ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 20 Resilience

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

October 30, 2024

Outline

Reading Assignment

- This lecture: 9
- Next Lecture: 9, 10

Outline

Resilience

Retries Revisited

Why Resilience Matters?

- An incident from Amazon
  - Brief failure of a portion of internal network.
  - Some distributed database servers are affected.
  - When network was restored, these servers simultaneously requested their states from the metadata service.
  - The metadata service was overloaded and not able to serve servers that were not affected by the network failure.
  - Servers started a “retry storm” to the metadata service.
  - Engineers had to resolve the incident manually.
- Failures in complex systems never have a single root cause.
  - A failure in a subsystem may trigger a latent fault in another subsystem and cause it to fail.
  - And another, until the whole system goes done.
  - If a subsystem like the metadata service is able to isolate and recover from other failures, more likely the whole system can recover without human intervention.

What is Resilience?

- Resilience is the ability for a system to withstand and recover from errors and failures.
  - The system can continue operating correctly when some subsystem fails, possibly at a reduced level.
  - Instead of failing completely.
- Resilience is not reliability
  - Resilience allows a system to degrade its performance to cope with failures.
  - Reliability requires a system to behave as expected for a given time interval, e.g. to meet dynamic demand via scalability.
  - Resilience, together with scalability, loose coupling, manageability, and observability, are factors contributing to the reliability of the system.

Understand System Failures

- A system consists of components.
  - Each component, or subsystem, is also a system by itself, consisting of smaller components, and so on.
- Progress of system failure
  - All systems contain faults, e.g. bugs, and have limitations. Under certain conditions, errors are produced.
  - Errors are those system behaviors differ from intended ones. If not handled properly, errors cause failures.
  - A system with failures can no longer provide correct service.
  - Failure at subsystem level becomes fault at system level.

Cascading Failures

- Cascading failure is a common mode of failure as shown in the incident from Amazon.
- Failures in subsystems lead to a positive feedback.
  - Requests from database servers cause metadata servers to time out, which in turn cause more database servers to fail and to generate even more requests.
  - Eventually all attempts to compensate for failed subsystems fail and the system fails.
  - Spread very quickly, often in a few minutes.

Overload

- Overload is a classic cause of such cascading failures.
- Every system has certain amount of redundancy, especially for scalable system.
- A failed node doesn't cause system failure as its load can be redistributed to remaining nodes.
- However, if the increased load causes one of the remaining nodes to fail, then loads on remaining nodes will further increase.
- The positive feedback causes the failure to propagate too quickly so scalability doesn't have enough time to kick in to decrease loads on nodes.

Preventing Overload

- Be defensive: services should reject requests beyond their functional limitations.
- Throttling: make sure no particular user consumes more resources than they would reasonably require.
  - Isolate errors to subsystems that send those requests.
- Load shedding: intentionally drop requests.
  - Limit errors to this subsystem by not sending more requests.
- Graceful service degradation
  - Not possible for all services but for services that could, more gracefully than simply drop the requests.
  - E.g. serve images at lower resolution, videos at smaller bit rate, and data from cache that could be stale.

Load Shedding Implementation

```
const MaxQueueDepth = 1000
func loadSheddingMiddleware(next http.Handler) http.Handler {
  return http.HandlerFunc(func (w http.ResponseWriter, r *http.Request) {
    // CurrentQueueDepth is fictional and for example purposes only.
    if CurrentQueueDepth() > MaxQueueDepth {
      log.Println("load shedding engaged")
      http.Error(w, err.Error(), http.StatusServiceUnavailable)
      return
    }
    next.ServeHTTP(w, r)
  })
}
```

- Load shedding can usually be implemented via a queue.
  - Large queue depth (length) implies overload.
- It is better to have some clients receiving error codes than causing most of them to timeout.
  - We cannot afford to waste more server resources processing requests that are going to be timeout soon.

Outline

Resilience

Retries Revisited

Retries

- Overload prevention applies to services.
  - Make them defensive for errors from clients
- Clients can take proactive actions when errors are observed.
  - Make errors easier to handle for services so that failures are less likely to happen.
  - Not all services are defensive and prepared for those errors.
- Simple retries won't work.

```
res, err := SendRequest()
for err != nil {
  res, err = SendRequest()
}
```

  - A lot of clients doing the same are spamming the service, causing a "retry storm".
  - Overall retrying frequency is usually limited by the network bandwidth to the service.

Simple Backoff

```
res, err := SendRequest()
for err != nil {
  time.Sleep(2 * time.Second)
  res, err = SendRequest()
}
```

- What if we ask clients to wait a while before retrying?
  - Cannot wait for too long as service may be back online soon.
- Overall retrying frequency will be greatly reduced.
  - However, it still grows as number of clients grow.

Exponential Backoff

```
res, err := SendRequest()
base, cap := time.Second, time.Minute
for backoff := base; err != nil; backoff <<= 1 {
  if backoff > cap {
    backoff = cap
  }
  time.Sleep(backoff)
  res, err = SendRequest()
}
```

- Clients can wait longer as more errors are observed.
  - Double the wait time until an upper bound is reached.
- Overall retrying frequency will be reduced as service takes more time to recover.
- However, if a service fails, very likely many clients observe the error at the same time and follow the same retry schedule.
  - Lead to spikes in retries that may be difficult to cope.

Randomized Exponential Backoff

```
res, err := SendRequest()
base, cap := time.Second, time.Minute
for backoff := base; err != nil; backoff <<= 1 {
  if backoff > cap {
    backoff = cap
  }
  jitter := rand.Int63n(int64(backoff * 3))
  sleep := base + time.Duration(jitter)
  time.Sleep(sleep)
  res, err = SendRequest()
}
```

- Adding a random jitter allows clients to send retries at different times.
- Make sure to seed the random number generator differently at the beginning of your program so the clients don't follow the same random sequence.

Other Proactive Mechamisms

- Circuit breaker: avoid retrying after certain amount of errors.
  - Don't waste resources and clog network – give more time for services to come back.
- Timeouts: allow clients to give up progresses – fail fast
  - A client may depend on many services to complete a task.
  - If one service fails, the client can release resources that are obtained from other services, reducing overall system load.
  - Instead of holding resources that cannot be immediately used.
- Don't forget that in order to handle errors properly, services must be idempotent.
  - Otherwise retries and restarts may cause additional faults to happen as integrity of data cannot be guaranteed.

Summary

- Services and clients can work together to prevent cascading failures.