We need distributed systems

We often have a lot of data to ingest, process, and/or store

- The data or request volume (or both) are too big for one system to handle
- Scale – distribute input, computation, and storage

We also want to distribute systems for
- High availability
- Parallel computation (e.g., supercomputing)
- Remote operations (e.g., cars, mobile phones, surveillance cameras, ATM systems)
- Geographic proximity (reduced latency)
- Content & Commerce: news, social, etc.
- Cloud-based storage & sharing
- Cloud services – SOA (e.g., file storage, authentication)

Designing distributed systems is not easy
Programmers tried to make it simpler
That also made it complicated

Good design
Design software as a collection of services

Well-designed services are

- Well-defined & documented
- Have minimal dependencies
- Easy to test
- Language independent & platform independent
  
- Can be developed and tested separately

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

**KISS: Keep It Simple, Stupid!**

- Don't over-engineer or over-optimize
- Understand where potential problems may be
- Redesign what's needed

**Good protocol design is crucial**

- Interfaces should make sense
- Sockets are still the core of interacting with services
- RPC (& remote objects) great for local, non-web services
  - but think about what happens when things fail
    - Will the service keep re-trying?
    - How long before it gives up?
    - Was any state lost on the server?
    - Can any failover happen automatically?

**Interfaces**

- Efficiency & interoperability
  - ... and avoid writing your own parser
- REST/JSON popular for web-based services
  - XML is still out there ... but not efficient and used less and less
  - REST/JSON great for public-facing & web services
- But you don’t need to use web services for all interfaces
  - There are benefits ... but also costs
- Use automatic code generation from interfaces
  - It’s easier and reduces bugs

**Efficient & portable marshaling**

- Google Protocol Buffers gaining in lots of places
  - Self describing schemas – defines the service interface
  - Versioning built in
  - Supports multiple languages
  - Really efficient and compact
- Investigate successors ... like Cap’n Proto (capnproto.org)
  - Pick something with staying power – You don’t want to rewrite a lot of code when your interface generator is no longer supported
- Lots of RPC and RPC-like systems out there — many use JSON for marshaling
  - Supported by C, C++, Go, Python, PHP, etc.

**Fundamental Issues**

- Partial failure
- Concurrency
- Consistency
- Latency
- Security

**Design for Scale**
Prepare to go from this…

1996: Basement lab of Gates Information Sciences, Stanford

… and this…

1996: Basement lab of Gates Information Sciences, Stanford

… to this

Google Data Center: Douglas County, Georgia

… and this

Google Data Center: Council Bluffs, Iowa

… or this

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
Design for scale & parallelism

- Figure out how to partition problems for maximum parallelism
  - Shard data
  - Concurrent processes with minimal or no IPC
  - Do a lot of work in parallel and then merge results
- Design with scaling in mind – even if you don’t have a need for it now
  - E.g., MapReduce works on 2 systems or 2,000
- Consider your need to process endless streaming data vs. stored data
  - Partition data for scalability
    - Distribute data across multiple machines (e.g., Dynamo, Bigtable, HDFS)
  - Use multithreading
    - It lets the OS take advantage of multi-core CPUs

Design for High Availability

Availability

- Everything breaks: hardware and software will fail
  - Disks, even SSDs
  - Routers
  - Memory
  - Switches
  - ISP connections
  - Power supplies; data center power, UPS systems
- Even amazingly reliable systems will fail
  - Put together 10,000 systems, each with 30 years MTBF
  - Expect an average of a failure per day!

It’s unlikely everything will fail at once

- Software has to be prepared to deal with partial failure
- Watch out for default behavior on things like RPC retries
  - Is retrying what you really want … or should you by 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?
    - Easier with designs that support asynchronous sending and responses and handle timeouts
- Replicated data & distributed state machines can help
  - Decide on stateful vs. stateless services
  - 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 may increase performance by reducing latency
  - It reduces contention & load on each system and gives geographic diversity
- BUT – it has a cost – 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
  - Need consensus – Raft or Paxos
Fault Detection

- Detection
  - Heartbeat networks: watch out for high latency & 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?
  - You can have it restarted
  - But a user may not have patience
    - Maybe fail gracefully
  - Or, better yet, have an active backup

- Logging
  - It may be your only hope in figuring out what went wrong with your systems or your software

Think about the worst case

- Deploy across multiple Availability Zones (AZs)
  - Handle data center failure
- Don’t be dependent on any one system for the service to function
- Prepare for disaster recovery
  - Periodic snapshots
  - Long-term storage of data (e.g., Amazon Glacier)
  - Recovery of all software needed to run services (e.g., via Amazon S3)

Design for High Performance & Low Latency

- 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
- CPU performance scaled faster than networks or disk latency
- You cannot defeat physics
  - It’s 9,567 miles (15,396 km) from New Jersey to Singapore
  - 75 ms via direct fiber … but you don’t have a direct fiber!

Think about the worst case

- Deploy across multiple Availability Zones (AZs)
  - Handle data center failure
- Don’t be dependent on any one system for the service to function
- Prepare for disaster recovery
  - Periodic snapshots
  - Long-term storage of data (e.g., Amazon Glacier)
  - Recovery of all software needed to run services (e.g., via Amazon S3)

Design for Low Latency

- 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
- CPU performance scaled faster than networks or disk latency
- You cannot defeat physics
  - It’s 9,567 miles (15,396 km) from New Jersey to Singapore
  - 75 ms via direct fiber … but you don’t have a direct fiber!

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 (can you live with them?)
  - But if you need consistency, use frameworks that provide it
    - Avoid having users reinvent consistency solutions

Design for Low Latency

- 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
- CPU performance scaled faster than networks or disk latency
- You cannot defeat physics
  - It’s 9,567 miles (15,396 km) from New Jersey to Singapore
  - 75 ms via direct fiber … but you don’t have a direct fiber!
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

Testing, profiling, and optimization

- Continuously benchmark and test
  - Avoid future surprises
- Optimize critical paths
  - Watch out for overhead of interpreted environments
  - Consider languages that compile, such as go

Understand what you’re working with

- Understand underlying implementations
  - The tools you’re using & their repercussions
  - Scalability
  - Data sizes
  - Latency
  - Performance under various failure modes
  - Consistency guarantees
- Design services to hide the complexity of distribution from higher-level services
  - E.g., MapReduce, Pregel, Dynamo

Don’t do everything yourself

- There’s a lot of stuff out there
  - Use it if it works & you understand it
- Security is really difficult to get right
  - Authentication, encryption, key management, protocols
  - Consider using API gateways for service authorization
  - Secure, authenticated communication channels
  - Authorization service via OAuth OpenID Connect
  - Pay attention to how & where keys are stored and managed
  - Security should not be an afterthought!

Test & deployment

- Test partial failure modes
  - What happens when some services fail?
  - What if the network is slow vs. partitioned?
- Unit tests & system tests
  - Unit testing
  - Integration & smoke testing (build verification): see that the system seems to work
  - Input validation
  - Scale: add/remove systems for scale
  - Failure
  - Latency
  - Load
  - Memory use over time
**Infrastructure as code**

- Version-managed & archived configurations
- Never a need for manual configuration
- Create arbitrary number of environments
- Deploy development, test, & production environments
- E.g., TerraForm

**Blue/Green deployment**

- Run two identical production environments
- Two versions of each module of code: blue & green
  - One is live and the other idle
- Production points to code versions of a specific color
- Staging environment points to the latest version of each module
  - Deploy new code to non-production color
  - Test & validate
  - Switch to new deployment color
- Simplifies rollback

---

The Eight Fallacies of Distributed Computing

Peter Deuker

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 big troubles and painful 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. There is one owner
8. The network is homogeneous

For more details, read the article by Aaron Batten-Gal-Oz

---

The end