We need distributed systems

• For a lot of applications, either the data or request volume (or both) are too big for one system to handle
  – load balancing: distribute computation

• We also want them for
  – High availability
  – Geographic proximity (reduced latency)
  – Access to services from anywhere (cloud-based)
  – Segmenting services (e.g., files, authentication)

Good services

• Well-designed services (whether network or libraries) are
  – Well-defined & documented
  – Have minimal dependencies
    Can be developed and tested separately
  – Easy to test
  – Language independent & Platform independent
    Will you be able to access your Java service from a Go or Python program?
    Does the service only work with an iOS app?

KISS: Keep It Simple, Stupid!

• Make services easy to use
  • Will others be able to make sense of it?
  • Will you understand your own service a year from now?
  • Is it easy to test and validate the service?
  • Will you (or someone else) be able to fix problems?

  Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?

  – Brian Kernighan

Good protocol design is crucial

• Interfaces should make sense

• Sockets & RPC are still the core of interacting with services

• Efficiency & interoperability
  – REST/JSON popular for web-based services
  – XML is still out there ... but not that efficient
  – Great for public-facing services
  – Google uses Protocol Buffers in lots of places
  – Self-describing – defines the service interface
  – Supports multiple languages
  – Really efficient and compact
  – Lots of RPC and RPC-like systems out there – many built on top of Java
  – XML or JSON (more JSON than XML)

Design for Parallelism

• Need to figure out how to partition problems for maximum parallelism
  – Sharding data
  – Concurrent processes with minimal or no IPC
  – Do a lot of work in parallel and then merge results

• Also, partition data for scalability
  – Distribute user data across multiple machines
    (e.g., Dynamo or Bigtable)

  E.g., at Google, one web search touches 50+ separate services and 1000s of machines
Design for Low Latency

- Users hate to wait
  - Amazon: every 100ms latency costs 1% sales
  - Google: extra 500ms latency reduces traffic by 20%
  - Sometimes, milliseconds really matter, like high frequency trading
    - E.g., Spread Networks built NYC-Chicago fiber to reduce RTT from 16ms to 13ms
- Avoid moving unnecessary data
- Reduce the number of operations through clean design
  - Particularly number of API calls
- Reduce amount of data per remote request
  - Efficient RPC encoding & compression (if it makes sense)
- Avoid extra hops
  - E.g., Dynamo vs. CAN or finger tables
- Do things in parallel
- Load balancing, replication, geographic proximity

Asynchronous Operations

- Some things are best done asynchronously
  - Provide an immediate response to the user while still committing transactions or updating files
  - Replicate data eventually
    - Opportunity to balance load by delaying operations
    - Reduce latency
      - The delay to copy data does not count in the transaction time!
    - But watch out for consistency problems

Know the cost of everything

- Don’t be afraid to profile!
  - CPU overhead
  - Memory usage of each service
  - RPC round trip time
  - UDP vs. TCP
  - Time to get a lock
  - Time to read or write data
  - Time to update all replicas
  - Time to transfer a block of data to another service
    - in another datacenter?
- Systems & software change frequently
  - Don’t trust the web … find out for yourself

Design for Scale

Prepare to go from this…

… and this
Google Data Center: Douglas County, Georgia

Google Data Center: Council Bluffs, Iowa

Facebook’s Data Center: Prineville, Oregon

Scalability
• Design for scale
  – Be prepared to re-design

• Something that starts as a collection of three machines might grow
  – Will the algorithms scale?

• Don’t be afraid to test alternate designs

Availability
• Things will break
  – Hardware and software will fail

• Even amazingly reliable systems will fail
  – Put together 10,000 systems, each with 30 years MTBF
  – One will fail each day on average!

• Google’s experience
  – 1-5% of disk drives die per year (300 out of 10,000 drives)
  – 2-4% of servers fail – servers crash at least twice per year

It’s Unlikely Everything Will Fail
• Software has to be prepared to deal with partial failure

• Watch out for default behavior on things like RPC retries
  – Is this what you want … or try alternate servers?
  – Failure breaks function-call transparency. RPC isn’t always as pretty as it looks in demo code
  – Handling errors often makes code big and ugly
  – What happens if a message does not arrive?

• Replicated data & distributed state machines can help
  – Incoming messages take a module to a different state
  – Know the states in your system and valid transitions
  – Be sure software does not get into an unknown state
Replication

- Replication helps handle failure (it’s a form of backup)
  - and increase performance by reducing latency
  - It reduces contention & load on each system and gives geographic diversity

- BUT – we need to understand consistency
  - Strict consistency impacts latency, partition tolerance, & availability
  - Eventual consistency
    - ... lets us replicate in the background or delay until a system is reachable
  - But we need to be aware of the repercussions

- Total ordering and state synchronization can be really useful
  - But needs to be done reliably

Fault Detection

- Detection
  - Heartbeat networks: watch out for partitions!
  - Software process monitoring
  - Software heartbeats & watchdog timers
  - How long is it before you detect something is wrong and do something about it?

- What if a service is not responding?
  - Sure, you can have it restarted
  - But a user may not have patience. Fail gracefully.
  - Use logging – it may be your only hope in figuring out what went wrong with your systems or your software

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause trouble and poor learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn’t change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arun Rotem-Gal-Oz.