Elasticity

COSC349—Cloud Computing Architecture

David Eyers

Learning objectives

- Illustrate what a **scale-out system** is, using an example
- Define **elasticity** in the context of cloud computing
- Explain why cloud computing is suited to offer elasticity
- List some ways in which **services can be partitioned**
- Describe how **caching** can help effect scalability

Elasticity: servers matching client load

- Online systems may have **highly variable load**
  - Large differences between peak / average demand
  - Inefficient to provision for **peak demand**
  - Unsafe to provision only for **average demand**

- **Scalability**: service's ability to handle high peak loads
- **Elasticity** means that service can scale up and down
  - Pay for what load is relevant at the time: service-based pricing
  - Usually technically effected by auto-scaling

Scalability: required for elasticity

- Highly scalable systems need to **avoid dependencies**
  - *e.g.*, global lock on a shared data structure kills scalability
  - FYI Python and Ruby both have global interpreter locks! (GILs)
    - Scaling CPU-bound Python needs multi-process not multithreaded

- Making **locks finer-grained** helps scalability
  - However it may lead to more complex software
  - Higher-order problems can be caused: *e.g.*, deadlock, livelock
  - Understand the application: is **resource contention** necessary?

Scalable software designs

- **Partitioning** is a typical approach to scalability
  - *e.g.*, subsets of users and objects **handled by different servers**
  - need to understand **interaction patterns** on systems
    - *e.g.*, internal traffic versus external traffic

- **Caching** of data can greatly assist scalability
  - Workload type needs attention: *e.g.*, **read-only** / **read+write** ?
  - Web originally scaled well because of caching:
    - caching avoided all requests needing to **reach the origin server**

Multiple places to partition workload

- **Application-level**: partition users and/or objects
  - Use application semantics to partition effectively
  - Most common mechanism for 'scale-out' systems
- **Programming language**: partition your application
  - Some procedural languages permit distributed deployment
  - Data-flow programming can optimise distributed execution
    - *e.g.*, operator placement in distributed stream processing systems
- **Server-level**: run code across a large number of CPUs
  - Requires software systems to support multi-processing

Designs and tools for caching

- Most scalable application architectures have **caching**
  - *e.g.*, caching within first tier of **three-tier web architecture**
    - Tiers: (1) front-end server; (2) business logic; (3) back-end storage
    - FYI Varnish is an excellent web caching system—ideal design

- Many caching systems are **key-value databases**:
  - Memcached—open source object caching system (multicore)
  - Redis—open source in-memory key-value store (unicore)
  - Amazon DynamoDB
  - Often systems work in-memory with data snapshots on disk

Scale-out approaches for server types

- **Email**—partition on subsets of mailboxes
  - Efficiency depends on inter/intra-subset interaction patterns
- **Storage**—partition on user accounts
  - …but noting that copying between partitions will be expensive
- **Databases**—'sharding': tables, or sets of columns/rows
  - Also, add scale-out cache for read-only access
- **Web**—design site's content to be cache friendly
  - Use scale-out caching and database systems
- Examine system for behaviours that block scale-out

Elasticity in the cloud context

- Client ensures cloud provider can scale up application
  - At **IaaS**-level: provider knows how to **image and start VMs**
  - At **PaaS**-level: provider understand **components to replicate**
  - **SaaS** software should be **elastic transparently**, if done right
- Other system components also need reconfiguration:
  - Load balancing components need to know **set of workers**
- After scaling up, need to know when to scale back:
  - *e.g.*, use **time-based leases** of resources with periodic renewal

Monitoring is required to effect elasticity

- Not useful knowing the need for scaling up too late
  - *e.g.*, being notified that front-end servers are falling over
- **Monitoring of infrastructure** is required for elasticity
  - Understand the **load** on system components & **rate of change**
- Need to agree heuristics for **scaling actions**
  - *e.g.*, upper-and lower bounds on servers' resource use
- Typical control-system challenges
  - Must not react **too quickly (cost)** or **too slowly (disruptions)**
  - Need to factor in that scaling itself may have a **transition cost**

Examples of Amazon's elastic services

- Recall that EC2 stands for Elastic Compute Cloud
  - EC2 supports **auto-scaling groups** of VMs. Scaling options:
    - maintain count; manual; schedule; on demand;
    - also a predictive option that works with other AWS offerings
- Amazon **Elastic Beanstalk**—PaaS-level orchestration
  - Can include: EC2; S3; load balancers; SNS (notifications); …
- **Elastic MapReduce**—big data frameworks
  - Hadoop; Spark; HBase; Presto (SQLs); Hive (data warehouse)
- **Elasticsearch**—Kibana (visualisation); Logstash