

Scalable Call Metering System using Akka

Internship Project at Nagarro

by

Aditya Gupta and Kamaldeep Verma

addiegupta@gmail.com | 94kamal@gmail.com

Table of Contents

- I. Introduction to Scalable Call Metering System using Akka
- II. System Architecture
- III. System Requirements
- IV. Running the project
- V. Testing the project

Introduction to Scalable Call Metering System using Akka

Scalable Call Metering System is a backend system that carries out the metering of user calls in real time. It is an Actor Model and REST API based application which can be used by telecom companies for managing and metering user calls in real time. The system is developed using Actor Model Programming with Akka, keeping in view the guidelines of reactive programming. Hence it meets the non-functional requirements such as **scalability, resilience, fault tolerance and self healing** which are some of the key features provided by Akka.

Being a message driven toolkit, Akka functions as a network of actors (which are essentially separate entities) that perform operations by sending and receiving messages between each other. Each actor has access to its own state only and all operations are performed asynchronously.

System Architecture

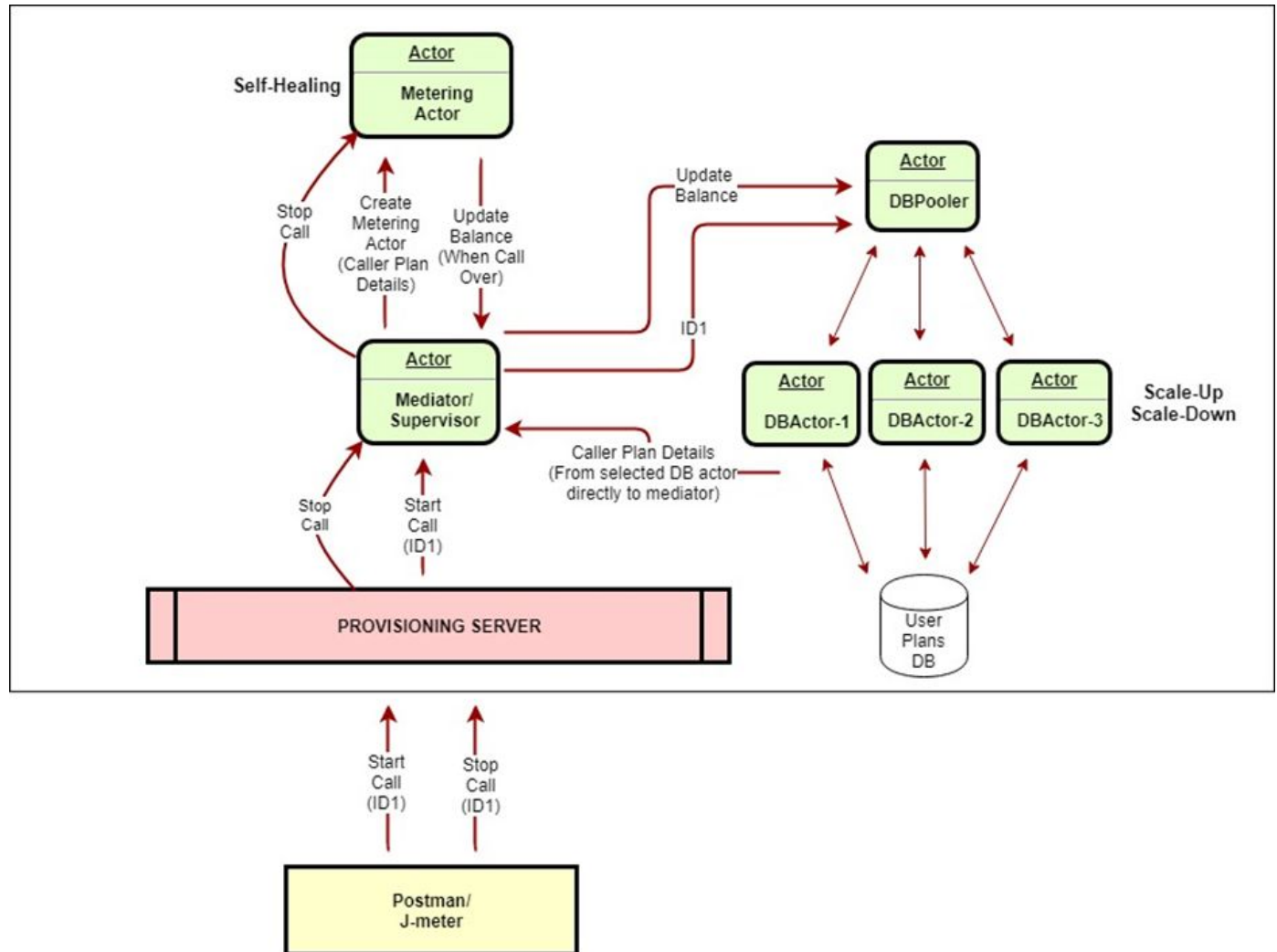


Fig. 1 Architecture of Scalable Call Metering System using Akka

Fig 1 explains the complete architecture of the system. It consists of a Web Application which can be accessed using REST based APIs. The **provisioning server** acts as the interface between the API calling tool and the metering system which comprises of the entire Actor System network. The complete technical flow of the system is explained below.

Actors message flow:

- **Starting a call**

1. Whenever the provisioning server is up and running an **Actor System and DB Pooler Actor** are created and the server listens for requests.
2. The DB Pooler is a router actor which helps in scaling and routing of requests to DB Actor. It creates and manages the scheduling of requests to DB Actors at run time. Random Router Pooler with resizable capabilities has been used which makes the system vertically scalable. The router configuration can be changed in **application.conf** file present in resources directory (other configurations are also provided in the configuration file for switching between routers) The name of the router can be changed in WebServer.scala file.
3. Whenever a start call is to be initiated, a REST based web request (Start Call (ID)) is sent to the provisioning server.
4. The provisioning server contains the Actor System which spawns a new **Mediator Actor** (named as mediator-<id>) for the particular ID. It then sends **StartCall** message to mediator and passes it the reference of the DB Pooler.
5. Now the mediator actor will act as a supervisor for its children. Mediator actor doesn't have the current plan details of the user. Hence it will request the balance from the User Plans Database through DB pooler by sending message **FindPlanById** to DB pooler which will select a Database Actor (**DbActor**) and forward the message with Mediator actor reference so that DB Actor can reply directly to the mediator with the data needed.
6. The DbActor will receive the message sent by the mediator. It will fetch the plan details from the async database (**using Slick**) and send a Future (an enclosure around the database response which denotes that the value will be available at a later point in time) to the mediator actor as message **ReceivePlanDetails**.

7. The RecievePlanDetails in Mediator actor will wait for the future to complete and then spawn a new **Metering Actor** and pass it the balance of the user. Each user ID will have its own Mediator and Metering actor for independent processing.
8. Now the metering actor starts the balance meter automatically upon its creation with the data balance. The timer starts running and decreases the balance on every tick.
9. If so happens that the balance gets over before the **stop call request** from the user, then the metering actor will stop the timer, send **UpdateBalance message** to the parent(Mediator) and destroy itself, freeing up resources.
10. On receiving UpdateBalance message from metering actor, the Mediator actor will send **UpdateBalance** to DBPooler to update the balance and then will destroy itself.

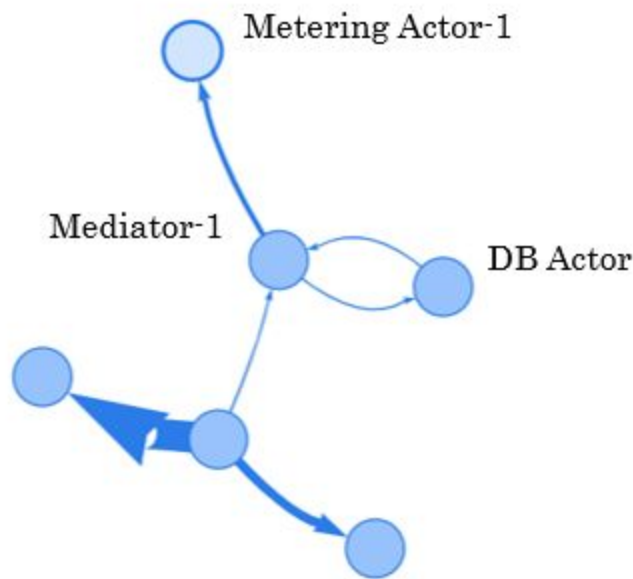


Fig 2 Flow for Starting a call

Fig 2 shows communication between Mediator 1 and DB Actor(which has been assigned to mediator via DB Pooler) and a message sent to Metering Actor that starts the metering process.

- **Stopping a call**

1. Whenever a stop call is to be initiated, a REST based web request (Stop Call (ID)) is sent to the provisioning server.
2. Now the actor system searches for the mediator actor running corresponding to the ID of user(**mediator-<id>**).
3. If the mediator is not found, then the call is not running and hence cannot be stopped. Therefore the system responds with the exception of invalid ID.
4. Else the Actor System gets reference of the mediator actor and sends **EndCallMediator message** to the mediator actor.
5. The mediator actor on receiving the EndCallMediator message, sends **EndCallMeter** message to its corresponding metering actor.
6. Upon receiving EndCallMeter message from mediator, the metering actor stops the timer, sends **UpdateBalance message** to the parent (Mediator) and destroys itself.
7. On receiving UpdateBalance message from metering actor, the Mediator actor will send **UpdateBalance** to DBPooler to update the balance and then destroy itself.

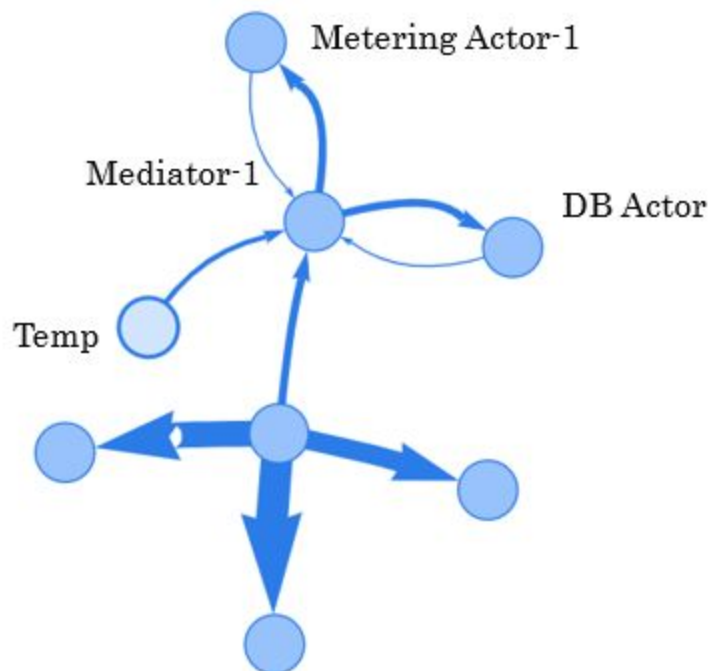


Fig 3 Flow for stopping a call

Fig 3 shows Mediator sending another message to meter (thick arrow) and meter replying with updated balance value. This is followed by communication with DB Actor. Temp Actor contains internal functioning of the Akka system.

- **Actor Crashing**

1. The metering actor is designed to exhibit self healing and fault tolerant characteristics.
2. Whenever the metering actor is crashed, the supervisor (mediator) handles the exception and takes necessary actions accordingly.
3. In the current implementation, the mediator actor resumes the metering actor from its previous state. All the other exceptions that can occur can be handled separately.
4. In this implementation, currently only metering actor exhibits self healing but all the actors of the system can be designed to do the same by using **Supervisor Strategy** mechanism.

System Requirements

HW Requirements

- Hard Disk - 150 MB
- RAM - 2GB

SW Requirements

- JDK v8
- Scala 2.12.6
- SBT 1.2.8
- Akka 2.4.19
- AkkaHttp: 10.0.9
- Visual mailbox: <http://www.github.com/ouven/akka-visualmailbox>
 - This is a visualisation tool that displays actors and actor message flows in real time. It is used only for observing the Actor System and is an **optional requirement**
- Apache-jmeter-5.1.1 (Any basic tool which can be used for Rest API Testing)
- Postgresql-11.3-4
- IntelliJ IDEA (Latest Stable Version)- optional

Running the project

- First a Postgresql database is to be created with the following configuration.

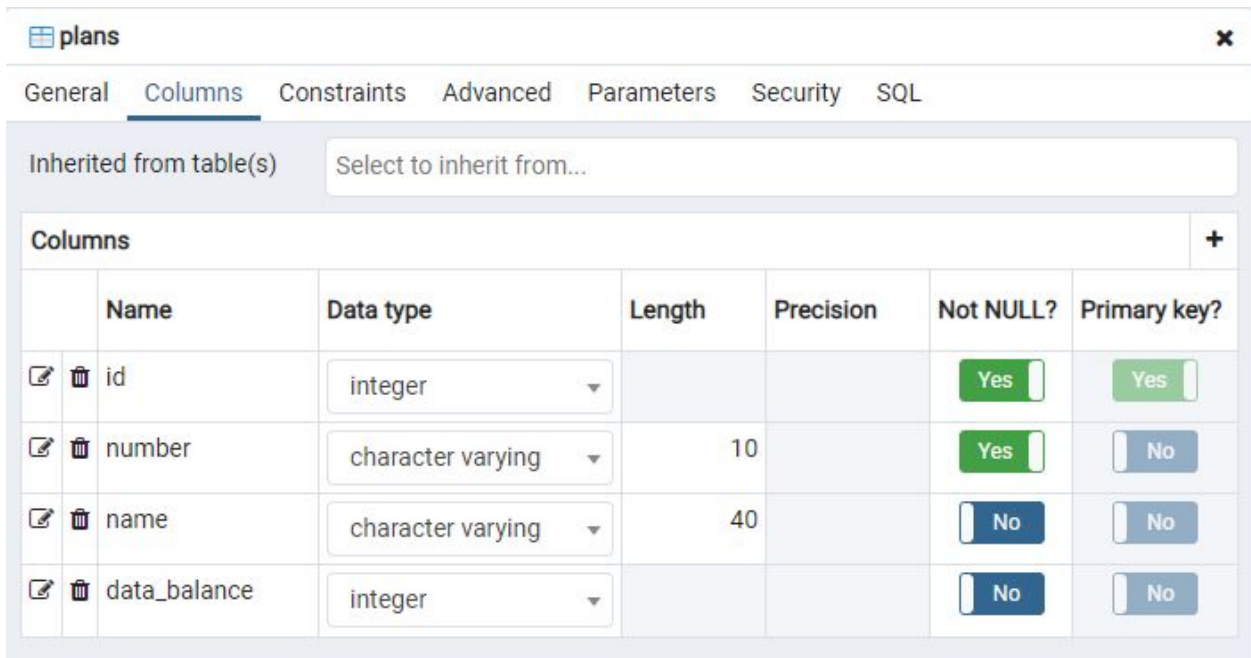
Note: Postgresql database is present at the git repo and can be directly imported. Or it can manually be created with the following configurations:

```
url
="jdbc:postgresql://localhost:5432/metering_db"
  user = "postgres"
  password = "root"
```

- In the metering_db, create plans table with the following schema

```
CREATE TABLE plans (
    id SERIAL PRIMARY KEY,
    number VARCHAR(10),
    name VARCHAR(40),
    data_balance INTEGER
);
```

- Add sample values to the plans table



The screenshot shows the configuration for a table named 'plans'. The 'Columns' tab is selected, displaying a table with the following columns:









	Name	Data type	Length	Precision	Not NULL?	Primary key?
 	id	integer			<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
 	number	character varying	10		<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
 	name	character varying	40		<input type="checkbox"/> No	<input type="checkbox"/> No
 	data_balance	integer			<input type="checkbox"/> No	<input type="checkbox"/> No

Fig 4 Schema of the plans table

- Pull the source code from its GitHub repository.
- **Command Line Instructions:**
 - Run the command:

```
$ sbt run
```
- **IDE Instructions**
 - Import the project using IntelliJ IDEA(or a preferred IDE)
 - Wait for the sbt project to get updated then build the project.
 - After successful build, setup the configurations for running the WebServer file.
 - Edit the configuration to export the logs to an external file
 - Run the file WebServer.scala.
- The Server is now ready to serve requests.
- API calls can be made using JMeter or other API tools.
- For visualising the actor flow, akka-visualmailbox library can be used which has been configured inside this project. Instructions for starting the visualizer can be found at its repository.

Testing the project

There are three REST based web services for testing the project, any one of which can be used for testing the application.

1. Apache Jmeter (for automation)
2. Postman
3. Chrome/Mozilla/Any web browser

The Web Services(Endpoints) to test the project are given below.

Protocol : http

Server Name or IP : 127.0.0.2

Port Number : 8181

1. Starting the Call

Method : GET

Path : start-call?id=<id>

Complete URI : <http://127.0.0.2:8181/start-call?id=<id>>

2. Stopping the Call

Method : GET

Path : stop-call?id=<id>

Complete URI : <http://127.0.0.2:8181/stop-call?id=<id>>

3. Crashing the Metering Actor (for testing fault tolerance/Self Healing property)

Method : GET

Path : crash-meter-actor?id=<id>

Complete URI : <http://127.0.0.2:8181/crash-meter-actor?id=<id>>

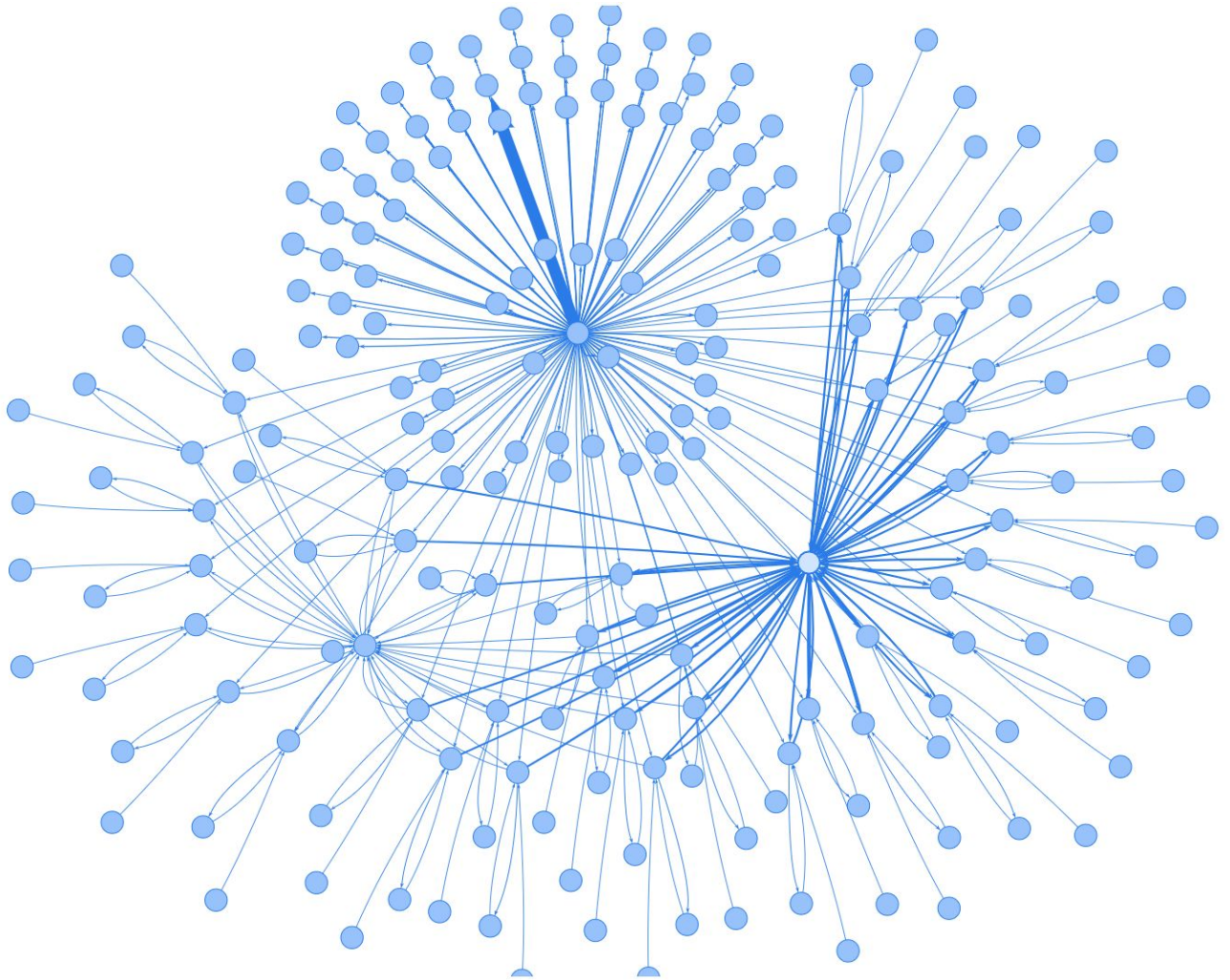


Fig 5 A visualisation of the Actor network under a slightly moderate load