Class Test

Mirrored databases

Introduction:

In this emerging world of e-commerce, tremendous amount of data are collected, processed and exchanged everyday. Consequently, the e-commerce based companies maintain their data in a distributed fashion with redundancy to make their system fail-safe.

Description:

ShopKraze, an e-commerce website. The database is maintained (Mirrored) at two locations, Kharagpur and Mumbai. Both the servers should have exactly same data at all times. To ensure this, the company chooses the following design:

The servers:

A server is installed corresponding to each database for performing various operations over the dataset. The server installed at Kharagpur is designated as ***primary*** server as the a client interfaces with this server only. The Bombay server is designated as ***backup*** server; it executes various commands sent by the Kharagpur server.

A client sends requests to the *primary* (Kharagpur) server. If a request involves a write operation (buying a product) on the database, then the *primary* server sends a command to the *backup* (Mumbai) server stating the buy request; the procedure is explained subsequently.

Otherwise, if a request involves only a read operation (browse products list), then the primary server responds to the request by reading its database (no message is sent to the backup server).

Available functionalities (requests by a client):

Browsing the available products:

Product ID	Product Name	Available Stock
P1	Jacket	21
P2	Boots	123
P3	T-shirt	0

Table1: The stock list on both servers

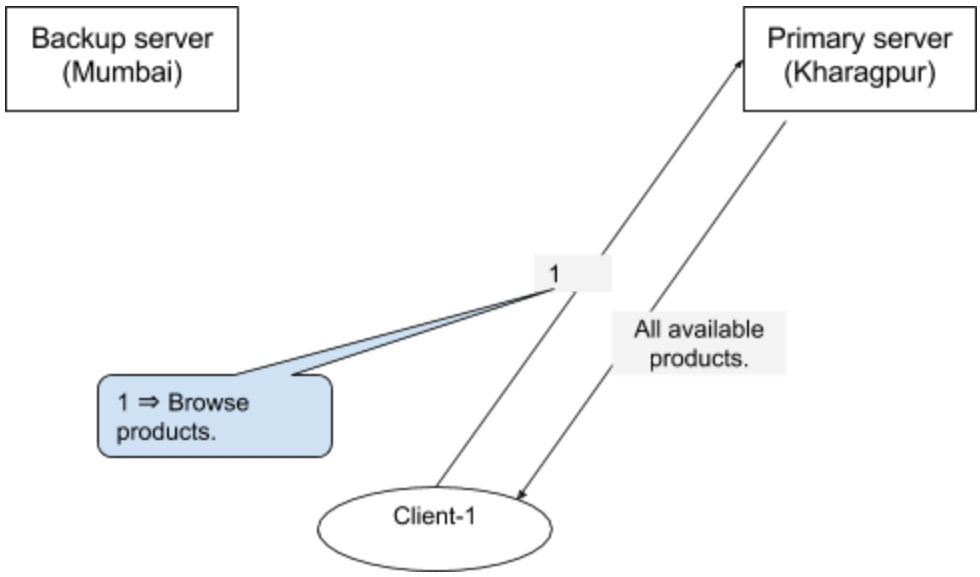

Fig.1: Browsing products and past orders

A client sends a *browse* request to the *primary* server. The server responds to the request based on the availability in the stock list (see *Table 1*). The procedure is illustrated in the *Fig. 1*.

Buying a product:

A buy request contains the ***ID*** of the product that the client wants to buy. Figure 2 illustrates a *buy* request.

An Example of a buy request first:

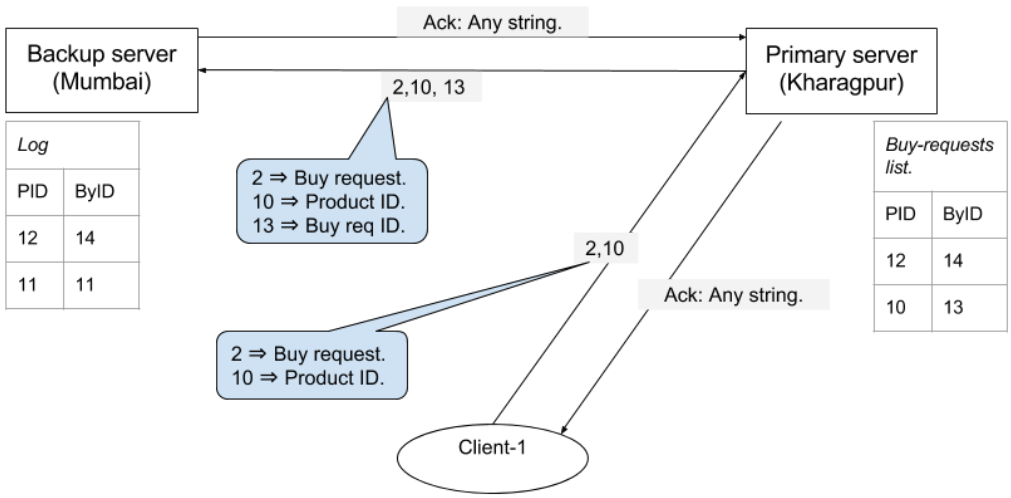

Figure 2: An example of a purchase.

A primary server maintains a variable called *sync* and a list called *buy-requests*. The *sync* variable is a boolean variable and have a value 1 if the two databases, at the primary and the backup server, are consistent and 0 otherwise. The variable is initialized to one because initially both the databases are synchronized.

The *buy-requests* list contains those requests which have been forwarded to the backup server but the backup server has not acknowledged them. It is initialized as an empty list.

Upon receiving a buy request the primary server performs the following actions, only if the product is in stock at the primary server. Otherwise, if the product is out of stock, the request is responded with an error message sent to the client. The whole control flow is illustrated in *Figure 3.*

1. If the *sync* flag is set to one i.e. the databases are consistent, then the primary server adds the request to the *buy-requests* list along with a unique ID identifying the *buy* request, forwards the *request* to the *backup* server and waits for an acknowledgment from the backup server.
   The connection between the primary server and the backup server is assumed to be **synchronous.** Hence, the primary server waits for ***t*** time units for the acknowledgment.

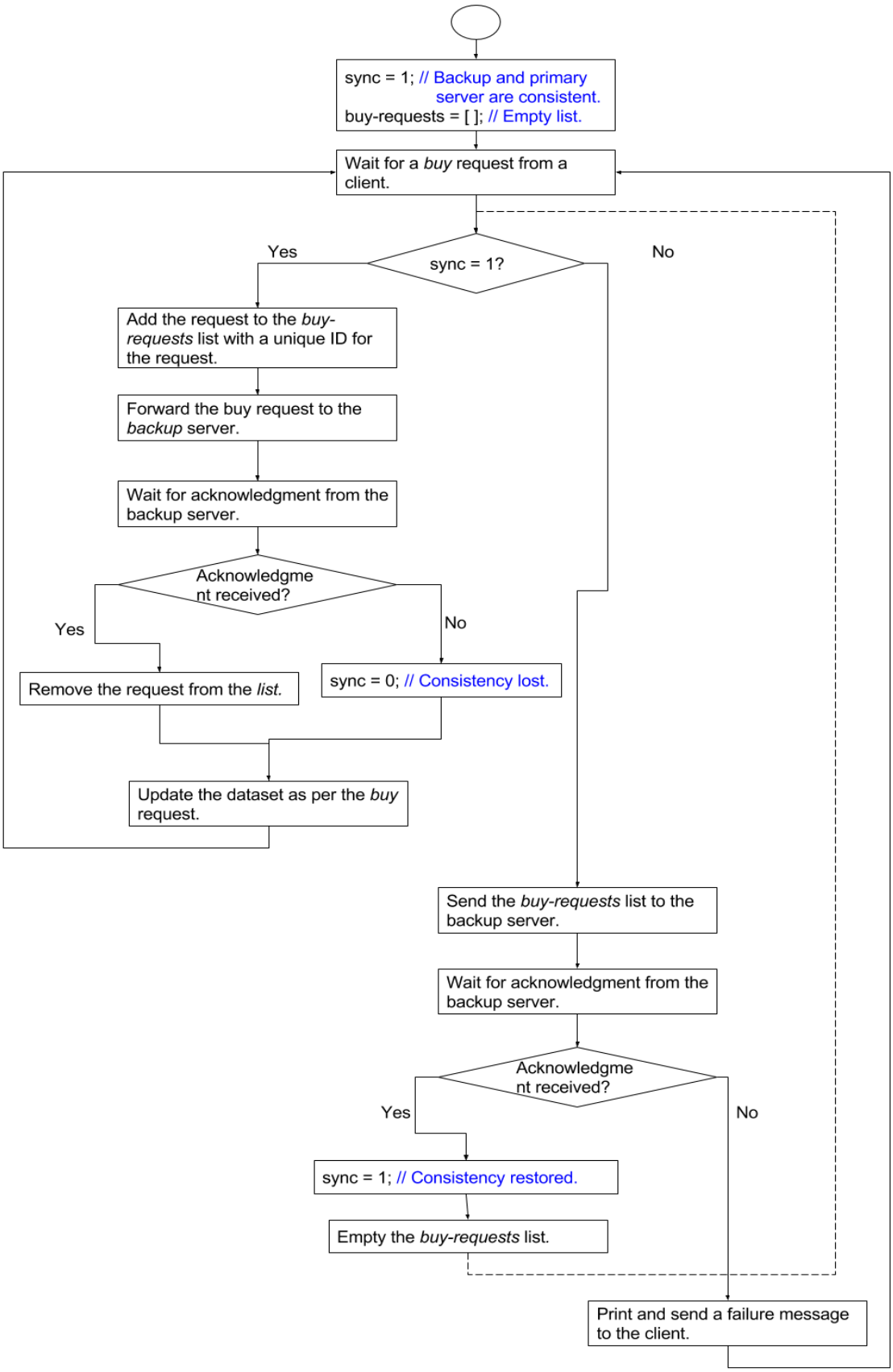

Figure 3: Control flow for handling buy request by a primary server.

2. If an acknowledgment arrives within ***t*** time units, then the primary server removes the request from the *buy-requests* list. Otherwise, it sets the *sync* flag to 0.

3. After removing the *buy* request from the *buy-requests* list (if ack is received) **or** updating the *sync* flag (if ack is not received), the primary server makes the necessary updates in its database i.e. reducing the number of items in stock for the purchased product id and sends an acknowledgment to the client.

4. Otherwise (else of the point 1), if the *sync* flag is set to zero, then the two databases are not synchronized. In this case the primary server sends the *buy-requests* list to the backup server and waits for an acknowledgment.

5. If an acknowledgment is received, then the primary server sets the *sync* flag to one, clears the *buy-requests* list and treats the *buy* request as a fresh request.

6. Otherwise, if an acknowledgment is not received in ***t*** time units, then the primary server sends a *failure* message to the client and ignores the request.

The Backup Server

The backup server maintains a log in which it stores all the requests it has updated in its database.

Upon receiving a buy message from the primary server, the backup server performs following steps.

1. It reduces the the number of items of the product by one.

2. Sends an acknowledgement to the primary server.

Upon receiving the *buy-requests* list:

1. For each request in the received list, the backup server checks its log.

2. If it has already worked on the request, then it ignores the request.

3. Otherwise, it updates its database accordingly and adds the request to its log.

The backup server has following three probabilities[1]:

1. Any received message is accepted with probability p_1, or in other words, a received message is discarded with probability $(1-p_1)$.

2. Any received message is processed (update database) with probability p_2.

3. After updating the database an acknowledgment is sent with probability p_3.

Use a macro for the value of the probabilities. Initially use any value; we will specify different values while evaluating.

[1] Implement it only after writing the whole test.

If a message is discarded or not processed, then an acknowledgement will NOT be sent to the *primary* server.

Tasks:

1. Implement the *primary* and the *backup* server along with client and simulate the system with both servers and one client at a time.
2. At each request towards primary server, print the client's IP address on server's terminal.

Notes:

1. Logs and buy-requests list can be arrays, no need to store them in files.
2. Use random function for probabilities.
3. Use TCP between the two servers.
4. Use UDP between the client and the primary server.
5. For the purpose of the test it is assumed that the two servers never crash.