Assignment A-4

Problem Definition: In an embedded system application Dining Philosopher's problem algorithm is used to design a software that uses shared memory between neighboring processes to consume the data. The Data is generated by different Sensors/WSN system Network and stored in MongoDB (NoSQL). Implementation be done using Scala/ Python/ C++/ Java. Design using Client-Server architecture. Perform Reliability Testing. Use latest open source software modeling, Designing and testing tool/Scrum-it/KADOS, No SQLUnit and Camel.

Learning Objective:

1. Implementation of the problem statement using Object oriented programming.
2. Solve Dining Philosopher's using mongoDB.

Theory:

Introduction

In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this

problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

- Think until the left fork is available; when it is, pick it up;
- Think until the right fork is available; when it is, pick it up;
- When both forks are held, eat for a fixed amount of time;
- Then, put the right fork down;
- Then, put the left fork down;
- Repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

Resource hierarchy solution

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks he plans to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks.

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.

What is MongoDB?

MongoDB is one of several database types to arise in the mid-2000s under the NoSQL banner. Instead of using tables and rows as in relational databases, MongoDB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in MongoDB. Collections contain sets of documents and function as the equivalent of relational database tables.

Like other NoSQL databases, MongoDB supports dynamic schema design, allowing the documents in a collection to have different fields and structures. The database uses a document storage and data interchange format called BSON, which provides a binary representation of JSON-like documents. Automatic sharding enables data in a collection to be distributed across multiple systems for horizontal scalability as data volumes increase.

Program Flow:-

- First of all create a server in mongoDB (Use terminal)
- Now Write main program add the mongo jar file.

- Now run the main code to see the output from other terminal using commands like "show databases;" and "db.collection_name.find();"

Reliability Testing:- Software reliability testing a testing technique that relates to testing a software's ability to function given environmental conditions consistently that helps uncover issues in the software design and functionality.

Parameters involved in Reliability Testing:
Dependent elements of reliability Testing:

- Probability of failure-free operation
- Length of time of failure-free operation
- The environment in which it is executed

Key Parameters that are measured as part of reliability are given below:

- MTTF: Mean Time To Failure
- MTTR: Mean Time To Repair
- MTBF: Mean Time Between Failures (= MTTF + MTTR)

Reliability refers to the consistency of a measure. A test is considered reliable if we get the same result repeatedly. Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability.
Reliability testing will tend to uncover earlier those failures that are most likely in actual operation, thus directing efforts at fixing the most important faults.
Reliability testing may be performed at several levels. Complex systems may be tested at component, circuit board, unit, assembly, subsystem and system levels.
Software reliability is a key part in software quality. The study of software reliability can be categorized into three parts:

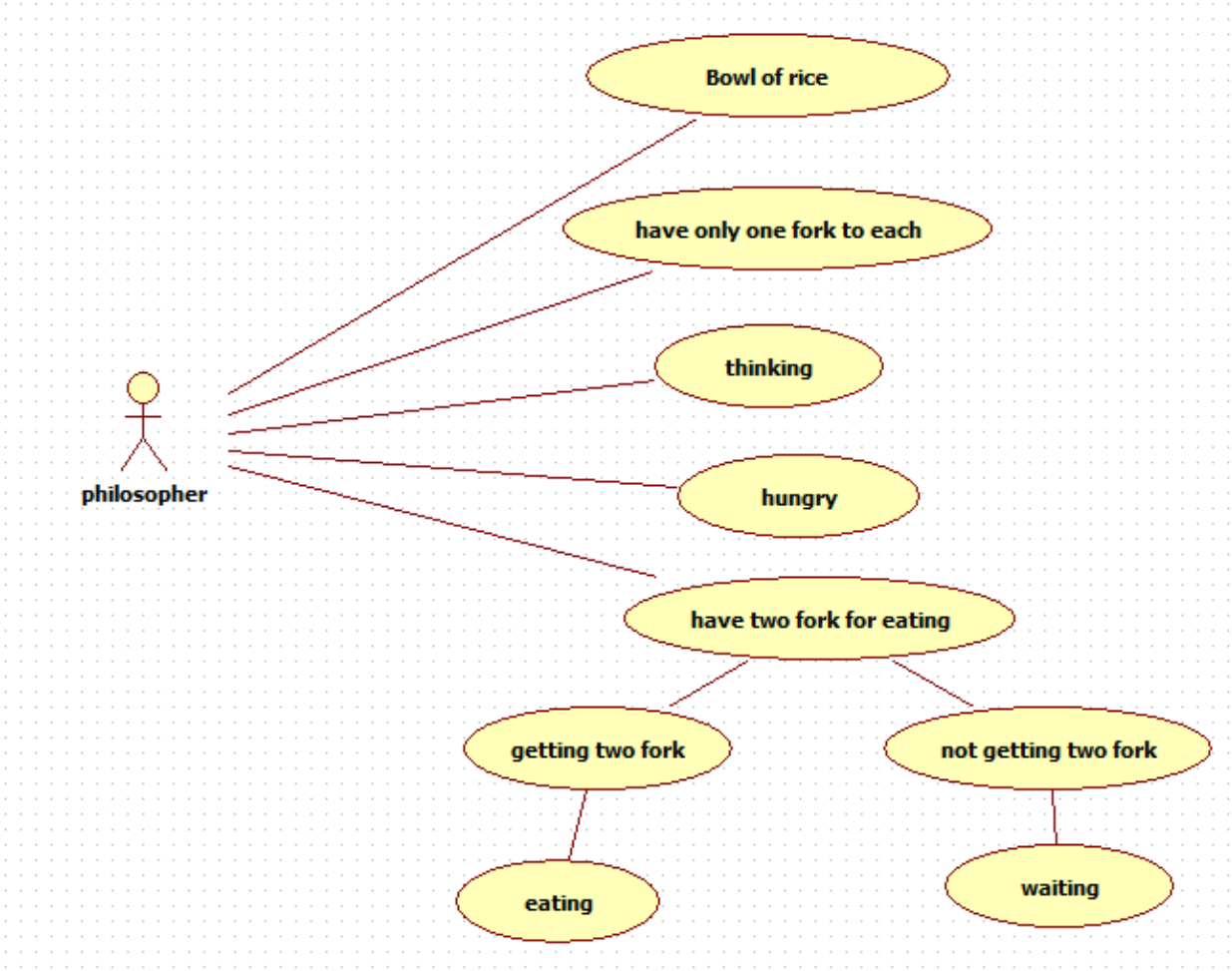
1. Modeling:-Software reliability modeling has matured to the point that meaningful results can be obtained by applying suitable models to the problem.

2. Measurement

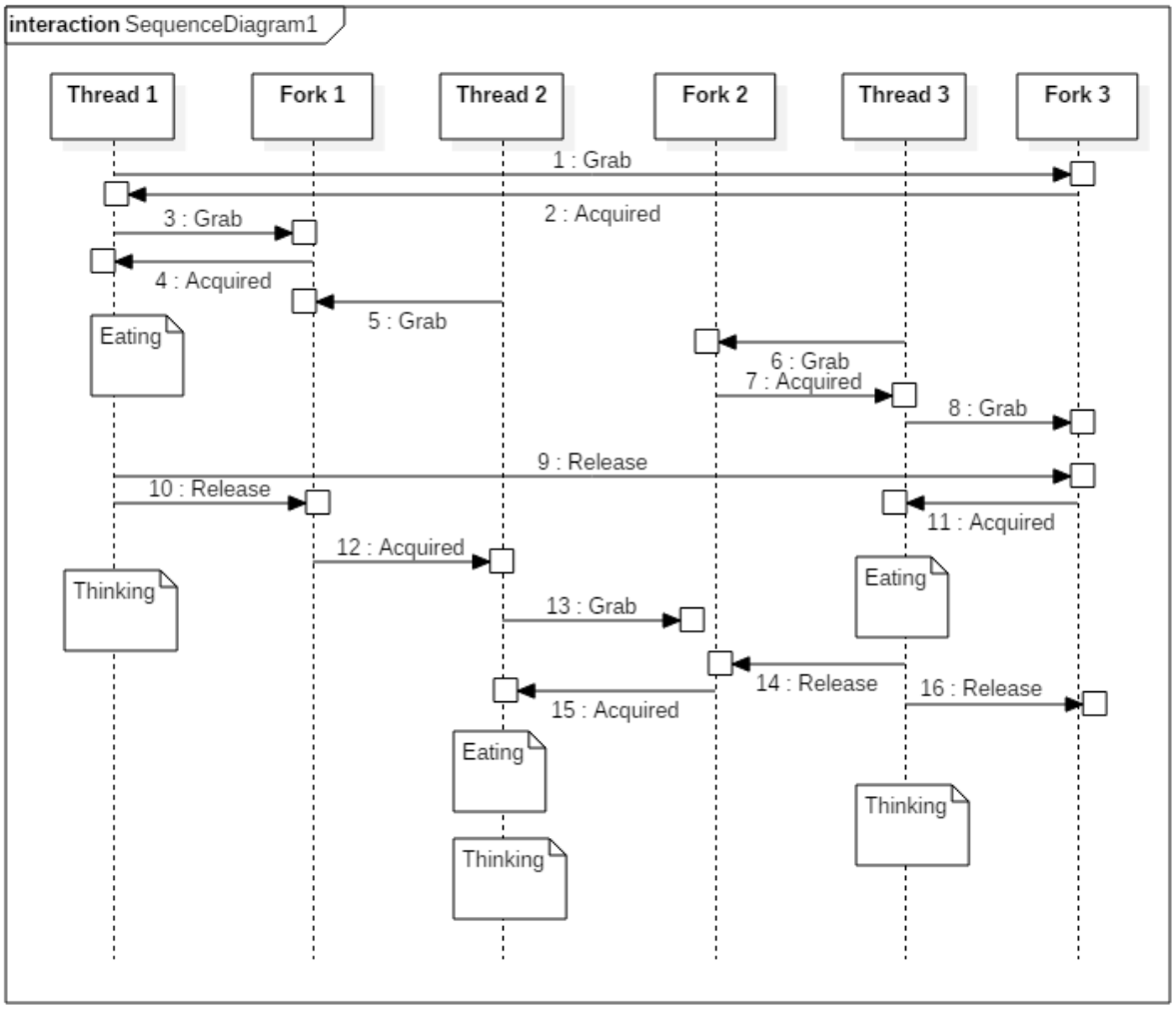
3. Improvement

UML:

Use Case Diagram:



Sequence Diagram:



Risks (Architectural):

- If synchronization will not be there then database will be in deadlock.
- If database is lost then there will be a problem.
- Sensor machine down.
- Sensors not active.

Design Patterns: FAÇADE pattern.

Mathematical Model

Let, S be the System Such that,

S= {I, F, O, success, failure}

Where,

Input:

I_1 = Initialization of database/server.

I_2 = Initialization of connection with the server.

I_3= Programme.

Functions:

F1 = main() : Here we create the threads and assign them to each philosopher and start their execution.

F2 = run() : This is a function to be written when we use runnable interface for parallel processing of threads.

F3= eat() : In this function, we lock the neighboring philosopher and the calling thread(philosopher) stars eating operation.

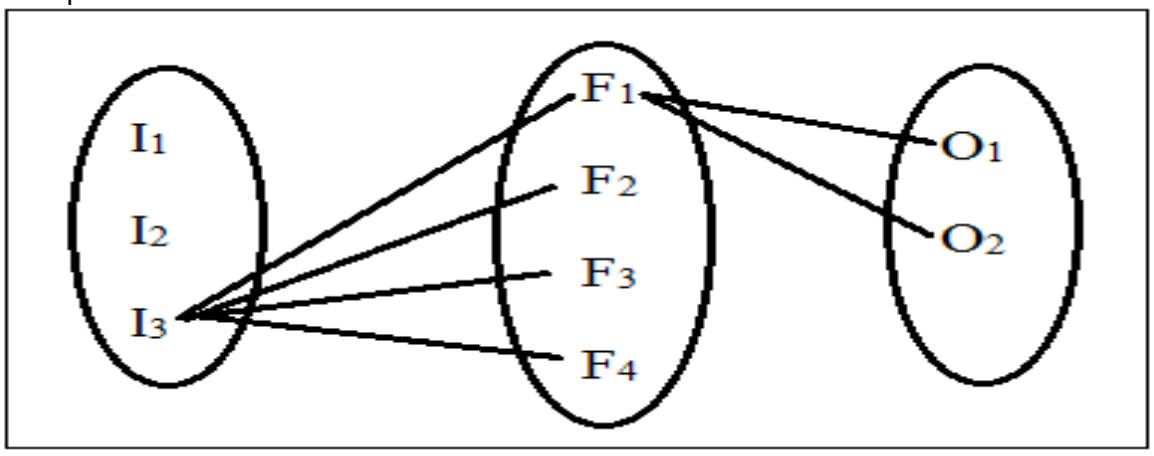
F4 = think() : In this function, the calling thread(philosopher) starts thinking.

Output:

O1 = Successful completion of the program and output written in database.

Success Case: It is the case the connection is Successful with the MongoDB server and threads start running parallel and gives output.

O2=Failure Case: It is the case when the connection is failed or there is an exception in threads.



Reliability Testing:-

Test-retest reliability is the most common measure of reliability. In order to measure the test-retest reliability, we have to give the same test to the same test respondents on two separate occasions. We can refer to the first time the test is given as T1 and the second time that the test is given as T2. The scores on the two occasions are then correlated. This correlation is known as the test-retest-reliability coefficient, or the coefficient of stability.

The closer each respondent's scores are on T1 and T2, the more reliable the test measure (and the higher the coefficient of stability will be). A coefficient of stability of 1 indicates that each respondent's scores are perfectly correlated. That is, each respondent score the exact same thing on T1 as they did on T2. A coefficient correlation of 0 indicates that the respondents' scores at T1 were completely unrelated to their scores at T2; therefore the test is not reliable.
So, how do we interpret the coefficients of stability that are between 1 and 0? The following guidelines can be used:

- 0.9 and greater: excellent reliability
- Between 0.9 and 0.8: good reliability
- Between 0.8 and 0.7: acceptable reliability
- Between 0.7 and 0.6: questionable reliability
- Between 0.6 and 0.5: poor reliability
- Less than 0.5: unacceptable reliability

Test ID	Test case	Expected Output	Actual Output	Test Case
1	Check the mutual exclusion in case the number of sensors are 10	No deadlock. All sensors write to the DB.	No deadlock. All sensors write to the DB.	Pass
2	Check the mutual exclusion in case the number of sensors are 20	No deadlock. All sensors write to the DB.	No deadlock. All sensors write to the DB.	Pass
3	Execute TC 1 again	No deadlock. All sensors write to the DB.	No deadlock. All sensors write to the DB.	Pass