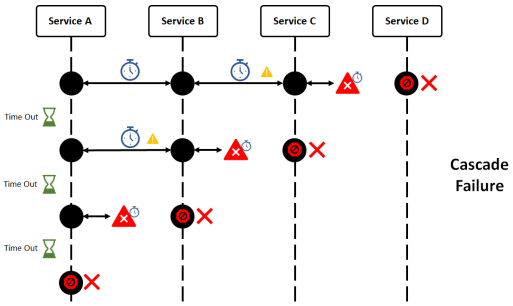
Circuit Breaker

Servisno orijentisane arhitekture

Univerzitet u Novom Sadu
Fakultet tehničkih nauka

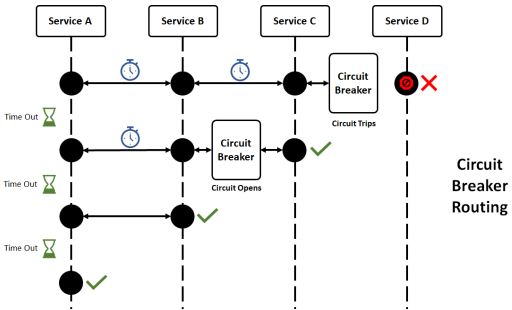
Problem

- Kada korisnički zahtev stigne do servisa na obradu, on nekad mora da komunicira sa drugim servisima kako bi uspešno obavio neku akciju
- Pozvani servis je možda nedostupan ili predugo obrađuje zahtev, što izaziva kašnjenje i u servisu koji je inicirao poziv
- Pored toga, servis koji poziva druge servise (koji su nedostupni) može istrošiti svoje resurse i time i on postati nedostupan
- Ovakva reakcija može se propagirati kroz nekoliko servisa kada imamo više ulančanih poziva, što može dovesti do kaskadnog otkaza čitavog (ili velikog dela) sistema



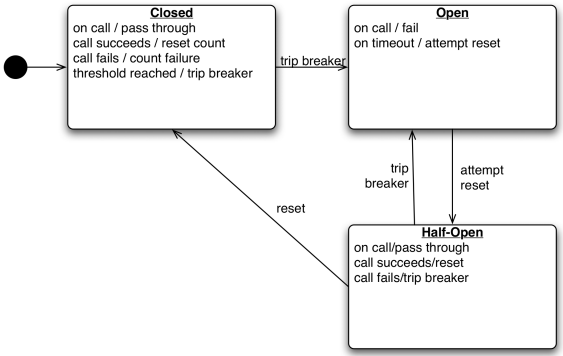
Circuit Breaker

- Ideja iza Circuit Breaker (osigurač) šablona slična je kao i kod električnog osigurača
- Circuit Breaker ponaša se kao proxy prilikom komunikacije sa drugim servisima
- Kada broj neuspešnih zahteva prekorači definisanu granicu, osigurač blokira dalje slanje zahteva nedostupnom servisu, sve dok neki vremenski period ne istekne, kada ponovo pokuša slanje
- Ovaj šablon omogućava nam da ne dozvolimo servisu da inicira komunikaciju koja će verovatno biti neuspešna



Stanja

- Moguća stanja osigurača su:
  - **zatvoren** - propušta zahteve
  - **otvoren** - blokira slanje zahteva
  - **poluotvoren** dozvoljava mali broj zahteva kako bi proverio da li je servis počeo da radi ispravno



Konfiguracija osigurača I

- Opisanu logiku osigurača za nas implementira gobreaker biblioteka, naš posao je da ispravno definišemo sve parametre i iskoristimo kreirani osigurač prilikom slanja zahteva nekom servisu
- Najznačajniji atributi Circuit-Breaker-a koje treba da definišemo su:
  - **MaxRequests** - broj zahteva koje će osigurač dopustiti kada je u poluotvorenom stanju
  - **Timeout** - vremenski period koliko će osigurač ostati u otvorenom stanju, nakon čega prelazi u poluotvoreno stanje
  - **ReadyToTrip** - funkcija koja se poziva kada poziv ne uspe dok je osigurač u zatvorenom stanju, ako vrati true, prelazi u otvoreno stanje
  - **Interval** - vremenski period nakon kog Circuit Breaker briše zabeleženi broj uspešnih i neuspešnih zahteva

Konfiguracija osigurača II

- Istorija uspešnosti zahteva, koja se prosleđuje ReadyToTrip funkciji ima sledeću strukturu:

```
type Counts struct {
    Requests             uint32
    TotalSuccesses       uint32
    TotalFailures        uint32
    ConsecutiveSuccesses uint32
    ConsecutiveFailures  uint32
}
```

Konfiguracija osigurača III

- Na sledeći način kreiramo Circuit Breaker:

```
cb := gobreaker.NewCircuitBreaker(
    gobreaker.Settings{
        Name: "cb",
        MaxRequests: 1,
        Timeout: 2 * time.Second,
        Interval: 0,
        ReadyToTrip: func(counts gobreaker.Counts) bool {
            return counts.ConsecutiveFailures > 0
        },
        OnStateChange: func(name string, from gobreaker.State, to
↪ gobreaker.State) {
            log.Printf("Circuit Breaker '%s' changed from '%s' to
↪ '%s'\n", name, from, to)
        },
    },
)
```

Poziv osigurača I

- Kada pozivamo servis, treba da pozovemo metodu Execute nad osiguračem zaduženim za taj servis/endpoint, i unutar nje izvršimo poziv servisu
- Poslednja povratna vrednost funkcije je greška, na osnovu koje osigurač povećava brojač uspešnih, odnosno neuspešnih zahteva
- Ako je osigurač u otvorenom stanju, prosleđena funkcija neće se izvršiti već ćemo odmah dobiti grešku

Poziv osigurača II

```
bodyBytes, err := cb.Execute(func() (interface{}, error) {
        client := http.Client{Timeout: 3 * time.Second}
        resp, err := client.Get("http://localhost:8000/timeout")
        if err != nil {
                return nil, err
        }
        defer resp.Body.Close()
        if resp.StatusCode != http.StatusOK {
                return nil, errors.New("error status: " + resp.Status)
        }
        bodyBytes, err := io.ReadAll(resp.Body)
        if err != nil {
                return nil, err
        }
        return bodyBytes, nil
})
```