ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 21 Manageability

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

November 4, 2024

Outline

Reading Assignment

- This lecture: 9, 10
- Next Lecture: 11

Outline

Health Check

Manageability

Service Redundancy

- Duplicate critical components or functions to improve reliability.
  - Deploy component to multiple server instances.
  - Ideally across multiple zones or even across multiple regions.
- Autoscaling helps to maintain certain level of redundancy as demand fluctuates. However, it takes time to start an instance so there should be room for redundancy without scaling.
- Fault masking: a system fault is invisibly compensated for without being explicitly detected.
  - Without careful planning, redundancy will lead to fault masking that conceals progressive faults.
  - E.g. loss of nodes for a service are not observed until all nodes are lost, causing a sudden and catastrophic outcome.

Health Check: Pull Model

- An API endpoint for clients to decide if a service instance is alive and healthy.
  - For clients that are aware of the redundancy, e.g. Cassandra and Kafka clients, as well as load balancers, monitoring services, service registries, etc.
- Usually implemented as an HTTP endpoint for simplicity.
  - E.g. available from /health that returns 200 OK for a health service or 503 Service Unavailable otherwise.
- Trade-offs between latency and scalability.
  - Frequent health checks may lead to inefficiency, in particular when there are a lot of services and a lot of clients.
  - For longer intervals between health checks, clients may miss critical information like when a service actually dies.

Health Check: Push Model

- Let services send health information to clients.
  - Periodically, e.g. heartbeats.
  - Proactively when health status changes.
- A more complex system.
  - Where are the clients?
  - What if there are more information than what a single client can handle?
  - Use message queues to decouple services from clients and to handle scalability better.
- What does it mean for an instance to be "healthy"?
  - Is a response of 200 OK from /health sufficient for both the clients and the instance?

“Healthy” Instances

- Simple definition: “healthy” means “available”
  - But availability of instances may be impacted by availability of services these instances depending on.
  - Restarting/replacing these instances won't help at all.
- Need to make choices depending on services.
  - Liveness checks: a simple response to indicate the service instance is reachable and responding, confirming correctness of network, security, and service configuration.
  - Shallow health checks: ensure local resources (memory, CPU, disk etc.) and dependencies (monotoring etc.) are available so the service instance is likely to be able to funciton.
  - Deep health checks: inspect the ability to interact with other subsystems, identifying potential issues like networking – however, it is costly and it is possible to have all instances reporting unhealthy.

Outline

Health Check

Manageability

Manageability

- Change behaviors without having to recode and redeploy.
  - By yourself or by someone else.
- Manageability allows to make changes from outside.
  - Maintainability allows to make changes from inside, usually by updating code.
- Manageability for complex systems.
  - Make configuration and control options available.
  - Use monitoring, logging, and alerting to identify components that require management, e.g. misconfigured components.
  - Manage deployment by updating, rolling back, and scaling system components.
  - Discover available services.

Application Configuration

- Configuration: anything likely to vary between environments like staging, production, developer, etc.
- Store configuration in the environment.
  - Configuration should be strictly separated from the code.
  - Configurations should be stored in version control – make it possible to inspect, review, rollback, and troubleshoot changes.
- Configuration practices
  - Command-line flags and environment variables: use start-up scripts for version control.
  - Configuration files: use standard format like JSON and YAML.
  - Simplify and minimize configuration effort by using default values that are reasonable and unsurprising.

Configuring with Environment Variables

- Use environment variables

```
name := os.Getenv("NAME")
place := os.Getenv("CITY")
fmt.Printf("%s lives in %s.\n", name, place)
```

- Distinguish between an empty value and an unset value.

```
if val, ok := os.LookupEnv(key); ok {
  fmt.Printf("%s=%s\n", key, val)
} else {
  fmt.Printf("%s not set\n", key)
}
```

Configuring with Command-Line Arguments

```go
package main
import (
  "flag"
  "fmt"
)
func main() {
  strp := flag.String("string", "foo", "a string")
  intp := flag.Int("number", 42, "an integer")
  boolp := flag.Bool("boolean", false, "a boolean")
  flag.Parse() // Call flag.Parse() to execute command-line parsing.
  fmt.Println("string:", *strp)
  fmt.Println("integer:", *intp)
  fmt.Println("boolean:", *boolp)
  fmt.Println("args:", flag.Args())
}
```

- Use the `flag` package for command-line flags.
  - Register with types, default values, and short descriptions
  - Map flags to variables.

Configuring with Command-Line Arguments (cont.)

```
$ go run . -help
Usage of /var/folders/go-build618108403/exe/main:
  -boolean
    a boolean
  -number int
    an integer (default 42)
  -string string
    a string (default "foo")
$ go run . -boolean -number 27 -string "A string." Other things.
string: A string.
integer: 27
boolean: true
args: [Other things.]
```

Configuring with JSON Files

```go
type Config struct {
  Host string
  Port uint16
  Tags map[string]string
}
func EncodeJson() {
  c := Config{
    Host: "localhost",
    Port: 1313,
    Tags: map[string]string{"env": "dev"},
  }
  bytes, err := json.Marshal(c)
  fmt.Println(string(bytes))
  // {"Host":"localhost","Port":1313,"Tags":{"env":"dev"}}
}
```

- Use `json.Marshal()` to encode any struct as JSON string.
  - Only public fields (begin with a capital letter) are encoded.

Configuring with JSON Files (cont.)

- Use `json.Unmarshal()` to decode JSON string into a struct.

```
c := Config{}
bytes := []byte(`{"Host":"127.0.0.1","Port":1234,"Tags":{"foo":"bar"}}`)
err := json.Unmarshal(bytes, &c)
```

  - Missing fields will have a default value of zero or empty.
  - Extra fields will be ignored.
- Use `interface{}` to decode JSON string as it is.

```
var f interface{}
bytes := []byte(`{"Foo":"Bar", "Number":1313, "Tags":{"A":"B"}}`)
err := json.Unmarshal(bytes, &f)
fmt.Println(f)
// map[Number:1313 Foo:Bar Tags:map[A:B]]
```

  - `f` has a type of `map[string]interface{}`, enabling a recursive tree-like data structure for arbitrary JSON data.
- Mapping between struct and JSON string can be customized via struct field tags (like annotations in Java).
- YAML strings are handled similarly.

Additional Considerations for Application Configuration

- Should we reload a configuration file if it changes?
  - No for simplicity: kill and restart
  - Yes for no downtime: use polling and hashing to watch for updates, or use OS filesystem notifications.
- Use more advanced (and more complicated) libraries like Cobra and Viper as a complete configuration solution.

Feature Management

- Allow control of program features and flows.
  - Enable experimental features conditionally for testing.
  - Adjust features like algorithms according to use cases.
- Feature flags: enable/disable features via configurations
  - Manage different code versions in one code base, encouraging smaller and faster iterations.
  - Integrate with resilience patterns like circuit breaker to automatically turn on and off.
  - Control feature rollouts to specific users.
- Scripting: complete control of features and flows.
  - For very complicated applications, e.g. mods for games and Tcl scripts for EDA tools.
  - Separate execution flow and features from program binary.
  - Very flexible – nevertheless, it blurs the boundary between manageability and maintainability.

Summary

- Make your application configurable via command-line flags and environment variable, as well as configuration files.