Investigation of Concurrency Issues in Dropbox

Christiaan Burrett-Bijlstra
Middlesex University
London, United Kingdom
School of Science and Technology
CB1291@live.mdx.ac.uk

***Abstract*— Due to Moore's law still holding the ever increasing number of transistors within integrated circuits, we have seen changes in the way systems are deployed. System engineers and programmers are able to run a greater number of modules concurrently. We have also seen an increase in companies offering cloud storage services not just to the relatively small market of businesses but also to the general consumer. This fact has brought companies like Dropbox into fruition and has allowed them to be able to offer their service to over 500 million customers. However one of the fundamental problems found when allowing users and systems to execute processes simultaneously is something we call "Interference". This paper is aimed at identifying this problem and discussing the possible solutions available to system engineers.**

Keywords—Concurrency; Interference; Deadlock; Erlang; MutEx; Dropbox; Semaphore; interleaving;

I. Introduction

With the exponential rise of technology and the relatively recent influx of cloud computing, companies face challenges to develop cloud computing services that can be accessed by multiple users at the same time, whilst still maintaining an uninterrupted user experience. This is a demanding feature to implement. The most common complication that arises is one of concurrency. The problem comes to light when two users are simultaneously uploading the same files, what we observe happening to systems that have not implemented any features to handle concurrency is a conflict within the system. To resolve this conflict the system automatically favours the files of the most recent user as the system interprets that information to be the most up-to-date and thus most relevant. Due to this inherited characteristic information can be overridden. However companies such as Dropbox have managed to overcome these tremendously complicated issues and scale there software to around 500 million users. In this journal I will be discussing this problem of interference and concurrency in more detail and identify the resolution that cloud storage services such as Dropbox have implemented. In the next section of this paper (section II) we will briefly describe the background of both Erlang and Dropbox and we shall identify the terms 'concurrency' and 'semaphore'. The next part of this paper will be focussed on identifying and explaining the problem faced by concurrency (section III). After identifying the problem we will try to resolve the problem using researched solutions (section IV). After resolving the problem we will go into the effectiveness of the resolution and identify any negative effects it might have on the system (section V). The last section of this paper will conclude the findings (section VI).

II. Background

A. Dropbox

Dropbox, was founded in 2007 by Drew Houston and Arash Ferdowsi [1], as a cloud storage service which today has over 500 million users and 300,000 businesses worldwide [2]. It is owned by Dropbox, Inc. who are headquartered in San Francisco, California, USA. Dropbox is currently the largest consumer cloud storage company holding around 77% of the market share within cloud storage in 2017 [3]. They aim to create a remote file storage service that is both aimed at ease-of-use and the privacy of data stored. Dropbox installs software onto the users device, which once installed creates a local Dropbox file which is mirrored from the users Dropbox cloud file. When there is an active internet connection Dropbox automatically synchronizes with the cloud file. However when the user no longer has an active internet connection the file is still functional and will mirror itself with the online file once an internet connection is reestablished. Dropbox integrates this functionality across multiple device platforms such as IOS, Android, Mac OS, Microsoft Windows and Windows Phone.

B. Erlang

Erlang is a functional language developed by Joe Armstrong, Robert Virding, and Mike Williams for Ericson [5]. It started development in 1981 due to a demand to develop a reliable computing language to program telecom

applications. However it was soon realised that a language which was efficient at handling telecommunications switching systems could also be applied to a wide variety of other industrial control systems. Due to the nature of Erlang it can scale large programs down to smaller more easily managed modules. One of the advantages of doing this is that it allows a program to be updated without having to stop the whole program. This being an incredibly valuable feature for telecommunications switching systems as minimal downtime is of the utmost importance. As a language Erlang's core principles are based on pattern matching and parallel processing. Erlang has an internal messaging system that can be used by its modules to communicate to one another.

C. Concurrency

Concurrency is the fact that two events or circumstances are happening or existing at the same time. Explained in a programming and cloud storage context: when two or more users are trying to access and interact with a shared resource.

D. Semaphore

A semaphore within a computational science context is regarded as a variable or abstract data type that is used to control access to a shared resource within a system. Semaphores are often used in systems which use multiple processes that are running concurrently.

III. Problem Definition

Concurrency errors can occur as a consequence of two user accessing and altering a file simultaneously. Let's take a cloud storage service like Dropbox as an example. In this example we assume that the system has no concurrency methods or solutions. First let's look at single user reading and writing a file to a cloud storage server, we can present this using a small flow chart:

read.User1[1] → write.User1[2]

What we observe here is a basic read and write program which seems to behave correctly and without error. However if we have a file that is shared between two users who both have made changes to said file and are simultaneously uploading the file back to the server. What we will observe is the following:

read.User1[1] → read.User2[1] → write.User1[2] → write.User2[2]

As you can see both files are simultaneously read and written however the result is user 2 overriding user 1 and user 1's data being destructively updated due to the arbitrary interleaving of read and write actions. If implemented within businesses and whilst handling valuable information such destructive properties could have catastrophic consequences within a cloud storage system. We refer to this problem as interference. To further elaborate on this problem, we can present it using a Finite-state machine. If we imagine a system for a shop warehouse that uses a cloud computing database to keep track of its stock and uses this information to keep stock levels at the desired state. The function below would be the function called to update the stock level after a purchase:

```
Reducestock ( Amount ) ->
        OldStock = get_stock () ,
        NewStock = OldStock - Amount ,
        set_Stock ( NewStock ).
```

This System will work flawlessly if there is only ever 1 shopkeeper updating the stock. The shopkeeper could use this system indefinitely without an error occurring. However as soon as there is a second shop opened with a separate stock updating system, there is a chance that an interference problem could arise. If the shopkeepers were to both update the stock at the same time the following would happen.

get_stock()[1] → get_stock()[1] → set_stock[2] → set_stock[2]

As we can see because both systems are interacting with the current stock they do not take each other into account. Let's say our original stock level was 10 and both shopkeepers sold a single item we would expect the stock level to decrease to 8 items. However what we observe is that in some cases there is a possibility that the stock will be set to 9 when both sales have occurred simultaneously.

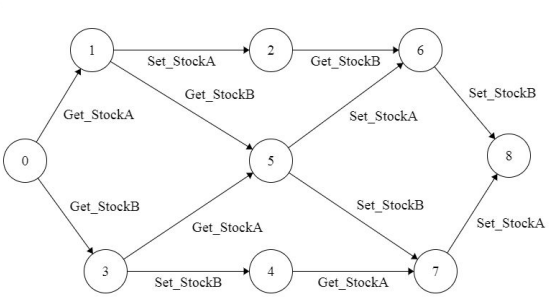

Fig. 1 Modeling interference within the Reducestock process in a FSM

As you can see in figure 1 there are interleaving traces in the system. In these traces going towards state 5 of the FSM we can clearly see that the current stock level is got before it has been set by the previous interaction, thus causing interference.

IV. Problem Solution

A. Mutual exclusion

Mutual exclusion also referred to as “Mutex”, is a semaphore used within programming languages. Mutex is often used within programming languages that share memory with one another [4]. However by looking into mutual exclusion what we find is that is properties have had a tremendous impact on the solution of concurrency and interference. A mutual exclusion works rather intuitively by blocking all access to a particular recourse when in use by another process. It does this until the original process is no longer using the recourse allowing any other single process to occupy the recourse [7]. Furthermore this Mutex solution works even more efficiently if the system has a queuing functionality implemented. Allowing other processes to queue to use a specific recourse.

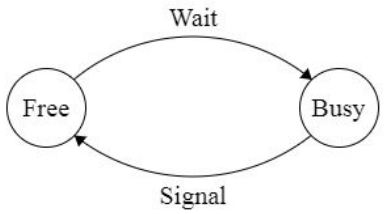

Fig. 2 FSM of a primitive MutEx system.

As you can see in Figure 2, the FSM of a Mutex semaphore is rather simple however it allows programs to overcome the huge obstacle of concurrent interference. If we apply this solution to our stock example what we will find is a swift resolution.

```
Reducestock ( Amoun;t ) ->
        Mutex: wait()
        OldStock = get_stock () ,
        NewStock = OldStock - Amount ,
        set_Stock ( NewStock ),
        Mutex: signal().
```

What we observe here is rather remarkable, when the process “Reducestock” is called the first part of that process is to run the “Mutex: wait()”. This command will then apply the busy state to the process as seen on figure 2. This state will inherently lock the process as we have previously described. Rendering the process “Reducestock” inaccessible. The process will then function as expected updating the stock information on shared file. Once the process has finished its operation of setting the stock the “Mutex: signal()” command is run, thus freeing up the process for another user to access. What we would therefore observe is the following:

get_stock()[1] → set_stock[2] → get_stock()[3] → set_stock[4]

As we can see the stock levels are no longer read simultaneously and what is achieved is the desired result of removing all possibilities of interference from the system[9].

B. Alternative solutions

Although Mutex is intuitive and rather effective, it can be limiting in some cases. Therefore I shall describe alternative solutions to our concurrent interference problem. If we take an alternative example of a bus used for public transport. We assume that the bus has 8 seats for passengers and therefore the maximum number of passengers that can be on board at one time is 8. Now we want to have a system in place that will display the number of seats left on the bus. This system will also determine whether or not the driver can let additional passengers on the bus. To apply a Mutex solution to this problem would require a large amount of code as we would have to define each seat as a separate process. However by applying a counting semaphore to this problem we are able to exactly control the variable amount of people on the bus. Take a look at the following FSM for the passenger counting semaphore:

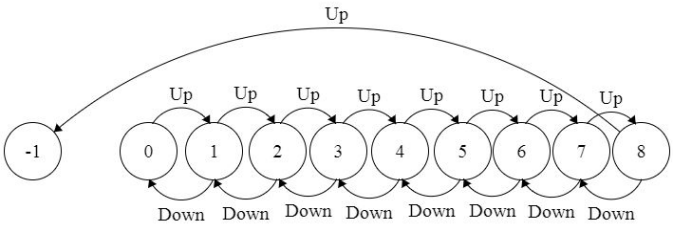

Fig. 3. Modelling a Counting semaphore in a FSM

As we can see from the FSM on figure 3 every time a passenger boards the bus the semaphore function “Up” is called, increasing the state and recording a passenger. This can continue until the 8th state is reached meaning that the 8th passenger has entered the bus, and therefore no other passengers are able to enter. As the FSM in figure 3 shows the system will return an error if another “Up” command is run. This of course is a oversimplified example, however this logic could be applied to our Dropbox scenario if we wanted a select amount of users to access a document simultaneously.

V. Discussion

To formally consider applying solutions to a problem one must always consider the possible limitations and negative impacts the proposed solution might bring to the system. One of the most probable limitations of Mutex is the possibility of “Deadlock”.

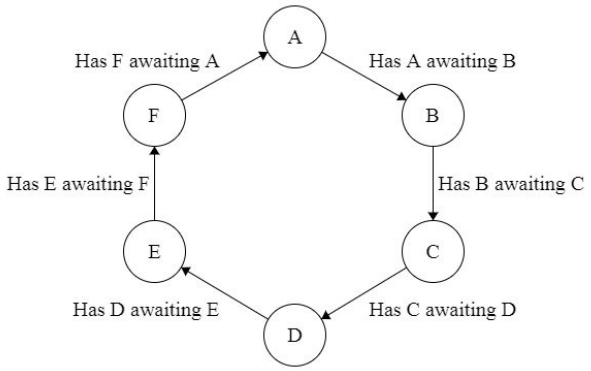

Fig. 4 Modelling FSM of Deadlock occurring.

The state of Deadlock is one where there is no possibility of an outgoing transition, therefore a continuous cycle takes place within the system indefinitely. Deadlock is visualised using a FSM in figure 4. Deadlock can be more clearly conceptualised by using the philosophers table example.

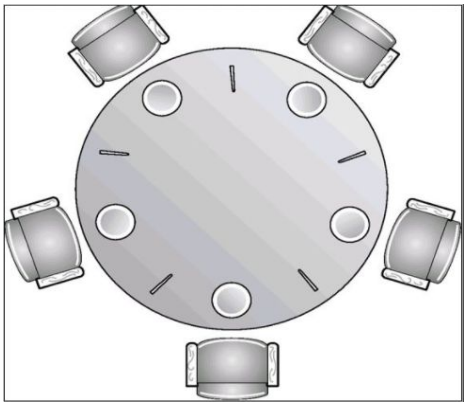

Fig. 3 Diagram of philosophers table.

The example states that five philosophers are dining at a circular table. In this example it is assumed that the only possible actions a philosopher can execute are thinking and eating. The philosophers spend their lives thinking except for when they are hungry at this point they will want to eat until they are full [8]. Each philosopher is given a bowl of spaghetti in front of them which they can consume only if they have two forks. One fork is placed in between each philosopher which means there are 5 forks on the table. The philosophers agree that they will only use the forks to the immediate left and right of them. What we can determine is that this system has a high probability of becoming Deadlocked. This would occur if 3 or more philosophers became hungry at the same time as they would not have both forks available. However Deadlock can only occur when a system is exposed to four specific conditions; The systems processes that are used under mutual exclusion are shared. The processes involved maintain the acquisition of the resources allocated to them whilst waiting to acquire additional resources. The system does not allow any process to be forcefully withdrawn from a recourse and only allows voluntary release of the recourse. The system inherently spawns a circular chain of processes due to every process maintaining is acquisition of its allocated recourse even if its successor is waiting to acquire it. Therefore by developing a system that actively avoids these four rules we can avoid Deadlocking. In the example of the philosophers we can avoid Deadlocking by introducing asymmetry into the system. If were to give each philosopher an integer and stated that every philosopher assigned to an even integer would pick up their left forks first and each philosopher with an odd number would pick up their right forks first. This would ensure that two philosophers would always be able to eat at the same time and thus avoiding Deadlock all together.

VI. Conclusion

This journal article was established to further investigate concurrency issues and specifically concurrency issues within cloud storage software such as Dropbox. We established that if the necessary precautions were not taken, a system with shared resources that are accessed by multiple processes simultaneously could potentially run into concurrent interference. However we established that by using a programming language like Erlang which is specifically designed for concurrent processing and by using specialised semaphores like MutEx we can avoid concurrency issues completely. We then discussed the limitations of MutEx. If a system wants to implement a specific number of processes that can access a shared resource simultaneously a more efficient solution would be to implement a counting semaphore rather than using MutEx. We further discussed possible system failures that could occur when using concurrency solutions such as Deadlock. However if this possible failure is taken into account when creating the system it can easily be avoided. Therefor we can conclude that although Mutex is not a solution to concurrency in all cases it can solve interference within a system effectively in a majority of cases. Furthermore we can more specifically conclude that in our evaluation of Mutex, it could be applied to most cloud file storage applications such as Dropbox.

VII. References

[1] "Company Info - Dropbox", Dropbox, 2017. [Online]. Available: https://www.dropbox.com/news/company-info. [Accessed: 14- Feb- 2018].

[2] *"About - Dropbox", Dropbox, 2017. [Online]. Available: https://www.dropbox.com/about. [Accessed: 14- Feb- 2018].*

[3] *"Cloud storage report 2017 - cloudrail", Cloudrail, 2017. [Online]. Available: https://blog.cloudrail.com/cloud-storage-report-2017. [Accessed: 14-02-2018]*

[4] *Seo, D. and Kim, S. and Song, G. (2017) 'Mutual Exclusion Method in Client-Side Aggregation of Cloud Storage', IEEE Transactions on Consumer Electronics, Vol. 63, No. 2, May 2017, pp. 185-19*

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

[6] *"Concurrent Programming" – Erlang Official Website [2018] http://erlang.org/doc/getting_started/conc_prog.html*

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

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

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