Design ed Implementazione del Back End del Sistema di Warehouse di SeismoCloud

SAPIENZA
UNIVERSITÀ DI ROMA

Andrea Stella
1869065

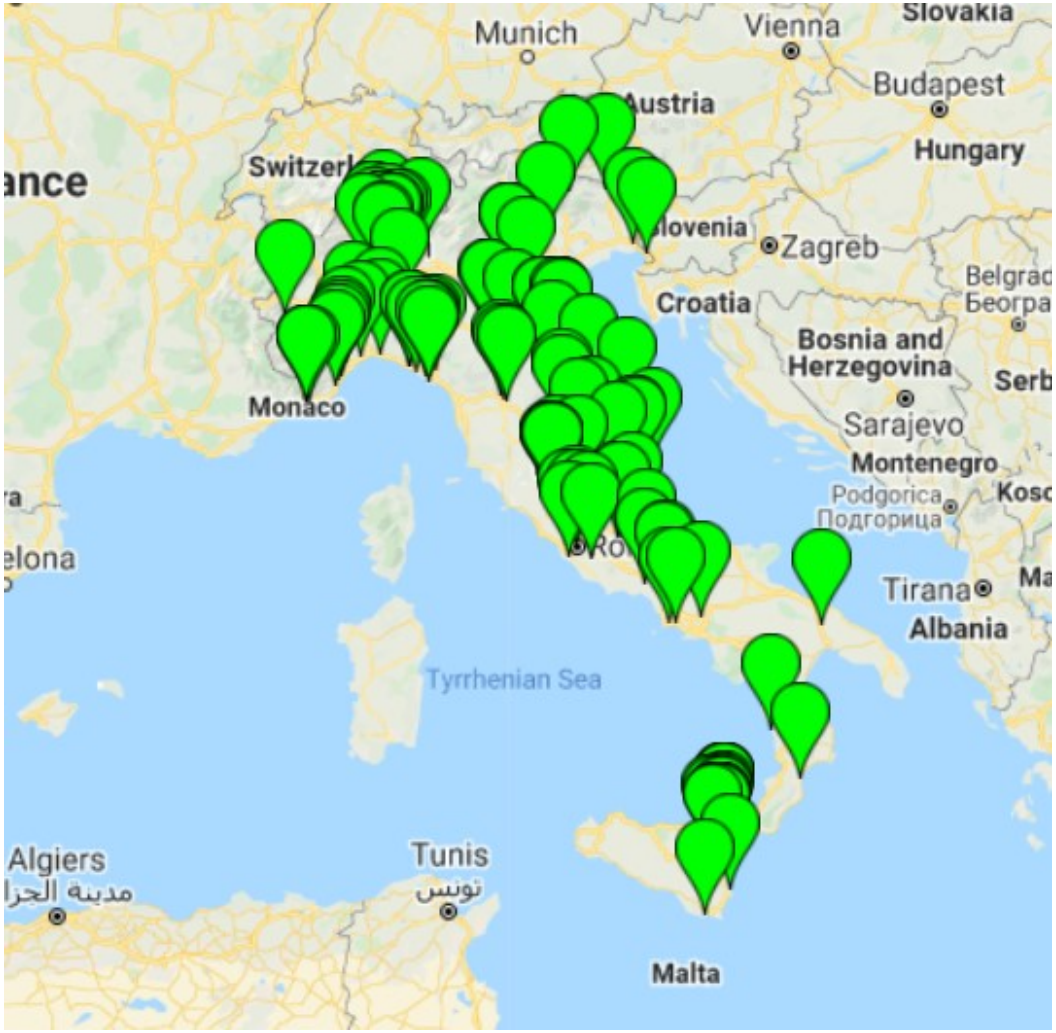
Dipartimento di Informatica,
Università di Roma "La Sapienza"

Ottobre 2021

Il progetto **SeismoCloud**

- SeismoCloud è un **Earthquake Early Warning system**
- Per rilevare le scosse sismiche sono impiegati **sensori low cost**
- Una notifica può far guadagnare **dai 2 ai 20 secondi** prima di una scossa sismica

I sismometri attivi



Obiettivo del tirocinio

- Realizzazione del sistema di Warehouse (raccolta e fruizione dati)
- Design ed implementazione del Back End del Warehouse

Analisi del problema

I dati

- I sensori inviano quotidianamente molti dati
- I dati non sono a disposizione degli utenti

La quantità di dati ricevuta

- I sensori inviano:
  - temperatura
  - scossa sismica rilevata
  - RSSI
  - etc

  per un totale di 9 dati diversi.

- I dati sono inviati ogni 5 minuti. Le trasmissioni in 24 ore, considerando ogni dato singolarmente e 100 sismometri, sono quindi:

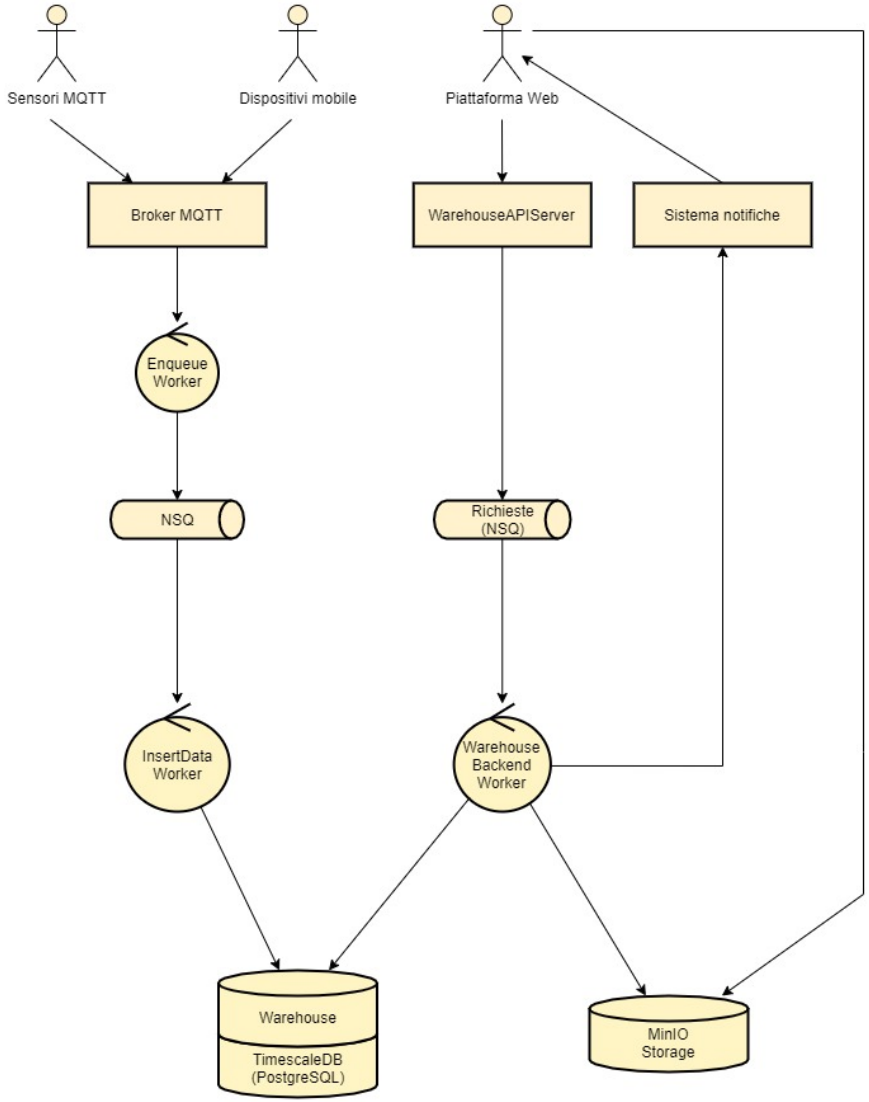
$$dailyData = \frac{timeFrame}{sendingInterval} * 9\ elems * 100\ seismos = \\ = \frac{24 * 60\ minutes}{5\ minutes} * 9 * 100 = 259.200 \quad (1)$$

Fornire i dati agli utenti

- Utile fornire l'accesso per analisi dei dati ed altre applicazioni
- Permettere la richiesta dei dati su un ampio time frame
- Essendo una grande quantità di dati, le richieste possono essere:
  - Elaborate in modo sincrono - l'utente deve attendere
  - Elaborate in modo asincrono - l'utente può svolgere altri task nel mentre

Design della soluzione

Il Sistema di **Warehouse**



Le tecnologie impiegate

- NSQ: sistema per la distribuzione di messaggi
- TimescaleDB: Database per time-series data
- MinIO: storage S3 compatibile

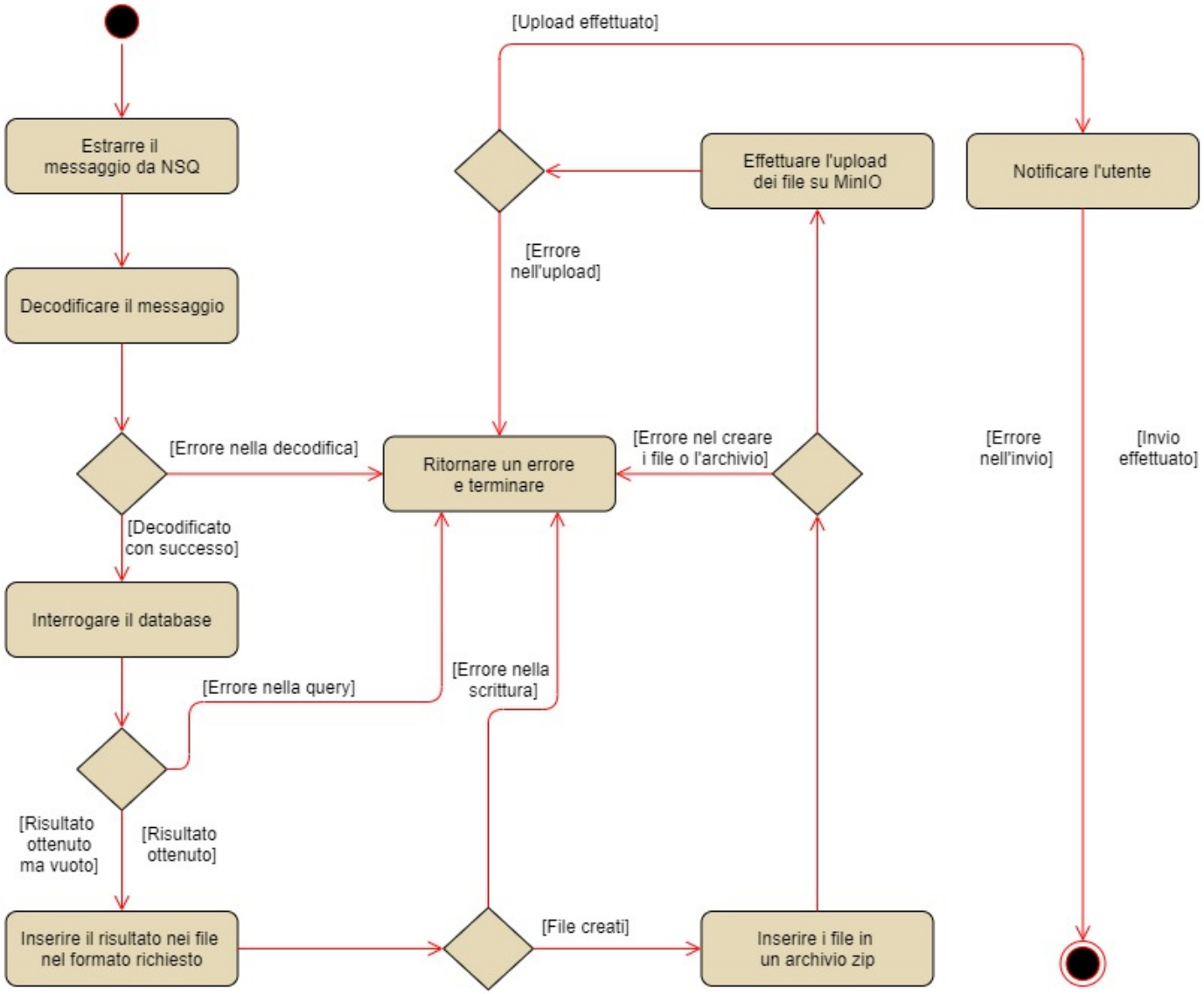
Richiesta di un utente

- L'utente effettua una richiesta tramite una API
- La richiesta viene codificata in un messaggio che entra in NSQ
- L'elaborazione è delegata al WarehouseBackendWorker, lato Back End del sistema

Il formato del messaggio


```
{
    "message": {
            "ID": "ABCDEFGHIJKLMNO",
            "Attributes": {
                "temperature": true,
                "rssi": true,
                "threshold": false,
                "quake": false
            },
             "Seismometers": [
                "HD341FD"
            ],
            "Location": {
                "Latitude": 0,
                "Longitude": 0,
                "Radius": 0
            },
            "From": "1984-04-12T23:20:20.52",
            "To": "1985-04-12T23:20:20.52",
            "Tg": "telegramusername",
            "Mail": "email",
            "Output": {
                "CSV": true,
                "GeoJSON": false,
                "HDF5": false,
                "MiniSEED": false
            },
            "Timestamp": "2001-04-12T04:20:50.56"
    }
}
```

Il **WarehouseBackendWorker**



Implementazione del Back End

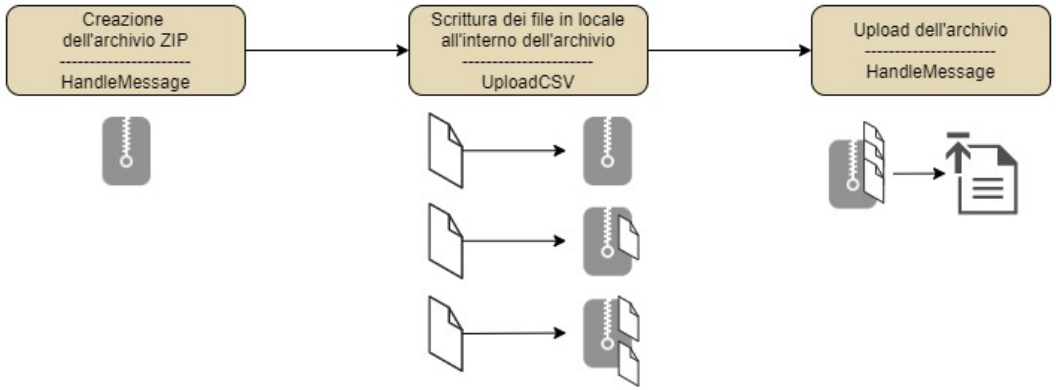
Go e SeismoCloud

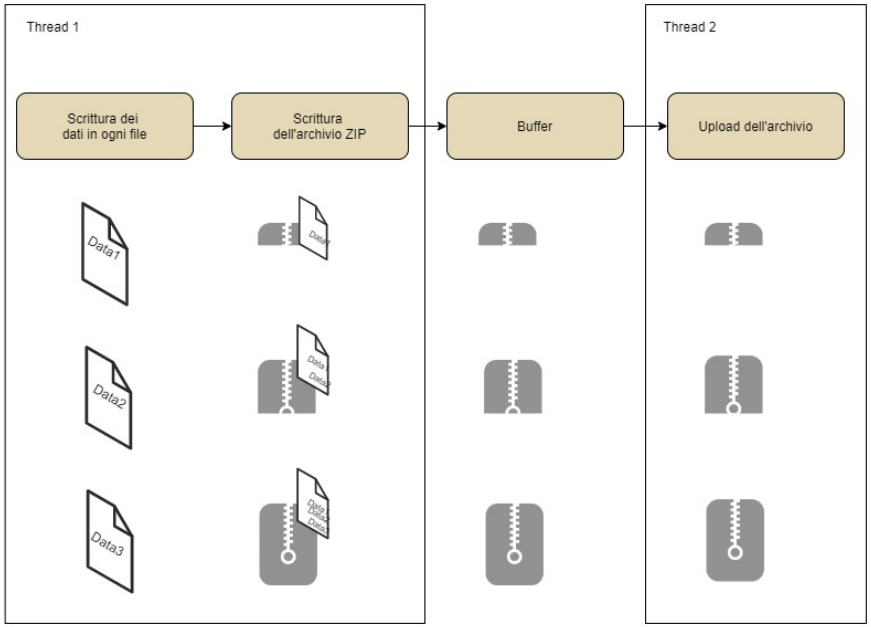
- Go: linguaggio Open Source, compilato e staticamente tipizzato
- Meccanismo di condivisione del codice sotto forma di package
- In SeismoCloud Go è utilizzato per il Back End

Implementazione delle funzionalità principali

```go
func (worker *_warehouseWorker) GetResultsFromDevices(ctx context.Context,
  userRequest Query, wg *sync.WaitGroup) error {
  var err error
  pipeReader, pipeWriter := io.Pipe()
  writer := zip.NewWriter(pipeWriter)

  // Chiusura delle risorse con defer ...

  // Operazioni preeliminari ...

  // Funzione asincrona per il consumo dei dati nel lato reader della Pipe.
  wg.Add(1)
  go func() {
   defer wg.Done()
   contentType := "application/zip"
   if err = worker.fs.PutWarehouseResults(ctx, userRequest.ID+".zip",
   pipeReader, -1, minio.PutObjectOptions{ContentType: contentType}); err != nil {
            // Log ed errori ...
   }
  }()

  // Passi da compiere per ogni attributo richiesto.
  if userRequest.Attributes.Temperature {
        rows, err := worker.GetTemperatureQuery(ctx, userRequest, queryArgs)
        if err != nil {
                // Log ed errori ...

        }
        if err = worker.CheckContextStatus(ctx); err != nil {
            // Log ed errori ...

        }
        if err = worker.UploadQueryResultsByDevices(userRequest, rows,
        writer, "temperature"); err != nil && err != ErrNoRows {
                // Log ed errori ...

        }
        if err = worker.CheckContextStatus(ctx); err != nil {
            // Log ed errori ...
        }
  }
```

La **Streaming Pipe**





Conclusione

Risultato ottenuto

- Creazione di un sistema per la memorizzazione e fruizione dati
- Progettazione ed implementazione del Back End del sistema
- Prodotto funzionante, testato localmente e manutenibile

Sviluppi futuri

- Come gli utenti useranno i dati
- Estensione delle funzionalità del sistema
- Creazione di nuovi package, come HDF5, in Go

Grazie!