Investigating Concurrency Issues in Dropbox

Ionita Daniel
School of Science and Technology
Middlesex University
London, United Kingdom
di118@live.mdx.ac.uk

Student number- M00592415

***Abstract*— Cloud storage platforms are widely used in many environments from individuals to businesses, increasing flexibility of accessing files from anywhere. Concurrency allows for this all to happen, however it also comes with issues. Interference appears when two processes try to manipulate a critical section in the shared resource. In this paper, you can find a multitude of possible solutions that attempt to resolve the aforementioned problem of interference. We also enter into a detailed discussion about the possible limitations of our solutions.**

Keywords—concurrency; Dropbox; cloud storage; file sharing; Erlang; interference; mutual exclusion; MutEx; counting semaphore; deadlock.

I. INTRODUCTION

Dropbox is the most preferred modern-day cloud storage solution among both technical and non-technical individuals. Dropbox along with many other cloud platforms now support file sharing, allowing two users to gain access to a single file at the same time. This paper highlights a problem faced in similar concurrent scenarios and explores some possible solutions.

Part II of this paper, presents the background of Erlang, concurrency and an in-depth look into Dropbox. Following on, part III introduces a detailed overview of the concurrency issue. Some possible solutions are discussed and evaluated in parts IV and V. The findings are then concluded and presented in part VI.

II. BACKGROUD

A. Erlang

Erlang was created by Joe Armstrong in 1986 as part of a thesis project for Royal Institute of Technology, Sweden [1]. Its core functionality was designed to satisfy the needs of Ericsson, a telecommunication company and Joe's employer, in managing telephone equipment.[5]. Due to this, Erlang has built-in support for concurrency, distribution and fault tolerance. [1]

B. Concurrency

Concurrent programs have multiple threads of execution making them able to perform parallel computations and various control activities which arise at the same time. Compared to sequential programming usually concurrent programs are used to boost performance and responsiveness [7].

C. Dropbox

Dropbox is a file hosting service that allows users to share and store files and folders. The service allows you to access files in the shared folder from anywhere via an internet connection. It was founded in 2007 by Drew Huston and Arash Ferdowsi [6]. Currently, the number of registered users is above 500 million in over 180 countries. It offers a free service which allows users to store up to 2GB of data although additional storage can be purchased. The service is available online via a web-browser or as a native software application which can run on a variety of operating systems and devices[3].

III. DESCRIBING THE PROBLEM

Interference is a problem that can occur within the realm of concurrency. The problem manifests its self when two processes attempt to manipulate a shared resource known as the critical section. When interference occurs, it results in data loss or skipped operations as one process will be working on an out of date version of the resource that may have been manipulated by another process in the meantime.

It is common to find interference within file sharing applications such as Dropbox. This is due to the fact that multiple users are able to alter the same file which in this case is acting as the shared resource.

Dropbox has been modelled in the Erlang environment using a function called *storage(Content)* to simulate a file. *The* model features the two file operations *read()* and *write(NewContent),* used to interact with the model file. It also contains a function called *concatenate(StringToAdd)* which retrieves the contents of the file and concatenates this with *StringToAdd* and writes the resulting string to the file. See below

```
concatenate(StringToAdd) ->
  OldContent = read(),
  NewContent = OldContent ++ StringToAdd,
  write(NewContent).
```

This function does not face any issue if the processes are reading and writing to the file independently, yet if the read-write operations, executed by each process are interleaved with each other the results are not as expected. Let's assume we pass to the function the strings *"Process 1"* and *"Process 2 "*. The result is *"Process 2"* when the expected result should be

"Process 1 Process 2". As mentioned above this unexpected output is also known as interference.

Figure 1 presents the scenario using Finite State Machine (FSM), the transitions in blue are the one causing interference.

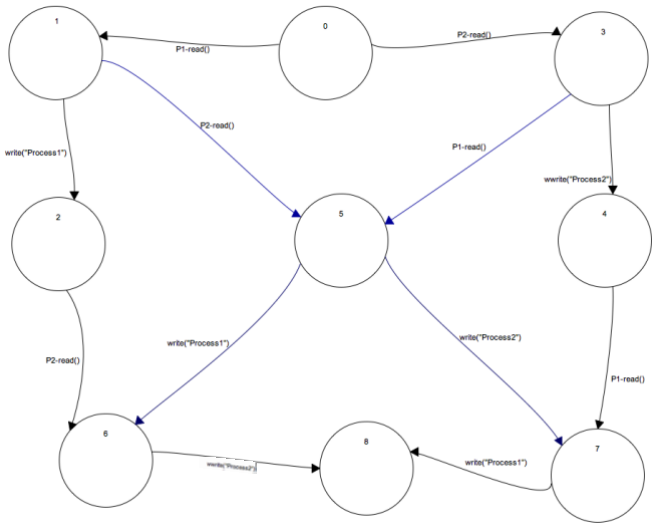


Figure 1. Finite State Machine modelling the actions read-write to a single file.

IV. POSSIBLE APPROACHES

A. Mutual Exclusion

In 1965 E.W. Dijkstra states in one of his papers that interference "has been an open question since at least 1962" [8]. A possible solution to interference is mutual exclusion, also known as MutEx. This can be realized by introducing atomic action, which behaves as locking mechanism [4]. Applied to the example mentioned above in Part III, it works as follows: content is locked from being accessed when one of the users works on it, the state of the file is changed to busy, then later being released after and only after the user finishes working. If the other user attempts to change the file simultaneously is being automatically placed in a queue waiting for the other process to send the signal (not busy).

It can be implemented in our function like this:

```
concatenate(StringToAdd) ->
    mutex:wait(),
    OldContent = read(),
    NewContent = OldContent ++ StringToAdd,
    write(NewContent),
    mutex:signal().
```

The result then will be as expected : " Process 1 Process 2".

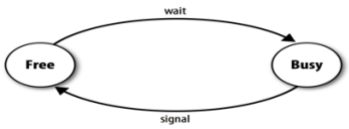


Figure 2. Mutual exclusion state diagram [2].

B. Other solution

Another possible solution to our issue is the *counting semaphore*. The way it works is very similar to the Mutex semaphore, the only difference is that the latest allows you to lock more than one file. A possible scenario could be a public toilet with 4 cubicles. Four clients enter and lock the door after them, with each entrance the counting semaphore increases its value by one (*up*), if there is any other client trying to access, they will have to wait until the toilet is vacated, with each vacated cubical the counter decreases its value by one (down). This way every time the counter is at 4 we know that no other client can enter. Also -1 represents the error state as we can see below in figure 3.

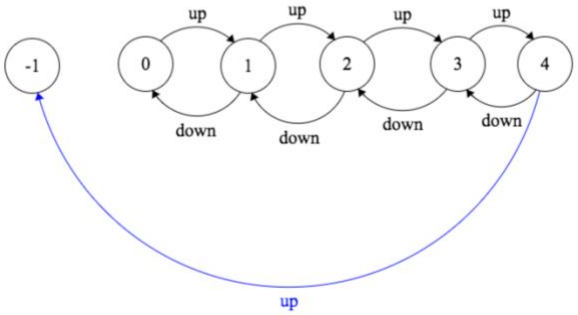


Figure 3. Finite State Machine modelling the counting semaphore.

This is another possible solution, however it is not as suitable as Mutex for our particular issue because we only need the problem to be solved for only two users trying to alter the same file.

V. DISCUSSION

Although MutEx is good in certain situations, if not used appropriately, it can lead to an infinite loop of requests also known as deadlock. The way it manifests is by the obligation of a client to wait access from a second client which again waits and so on. A graphical representation can be seen below in which C1 denotes Client 1.

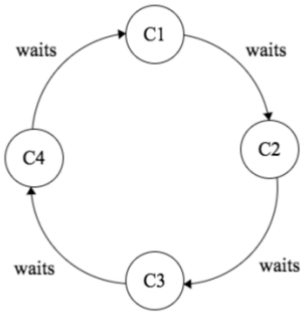


Figure 4. Finite State Machine illustrating deadlock.

A better scenario in which the problem can be identified is the story of "*the five dining philosophers*". The story discerns five philosophers situated around a dining table each with a plate of spaghetti in front of them as you can see in the diagram illustrated below where philosophers are marked 0 to 4.

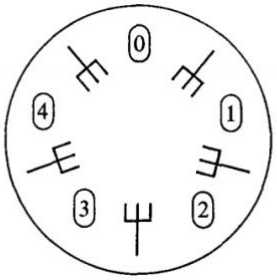


Figure 5. Diagram illustrating the tale of "*the five dinning philosophers*" [9].

We assume each philosopher requires two forks to eat. However they only have access to five forks placed in-between them. We also assume that each philosopher can only either eat or think. The problem arises when three or more philosophers are willing to eat simultaneously. Due to the limited number of forks available and the fact that if they want to simultaneously eat, this scenario transforms into deadlock where each of them waiting for the fork from the other. A possible solution to this problem is to establish a set of rules which prohibits the philosopher in an even position on the table to pick the fork on his right-hand side. On the other side, the philosophers at an odd position on the table should only pick the fork on their right hand.

Respecting the rules established two philosophers will always be able to eat, hence making the deadlock scenario impossible.

VI. Conclusion

This journal paper is revolved around interference, a problem which occurs in concurrent programs due to actions interleaving. We also analyzed the system damages caused by this problem in a cloud storing platform such as Dropbox.

The possible solutions were analyzed in Section IV and we concluded by implementing a series of tests which made us choose MutEx as the most proficient solution. Although Section V explains the negative effects which can arise when using MutEx, sometimes we can end up in a state of deadlock. In the second part of the discussion section, deadlock is reinforced by the tale of "*the five dining philosophers*". To conclude we strongly believe that MutEx remains the best possible solution for our Dropbox scenario.

VII. References

[1]J. Armstrong, Making reliable distributed systems in the presence of software errors. Stockholm, 2003.

[2]"Erlang -- Concurrent Programming", Erlang.org, 2018. [Online]. Available: http://erlang.org/doc/getting_started/conc_prog.html. [Accessed: 16- Feb- 2018].

[3]D. Quick and K. Choo, "Dropbox analysis: Data remnants on user machines", Digital Investigation, vol. 10, no. 1, pp. 3-18, 2013.

[4]F. Kammüller, "Distributed Computing and Networking Concurrency 4 - Recap and Mutual Exclusion", Middlesex University, 2018.

[5]"Inside Erlang – creator Joe Armstrong tells his story", Ericsson.com, 2018. [Online]. Available: https://www.ericsson.com/en/news/2014/12/inside-erlang--creator-joe-armstrong-tells-his-story. [Accessed: 16- Feb- 2018].

[6]"Dropbox | Company Info", Dropbox.com, 2018. [Online]. Available: https://www.dropbox.com/news/company-info. [Accessed: 16- Feb- 2018].

[7]J. Magee and J. Kramer, Concurrency. Chichester, England: J. Wiley & Sons, 2006.

[8]E. Dijkstra, "Solution of a problem in concurrent programming control", Communications of the ACM, vol. 8, no. 9, p. 569, 1965.

[9]E. Dijkstra, "Hierarchical ordering of sequential processes", Acta Informatica, vol. 1, no. 2, pp. 115-138, 1971.