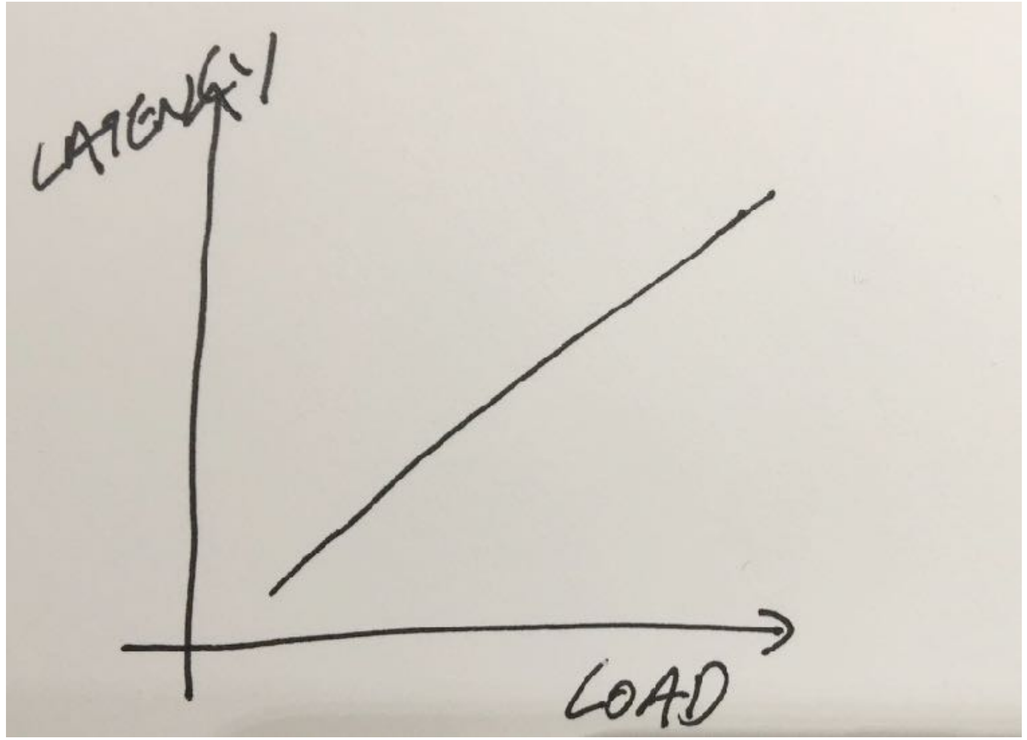
Just like security, include scalability in your thinking from day one.

Scalability behaviors are usually **discontinuous** - they exhibit phase changes rather than gradual improvement. (20->50 nodes, not 20->22)

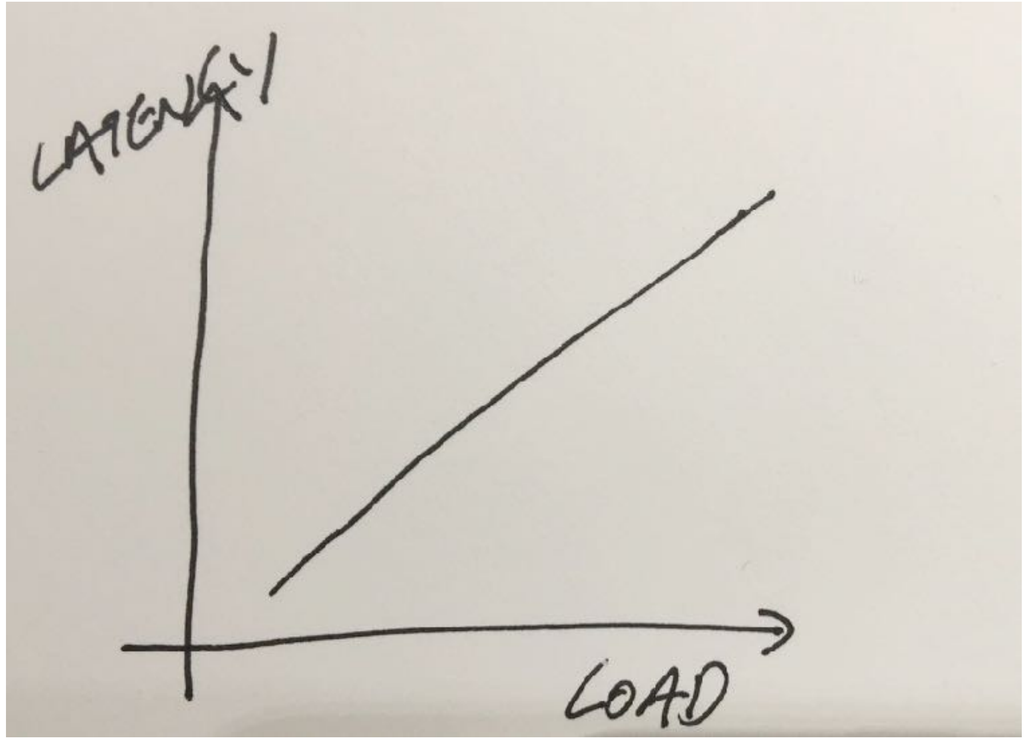
That means you can clearly identify scaling boundaries. **Do this wherever possible.** The rest of the your team - *and the systems you interact with* - will thank you for it.

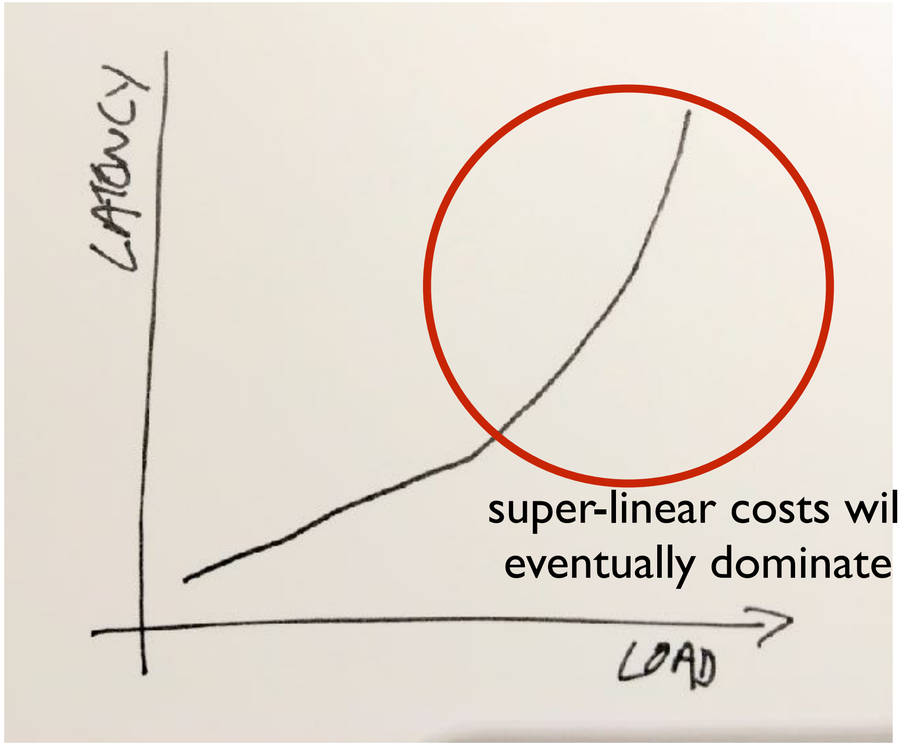
It also means that, by attacking the scaling boundary, you can have a large impact - when the time is right.

Draw your borders before you drive off a cliff



Draw your borders before you drive off a cliff





Decompose system properties into **safety** and **liveness**

System invariants

Safety

Liveness

System invariants

Safety

Liveness

“Nothing bad ever happens!”

For example:

- Queries never return incorrect results
- Corrupt data is never written to disk
- Data is never read remotely
- Only one leader exists at any time

System invariants

Safety

“Nothing bad ever happens!”

For example:

- Queries never return incorrect results
- Corrupt data is never written to disk
- Data is never read remotely
- Only one leader exists at any time

Liveness

“Something good eventually happens!”

For example:

- New nodes eventually join the cluster
- All queries complete
- Some data gets written to disk on INSERT

System invariants

Safety

“Nothing bad ever happens!”

For example:

- Queries never return incorrect results
- Corrupt
- Data is
- Only one leader exists at any time

Liveness

“Something good eventually happens!”

For example:

- New nodes eventually join the cluster

All system properties can be described as a combination of safety and liveness properties.

Example: Impala's query liveness and safety

- For queries, liveness means "all queries eventually complete"
  *(note I didn't say they complete **successfully**)*

- **Safety** property is more interesting. Choice between:
  1. Query never returns anything but its full result set
  2. Query must return anything, but must signal an error when it does.

- Impala chose option #2, despite #1 being much more attractive.

- ***Why?***

Example: Impala's query liveness and safety

- It's obviously better to always return complete results, but failures make that extremely hard.
- If Impala had tried to enforce strong query safety from day 1, it would never have been a success: achieving performance goals would have been much harder.
- Instead, make fault tolerance trivial by weakening the definition. By definition, such a system scales better.

Think global,
act local.

Coordination costs

- Coordination: getting different processes to agree on some shared fact.
- Coordination is incredibly costly in distributed systems and the cost increases with the number of participants.
- This is the reason most ZooKeeper deployments are 3-5 nodes.

Avoid coordination wherever possible

- Mostly got this right in Impala:
  - Metadata consistent on session level (sticky to one machine) -> no coordination required
  - Data processing is heavily parallel.
  - Coordination happens almost entirely at distinguished **coordinator node**, asynchronously wrt to query execution

Example: synchronous DDL

- Some users wanted cross-session metadata consistency, i.e. I create a table, you can instantly see it.

- Problem: symmetry of Impala's architecture means every Impala daemon needs to see all updates synchronously.

- Latency of these operations is by definition pessimal.

Small control plane,
big data plane

Two types of communication

- Communication in distributed systems serves roughly one of two purposes:
- **Control** logic tells processes what to do next
- **Data** flow exchanges data between processes for computation

Data vs control

Data protocols

- Simple protocols
- Typically need local-state only
- Very high data volume
- Heavy resource consumption
- Highly scalable
- Dominates CPU execution time

Data vs control

Data protocols

- Simple protocols
- Typically need local-state only
- Very high data volume
- Heavy resource consumption
- Highly scalable
- Dominates CPU execution time

Control protocols

- Complex protocols
- Global view of cluster state
- Relatively small data volume
- Lightweight resource consumption
- Not highly scalable
- Low relative cost

3. (im)possibility

YOU CAN'T DO THAT!

- Nothing trips up Distributed Systems Twitter faster than impossibility results

- Two camps:
  - "your system doesn't beat CAP, so I don't care"
  - "I don't care about CAP, it's really unlikely I'll lose that transaction"

- Impossibility results - and there are a lot of them - tell us about some fundamental tension. But they are completely silent on *practicalities*. Just because you can't do something, doesn't mean you shouldn't try.

- The best way to think about impossibility is to recognize the **safety** and **liveness** tension that a result represents.

  - Decide which you're willing to give up.

  - And then protect the other at all cost.

4. Papers

Read papers.

Read papers.
Not too many.

Read papers.
Not too many.
Mostly real systems papers.

Thank you!