ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 16 Scalability

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

October 8, 2025

Outline

Reading Assignment

- This lecture: 7
- Next two lectures (10/15, 10/20)
  - Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center `https://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdf`
  - Large-scale cluster management at Google with Borg `https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43438.pdf`

Outline

Considerations for Scaling

Efficiency without Scaling

Why Scalability?

- Able to add resources on demand avoids exteneded downtime
  - Serve unexpected amount of users.
  - Make resources ready to be available when some fails.
- Using more resources than necessary is expensive.
  - Only pay for what you need – a major advantage for startups.
- Scalable services can live longer than its original expectations.
  - Unscalable ones aren't capable of growing much.

Efficiency and Scalability

- Scalability is not all about adding physical resources to handle large swings in demand.
- Systems built with efficiency in mind are more likely to be scalable.
  - Able to absorb higher demand without the need to adding hardware.
- Go helps to build services that are more efficient.
  - In addition to scalable architecture and messaging patterns.

Methods of Scaling

- Vertical scaling (scale up)
  - Increase resource allocation for a single system.
  - E.g. to rent a better server instance – though there is a limit.
- Horizontal scaling (scale out)
  - Duplicate to limit the burden on any individual server.
  - The presence of state may make it difficult or impossible.
- Functional partitioning
  - Decompose large system into smaller functional units.
  - Each unit is independently optimized, managed, and scaled.

Performance Bottlenecks

- CPU: processing power
  - Better CPU, GPU/FPGA accelerators, caching (more memory), distributed and parallel processing (more network I/O).
- Memory: capacity, throughput, and latency
  - Better memory, more memory channels, compression (more CPU), paging (more disk I/O), distributed caching (more network I/O)
- Disk I/O: throughput and latency
  - Better drive, caching (more memory), compression (more CPU), distributed storage (more network I/O)
- Network I/O: throughput and latency
  - Shorter distance, better hardware, compression (more CPU).
- Scaling up is difficult since we are approaching limits of physics: device sizes, power density and heat transfer, and speed of light.

Application State vs. Resource State

- Application state: variables, objects, execution flow.
  - For an application to resume itself if terminated unexpectedly.
  - Not limited to the application itself: what about network connections and other resources managed by the OS?
  - Can we have a solution to maintain application state that is transparent to the application?
- Resource state: data stored explicitly in some data store.
  - Allow shared accesses via database operations or with stronger guarantees like ACID transactions.
  - Durability can be provided to survive failures.

Stateful vs Stateless Applications

- Stateful applications are those that cannot be safely restarted if their application states are not known.
  - E.g. after unexpected terminations.
- Stateless applications are those that can utilize resource state to restart from a known good configuration.
  - Benefit scalability since multiple requests can be processed independently by creating new application instances.
  - Make fail-fast possible which is simpler to implement than to recover from faults.
  - Encourage idempotent operations whose results are easier to cache.

Outline

Considerations for Scaling

Efficiency without Scaling

Caching

- Trade-off memory to save CPU and disk/network throughput, and to reduce access latency.
- Could be implemented as a key-value map, but need to support
  - Concurrent accesses.
  - Less contention at increaing amount of clients.
  - Bounded memory usage.
- A popular choice: LRU (least recently used) cache
  - Key-value pairs in the map are additionally connected via a doubly linked list.
  - When a pair is accessed, it is moved to the back of the list.
  - Pairs at the front of the list are the least recently used, and can be evicted if the memory usage reaches the limit.
  - Sharding may improve performance to access map concurrently, and there are efficient algorithms to manipulate linked list concurrently.

Synchronization

- Communication via shared memory depends a lot on synchronization primitives like locks.
  - Threads competing for the lock at the same time will cause contentions, which may degrade performance.
  - Contentions are more likely to happen if a thread hold a lock for a long time, e.g. to complete a long computation – this is when you need the performance the most.
  - Consider to use fine grained locks via sharding to reduce contention but be careful about deadlocks.
- Go prefers to use communication via message passing.
  - Still, since multiple goroutines may access the same channel concurrently for read and write, synchronization is unavoidable.
  - However, because each goroutine only need to access the channel briefly for only read or write but not long computation, there will be very few contentions.
  - Buffered channels further reduce blocking and enable all goroutines to run at their full speeds.

Watch out for Memory Leaks

- GC (garbage collection) gives the illusion that memory from all unused objects can be reclaimed for future use.
  - Apparently "no memory leak!"
- However, there are other resources not managed by GC.
  - And they will consume memory if not released properly, causing memory leaks.
  - E.g. network connections, file descriptors, threads not terminated but holding a lot of memory blocks.
- For Go, pay special attention to goroutines that do not have a clear exiting condition.
  - They may refer to channels that consume a lot of memory.
  - GC cannot reclaim these channels and the associated memory as the goroutines are still using them.

Summary

- Stateless applications are easier to scale horizontally.
- Scalability is not all about adding physical resources.
- Optimizing applications to overcome the performance bottlenecks helps to scale them better.