ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 02 Cloud Native Systems

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

August 21, 2024

Outline

Cloud Native Systems

Reading Assignment

- This lecture: 1,6
- Next lecture: 2,3

Outline

Networked Applications

- In early days of computing when there is only mainframes, all programs ran and all data was stored in a single location.
- With inexpensive network-connected PCs that can perform non-trivial tasks, multitiered architecture was introduced.
  - Data management tier: database server
  - Business logic tier: web and application server
  - Presentation tier: PC (and mobile devices)
  - Connected via networks and can be replaced independently.
- Complexity of software increases beyond what can be managed by a single developer team efficiently.
  - Tiers, especially business logics, are decomposed into microservices.
  - These systems are now <u>distributed systems</u>, which behave very differently than a single or a few interconnected computers.
- SaaS and IaaS business practices emerge.

Scaling

- With SaaS practices, it is possible to have exponential growth in customers.
  - No need to distribute software physically, e.g. via disks.
  - Can the services scale to meet the growth as soon as possible?
  - Can the services respond to the demand dynamically to maximize the profit?
- With IaaS offerings, especially S3 and EC2 from AWS, scaling seems economically viable.
  - No need to build facilities and buy physical servers.
- Scaling is not easy.
  - Managing those resources manually at such scale is impossible.
  - Components in a distributed system, e.g. servers and nerwork connections, will fail.
  - Most distributed systems will have nondeterministic behaviors – it is very difficult to reproduce the same outputs even if the inputs are the same.

What Is Cloud Native?

- A paradigm to design systems that can fully utilize the cloud computing architecture.
  - Promoted by the Cloud Native Computing Foundation.
- Cloud native attributes
  - Scalable
  - Loosely coupled
  - Resilient
  - Manageable
  - Observable

Scalability

- The ability of system to continue to behave as expected in the face of significant upward or downward changes in demand.
  - Not necessarily a must have for initial system design.
  - But hard to remedy for at a later time.
- Vertical scaling
  - Improve performance of an instance (server or virtual machine) by adding cores, memory, storage, etc.
  - Usually requires no software change but improvements are limited due to physical limites.
- Horizontal scaling
  - Improve performance of the system by using more instances.
  - Increased complexity in system design and management.
- If the service need to be scaled by hundreds or thousands times, horizontal scaling is the only choice.
  - Unfortunately, not all computations can be horizontally scaled.
  - In particular computations as finite state machines whose state cannot be decomposed, e.g. counting.

Loose Coupling

- Components in a system have minimal knowledge of any other components.
  - Thus a component can be changed without requiring to change other components.
  - E.g. web servers and web browsers.
- Components would need to communicate to each other using certain standard protocols.
- Note that the coupling needs to be loose and the knowledge need to be minimal.
  - If every component need to communicate with and thus has knowledge of many other components, the system will become a distributed monolith – a nightmare of microservices.

Resilience

- A measure of how well a system withstands and recovers from errors and faults.
  - A resilient system continues operating correctly, possibly at a reduced level, rather than failing completely.
- Failures result from faults
  - Any system can contain defects or faults, e.g. bugs.
  - Faults can lead to errors, which can cause failures.
  - Failures can propagate from components to components, and then the whole system.
- Build resilient systems
  - Prevention of all faults is unrealistic and unproductive.
  - Should assume components will fail and limit their impacts.
  - Use mechanisms like redundancy, circuit breakers, retry logic, intentional failure.

Manageability

- The ease of modifying system behavior to ensure security, and smooth operation, and to meet changing requirements.
  - Without altering its code.
- For example, consider a system with a service and a database.
  - How to update the reference to the database in the service?
  - What if multiple versions of services and databases need to coexist for troubleshooting and performance evaluation.
  - Make use of environment variables instead of hardcoding.
- Manageable systems adapt to changing requirements
  - Feature flags
  - Credential rotation
  - Component deployment (upgrades and downgrades)
  - Instead of ad hoc code changes

Observability

- The ability to answer unanticipated questions quickly with minimal prior knowledge and without reinstrumentation.
  - Make it possible to troubleshoot issues without reproducing nondeterministic behaviors.
  - In particular where an issue is.
- Observability in modern distributed systems
  - Build upon metrics, logging, and tracing.
  - Collected data to deduct internal states from external outputs.

Why Cloud Native?

- Dependability: produce dependable services in unreliable environments to keep users happy.
  - Availability: the ability of a system to perform its intended function at a random moment in time, e.g. uptime
  - Reliability: the ability of a system to perform its intended function for a given time interval, e.g. MTBF
  - Maintainability: the ability of a system to undergo modifications and repairs.
- Developers can no longer just prioritize feature development and leave dependability to system administrators.
  - Fault prevention
  - Fault tolerance
  - Fault removal
  - Fault forecasting

Fault Prevention

- Good programming practices, e.g. test-driven development
- Language features, e.g. garbage collection
- Scalability ensures correct behavior as demand changes significantly, but could be source of additional faults.
- Loose coupling reduces the risk of cascading failures as faults propagate from one component to another.

Fault Tolerance

- Resilience in two steps: error detection and recovery
- Seemingly simple case: retry failed requests.
  - Will retrying cause more faults?
- Make use of redundancy: have multiple copies of critical components.
  - Will a majority vote serve both as mechanisms for error detection and recovery?

Fault Removal

- Source of faults
  - Implemenation: human errors in system design and development, i.e. bugs
  - Environment: unexpected inputs or operation conditions.
- Verification and testing
  - Remove faults at development time.
- Manageability
  - Adapt to the environment: adjust resource usage, remedy security issues, turn features on or off, etc.

Fault Forecasting

- Observability helps to avoid guesswork on predicting future behavior of the system

The Twelve-Factor App

- Early wisdoms on building web apps remain valid.

1. One codebase tracked in revision control, many deploys.
2. Explicitly declare and isolate (code) dependencies.
3. Store configuration in the environment.
4. Treat backing services as attached resources.
5. Strictly separate build and run stages.
6. Execute the app as one or more stateless processes.

The Twelve-Factor App (cont.)

7. Each service manages its own data.
8. Scale out via the process model.
9. Maximize robustness with fast startup and graceful shutdown.
10. Keep development, staging, and production as similar as possible.
11. Treat logs as event streams.
12. Run administrative/management tasks as one-off processes.

Summary

- Cloud native systems provide dependability on top of unreliable cloud computing environments.