Solving The Concurrency Issue Generated By Users Attempting To Access A File In Dropbox

Posted on May 12, 2020

Abstract – This document is about the concurrency issue that arises when users try to access the same file for different operations, i.e., reading and writing the same file. The solution to this concurrency will be implemented using Erlang by using the mutual exclusion technique. We will try to resolve or minimize the concurrency issue through the use of a semaphore.

Keywords – *Erlang, semaphore, mutual exclusion (Mutex), concurrency*

Introduction

In this document, we will be trying to solve the concurrency issue generated by users attempting to access a file in Dropbox. The two users are trying to read and write at the same time. We will implement the solution using Erlang programming language through which users will be able to access the same file without any concurrency issue arising. Erlang is a useful programming language that has been planned to the point of improving concurrent programming (Huang, 2025).

Sequential programs, when rendered, have a single stream of operations, unlike concurrent programs. The concept of concurrent programs is the opposite. There are numerous streams of operations executing together, and each stream of operation behaves like a sequential program. This nature of concurrent programs is very oppressive as multiple streams of actions communicate with each other as well as interfere with one another (van den Heuvel & Pérez, 2024).

To analyze the concurrency issue that arises when users are trying to access the same file in a Dropbox for different operations, we will try to access a file for a reading operation and another file for a write operation simultaneously. After this, we will be finding possible solutions as to implement mutual exclusion. In the solution, we will handle the situation by prioritizing one of the processes so that the highly prioritized project completes its execution before the other one starts to execute. This approach can be achieved through semaphores. To achieve the objective, we have to make sure that the objects have mutex access. A mutex is a synchronization object provided by the Erlang programming language, “mutexes are ordinarily used to shield a mutual asset from concurrent access. An errand seizes (or procures) the mutex, at that point gets to the asset, and after that discharges the mutex.” (Microsoft, 2026).

In this paper, the background of the Dropbox service and Erlang programming language will be

provided, along with the concurrency problem itself. After describing the background, will we address some approaches to solve the given problem? We will also explain the pros and cons of every approach. Then, the decided approach will be implemented, and a description of every step will be provided. Then, we will move towards a conclusion by summarizing the problem and the solution discussed in this paper.

Background

A. What is Dropbox Service?

Dropbox cloud storage is a standout amongst the best and most helpful instruments accessible online today and is, from numerous points of view, the initial phase of cloud computing. Genuine cloud computing suggests that a PC comprises just a mouse, console, screen, and web association. All data that would ordinarily be hung on a PC's hard drive is kept inside the web. Cloud computing includes no product or information on the real PC. Everything is performed through the web program on a system of PCs that are "in the cloud." This large-scale level of cloud computing is a moderately new idea. Be that as it may, little-scale "small-scale" cloud computing has existed for quite a long time. Dropbox cloud storage is a case of miniaturized scale cloud computing.

Clients can store information in an online framework and get that information from any area on the web. Dropbox enables clients to accumulate to 2 gigs of any information, for example, records or music documents, for nothing in an online storage framework. Clients can access their accounts and retrieve documents through any device provided with an internet connection. Dropbox storage is like an external hard drive that can be accessed through any gadget, such as a smartphone, tablet, laptop, or PC.

B. What is Erlang Programming language?

Erlang is a useful programming language which likewise has a runtime domain. It was worded such that it incorporated help for simultaneousness, dissemination, and adaptation to non-critical failure. Erlang was initially created to be utilized as a part of a few expansive media transmission frameworks from Ericsson.

The principal rendition of Erlang was produced by Joe Armstrong, Robert Virding and Mike Williams in 1986. It was initially a restrictive language inside Ericsson. It was later discharged as an open-source language in the year 1998. Erlang, alongside OTP, an accumulation of middleware and libraries in Erlang, is presently upheld and kept up by the OTP item unit at Ericsson and alluded to as Erlang/OTP.

C. *Concurrency Problem*

To describe the concurrency problem, we will analyze two scenarios: bank and DropBox.

- *Bank:*

In this example, the amount is deposited into a bank account by using the Erlang programming language.

depositA(Amount) ->

old_balance = get_balance(),

new_balance = old_balance + Amount,

set_balance(new_balance).

The argument Amount is passed in a method depositA(), the argument is specified by the user. After this, the remaining balance in the account will be assigned to a variable old_balance. The new balance will be equal to the old balance and the amount specified by the user. The calculated value will be stored in the new_balance variable. Now, the amount will be deposited in a bank using the set_balance()method, which will take the new_balance variable and increment the balance of the account. Now, let's see how concurrency issues arise in this simple process. Let's take an example of User_A through a labelled transition system (LTS).

set_balance().A

get_balance().A

get_bal.A

set_bal.A

2

1

0

Similarly, User_B will complete its process in the same way as a User_A.

1

get_balance().B

set_balance().B

0

2

Now let's assume that the current balance of the bank account is 0 and User_A is rendering the method depositA(20), and at the same time, the User_B is busy executing the function named depositA(40). In the perfect scenario, the process will be like the following:

Balance ^{0} -> get_balance.A ^{0} ->set_balance.A ^{20} -> get_balance.B ^{20} -> set_balance.B ^{40}

In the above scenario, the new account balance will be 60. The balance of 10 was deposited by User_A, followed by User_B, who deposited a balance of 40 in the account, calculating the accurate balance in the end.

Balance ^{0} -> get_balance.A ^{0} ->set_balance.B ^{10} -> get_balance.B ^{30} -> set_balance.A ^{10}

The balance will be just 10 in the above scenario. After the amount of 40 is deposited in the account because of the concurrency issues, the account balance has a shortfall of 30, and this is not accepted in any condition.

- *Accessing files residing in a DropBox:*

Let's analyze an example of accessing a file in Dropbox for reading operations using the Erlang programming language.

read_file(filename) ->

{ok, File}=file:opem(filename,[read]),

txt= file:read(File,1024*1024).

The function read_file() will take a filename as an argument specified by the users. The process then uses the filename and tries to open the file with the same name. If the name exists, then the relevant file with the read mode will be opened for reading purposes.

The user can also access files for writing purposes, which is implemented by using Erlang as follows:

Write_file(filename,data)->

{ok, File}=file:opem(filename,[read]),

file:write(File,data).

The function write_file() will take the filename and data as an argument. The data is the content that

a user wants to write in a file specified by the user itself.

Now let's analyze the concurrency issue that will complicate things in the simple scenario. The scenario can be described by the Finite state machine:

Read/write

Free

In the above FSM, it is obvious that the file or resource will be treated as free, and any user can read or write on the file simultaneously, which ultimately generates the concurrency issue (Le et al., 2022). The following FSM will show the system without any concurrency issues.

Read

write

Free Busy

Signal

In the concurrence-free scenario, we will prioritize the write operation. When the user writes something in a file, the resource will be busy and unavailable until it is set free. The user can read or write in a free state. Now, let's analyze the approaches to solving the concurrency problem in Dropbox.

Approaches

We will analyze the pros and cons of the following approaches:

A. *Mutex And Semaphores*

As expressed before a mutex is a significant choice to deal with simultaneousness issues, this execution includes the property that one process should finish first the other process will begin its execution. A semaphore executes such behaviour by structuring and assembling the processes that will use the same resource.

An issue that can emerge is a state called a deadlock, which, if not focused on in the first code, can cause issues for the framework. "A deadlock is an unintended condition under which various activities are stuck on some synchronization crude sitting tight for each other to continue."

B. *Asynchronous message passing*

Message passing, if done effectively, can make every one of the exchanges engaged with a saving money framework nuclear, implying that an exchange must act totally or not in any way, giving no halfway outcomes.

The primary issue with this situation originates from a crash on the framework, either locally or through not including parameters when the framework gets a blunder.

Implementation

We will try to implement both approaches: The message-passing technique on the bank example and the mutex and semaphore approach to accessing the file in a drop box.

To apply the message-passing technique, we need to alter the read-modify-write process of the old deposit method. The older deposit function is modified by adding more parameters. For instance, the user cannot withdraw the amount when the balance is negative, or the account method should do the calculations and then provide the deposit method with an OK signal to finish the process.

depositB (Amount) ->

account_process ! {deposit, Amount, self()},

receive

{deposit, Amount, new_Balance} ->

{ok, new_Balance}

end.

It is obvious that the deposit method has been modified significantly. In the above function code, the process identifier (PID) has been assigned to account_process. It is also important to know the meaning of the symbol '!', which shows the send followed by the message. The account method provides the new balance to the account, signalling the OK and exhibiting that the account balance has been refreshed. This technique won't allow processes to interfere with each other because both setter and getter methods are defined inside the accounting method.

Moving onto the accounting procedure itself, you can see that with the utilization of message passing, the greater part of the calculations are done inside the capacity itself, guaranteeing that the procedure is unclear, enabling us to totally discount any probability of further simultaneousness issues as the one expressed.

```
account(Balance) ->

receive

{ set, new_Balance } ->

account( new_Balance );

{ get, From } ->

From ! { balance, Balance },

account( Balance );

{ deposit, Amount, From } ->

New_Balance = Balance + Amount,

From ! {deposit, Amount, new_Balance},

account( new_Balance )

stop -> ok

end.
```

As should be obvious, when { deposit, Amount, From } is gotten by the account() process, it will finish the calculations, refresh the account and send back the refreshed adjust to being gotten and giving it the all obvious from the inside the deposit method.

```
————————————————————- fileAccessing.erl ————————————————————

-module(fileAccessing).

-export([fileAccessed/1, start/0, stop/0, read_file/0, write_file/1]).

fileAccessed( FileName ) ->

receive

{write, From, Data} ->

ok = file:write_file(FileName, Data),

From ! ok,

file(FileName);
```

{read, From} ->

{ok, Data} = file:read_file(FileName),

From ! Data,

file(FileName);

stop -> ok

end.

start() ->

File_PID = spawn(file, file, [“somefile.txt”]),

registe0r(file_process, File_PID).

stop() ->

file_process ! stop,

unregister(file_process).

read_file (filename) ->

{ok, File}=file:opem(filename,[read]),

txt= file:read(File,1024*1024).

Write_file (filename,data) ->

{ok, File}=file:opem(filename,[read]),

file:write(File,data).

————————————————————- mutextest.erl————————————————————–

-module(mutextest).

-export([start/0, user/1]).

start() ->

fileAccessing:start(),

mutex:start(),

```
register(tester_process, self()),

loop(10, 20, 100),

unregister(tester_process),

mutex:stop(),

fileAccessing:stop().

loop(_, _, 0) ->

true;

fileAccessing:read_file( filename ),

spawn( fileAccessingtest, user, [filename] ),

spawn( fileAccessingtest, user, [filename] ),

receive

done -> true

end,

receive

done -> true

end.

user( filename ) ->

fileAccessing:write_file( filename ),

tester_process! done.
```

Conclusion

We have analyzed the concurrency issue that occurred in Dropbox. The issue was when one or more users attempted to access the same file; that is, one user tried to open a file in reading mode, and another one opened a file for writing purposes, which resulted in concurrency. We have analyzed the possible solutions required to solve these issues. Mutex and semaphore and applying asynchronous messages for atomic transactions are some possible solutions for this problem. We adopted the

mutex technique to solve the concurrency issue in Dropbox. We have implemented the mutex solution in the Erlang programming language. It is a fundamental programming language suitable for developing applications or systems which may require concurrent operations.

References

Huang, I. (2025, May 20). *Erlang/OTP 28 highlights*. Erlang/OTP. https://www.erlang.org/blog/highlights-otp-28/

Le, P., Lathia, T., & Baid, M. (2022, November 3). *Future-proofing our metadata stack with Panda, a scalable key-value store*. Dropbox Tech. https://dropbox.tech/infrastructure/panda-metadata-stack-petabyte-scale-transactional-key-value-store

Microsoft. (2026, July 17). *Using mutex objects*. Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/sync/using-mutex-objects

van den Heuvel, B., & Pérez, J. A. (2024). A gentle overview of asynchronous session-based concurrency: Deadlock freedom by typing. *Electronic Proceedings in Theoretical Computer Science, 414*, 1–20. https://doi.org/10.4204/EPTCS.414.1