Bloom Clock Project to Characterize Causality in Distributed Systems

Prajwal Kishor Kammardi

November 2020

1 Introduction

The goal of this project was to study the false positives, accuracy, precision, and the false positive rate of the Bloom Clock. The vector clocks can solve the problem of determining the causality between events, but on a cost of large storage overhead. The Bloom filter is a probabilistic data structure can be used as a bloom clock. This project tries to implement and test the Bloom clock to characterize causality as stated in the paper[1]. The Protocol specified is implemented and its properties are studied. Some of the insights obtained are tabulated.

The accuracy of the bloom filter depends on the size of the filter (m), the number of hash functions (k), and the number of elements added to the set (n), here, n represents the number of processes. Bloom filters suffer from false positives and the nature of these are demonstrated in this simulation. The simulations makes various assumptions while executing and some trade-offs while processing the results which were because of the performance bottleneck of the executing machine. All images presented in this report are vectorized and hence, can be zoomed in freely.

2 System Model

2.1 Project Overview

The simulation part for the project is written in Scala. It is a general purpose programming language which compiles to JVM. It offers both functional and object-oriented style of writing the code. The project uses a toolkit called Akka. Akka is a free and open-source toolkit and run-time simplifying the construction of concurrent and distributed application on the JVM. Akka emphasizes on *actor-based-concurrency*. The actor model treats *actor* as the universal primitive of concurrent computation. An actor can make decisions, create actors, send messages etc., in response to any message it receives. This basically creates an abstraction of an asynchronous message passing model on the existing JVM Thread. No mutable data and no synchronization primitives are used.

2.2 Project Architecture

The following figure 1 illustrates the architecture flow diagram of the simulation:

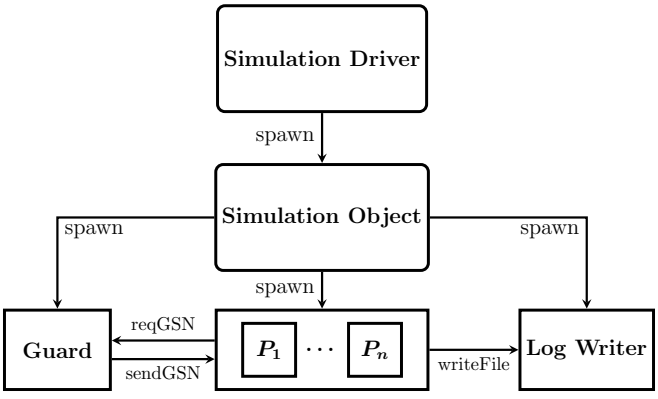


Figure 1: Architecture of Simulation

The architecture contains 5 major components. They are, *Simulation Driver, Simulation Object, Guard, Process Group, Log Writer*. Their working and the assumptions made while developing them are explained below. All the **messages** that the actors respond to are **bold faced** and the *actors* are *italicized*

2.2.1 Simulation Driver

The Simulation Driver is the main entry point for the application. It reads the simulation parameters (n, k, m) from a configuration file. The configuration file also contains some more parameters for file naming and the run-name, a folder that is created to hold all files of same run. Multiple runs of different parameters can be run at a single instance. The Simulation Driver creates a *Main* Actor-System. The *Main* Actor-System is the parent actor which *spawns* the *Simulation Object* actor. The *Main* actor sends a **Start** message to the Simulation object to begin a run . As the Simulation Driver can run multiple simulations at a single instance, there is a timeout between each run, so that all the actors can finish sending messages to each other, and then shut down gracefully at the final clock tick of that run is reached.

2.2.2 Simulation Object

The *Simulation Object Actor* is responsible for two things:

1. When it receives a **Start** message from the parent actor, it spawns its children, the *Guard Actor*, *Process Actors* (The number is n, provided in configuration), *Log Writer Actor*.
2. After spawning it's children, the Simulation Object goes to a scheduled *idle* state. It stays at this state for 60ms and then switches to *active* state. The *active* state sends a message to all n process actors to **ExecuteSomething** and then switches to *idle* state. The *Simulation Object* acts as a clock tick for each of the n processes to perform some task. There is an upper limit for clock ticks, if the clock tick counter exceeds this value, the Simulation Object waits for a certain time for all the *Process* to finish their execution and then shuts down the system.

2.2.3 Guard Actor

The *Guard* actor is a store for *Global Sequence Numbers (GSN)*. It is responds for messages below:

1. **IncremendAndReply*X***: Here, there are 3 messages ***X*** can be the event that a process is trying to be, i.e., **Sender, Receiver, Internal**. A process at that execution sends this message to get the *latest GSN* from the guard actor. The Guard actor will not process a message from any other actor while it is executing an increment. This ensures synchronization. The Guard actor on receipt of these messages increments its counter and replies with the latest value using,
2. ***X*Response**: Here ***X*** can be **Send, Receive, Internal**. It is a message sent to the process as a reply to send *GSN* request above. It marshals the *GSN* as the payload in the reply.

2.2.4 Log Writer

The *Process* actor after each event sends its state via the **WriteToFile** message to this actor. It is then written asynchronously to a file using Akka-streams.

2.2.5 Process

The main components of the actor system. There are n process actors in the system n = *[100, 200, 300]*. Each process actor has *process-ID, Vector-Clock[n], Bloom-Clock[n], Guard-Actor-Reference, Log-writer-Reference, Other-Process-Refs[n - 1], Event-Counter* as its state variables. A *Process actor* responds to these messages:

1. **InitProcess**: A *Process* is initialised. All its state variables are initialised. The process gets a **ListingResponse** from a system actor called *Receptionist* of other processes existing in the system, which then is used to initialize the *Other-Process-Refs* list.
2. **ExecuteSomething**: It is a message that comes in every clock tick of 60ms from the *Simulation Object Actor*. The *Process* executes either a *Send, Receive, Internal, None* event based on a probability. The probability can be varied. The *None* event is optional. It executes the respectively named events. It sends **IncremendAndReply*X*** to the *Guard Actor* and waits for a ***X*Response**.
3. **RecvMessage**: This message is sent by another *Process*. When a receiving process gets this message, before executing any event, it sends a **IncrementAndReplyReceiver** for the GSN of that event it is going to execute. Upon receipt of **RecvResponse**, the process marshals it to its readable type, **ReceivedReceiveGSN**. This ensures synchronization and it would not execute until receiving a GSN.

4. **ReceivedReceiveGSN**: When When a process receives a **RecvResponse** , it marshals it to a readable format that is, **ReceivedReceiveGSN**. increments its local clocks using the receipt clock from the sender and itself, updates event counter. Sends its state to *Log Writer*.

5. **ReceivedSendGSN**: When a process receives a **SendResponse** , it marshals it to a readable format that is, **ReceivedSendGSN**. Here, the process chooses another random *Process*, increments its local clocks, event counter, then sends **RecvMessage** to that process with its clock values marshaled in the message. Sends its state to *Log Writer*.

6. **ReceivedInternalGSN**: When a process receives a **InternalResponse** , it marshals it to a readable format that is, **ReceivedInternalGSN**. Here, the process increments its local clocks, event counter. Sends its state to *Log Writer*.

2.3 Simulation Execution

The simulation was executed for $n = [100, 200, 300]$. $n = 500$ was very inconsistent as the system that was executing the simulation could not handle the load of 500 actors. The number of hash functions was $k = [2, 3, 4]$ for each n and the Bloom Counter size was $m = [ceil(0.1n), ceil(0.2n), ceil(0.3n)]$. The experiments were repeated for different probabilities of Send and Internal events shown in the list of tuples $(Pr_{send}, Pr_{intr}) = [(1, 0), (0.5, 0.5), (0.1, 0.9)]$. Each send event ensured a receive event, hence was not specified in the probability. So, these ensured that there were 81 runs. The execution procedure was as follows:

1. A Configuration file will contain the n, k, m values . Multiple values for each could be passed as a list

2. The Program is run and upon execution of each individual combination of (n, k, m), a *csv* file would be created and the events and states would be written.

3. Each *csv* contains GSN, Process-ID, Local-Event-Counter, Bloom-Clock, Vector-Clock, Event-Type. The delimiter is ***;***

4. The Results were analyzed in a python - numpy - pandas environment.

3 Results

3.1 Analysis of Probability of Positives

pr_p is defined as $pr(B_z \geq B_y)$. The Probability of a positive result given by a Bloom Counter. The steps to process the data were as follows:

3.1.1 Data Setup

1. The value of y is fixed for the event with $GSN = 10n$. B_y is the respective Bloom Clock value.

2. The value of z is all events in range $[10n + 1, n^2 + 10n]$. B_z are the respective Bloom Clock values.

3. The z slice ensures that all processes are causally related to one another at least once.

4. Data was obtained by varying the number of processes $n = [100, 200, 300]$, hash functions $k = [2, 3, 4]$, and size of the Bloom Counter, denoted by $m = [0.1n, 0.2n, 0.3n]$, and probability of internal message $pr_{int} = [0, 0.5, 0.9]$

5. A graph representing pr_p vs GSN was plotted *(y-axis vs x-axis)*. pr_p is defined as:

$$\widehat{pr_p}(k, m, B_y, B_z) = \prod_{i=1}^{m}(1 - \sum_{l=0}^{B_y[i]-1} b(l, B_z^{sum}, 1/m)) \tag{1}$$

6. Every 10^{th} event was considered to make the graph look cleaner.

3.1.2 Observations

Some observations are displayed below.

1. A general graph will be of the form as in Figure 2. There is a steady increase in the positives given by the clock. This means, the probability of positive is low if z is close to y (y is fixed). This is mainly because of increase in the B_z^{sum} w.r.t y's m counters as in Eq(1).

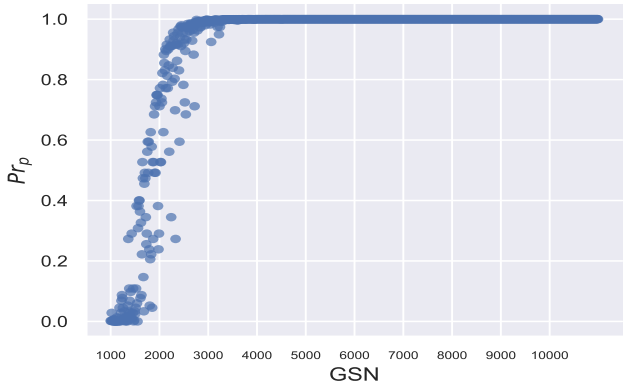
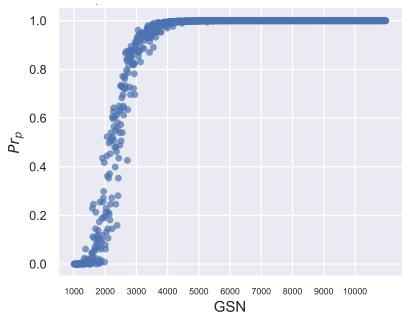
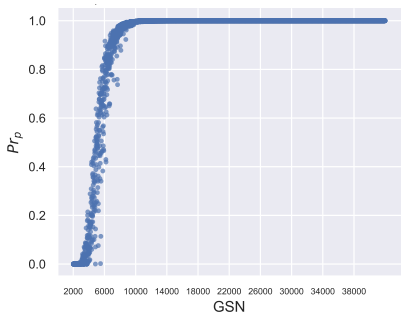


Figure 2: Pr_p against GSN with $Pr_{int} = 0$, $(n, k, m) = [100, 2, 10]$

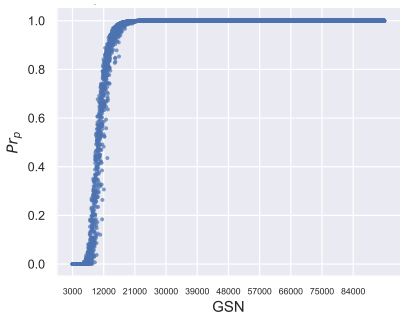
2. When, n ($[100, 200, 300]$) is varied, keeping the other components the same that is, $(k, m, Pr_{int}) = (2, 0.1n, 0)$.



(a) $(n, k, m) = [100, 2, 30]$



(b) $(n, k, m) = [200, 2, 20]$

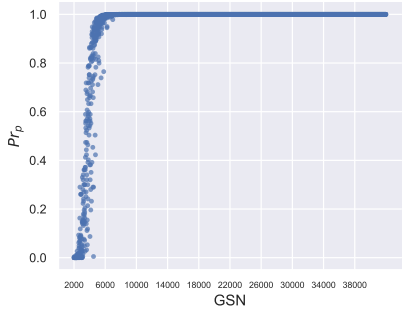


(c) $(n, k, m) = [300, 2, 30]$

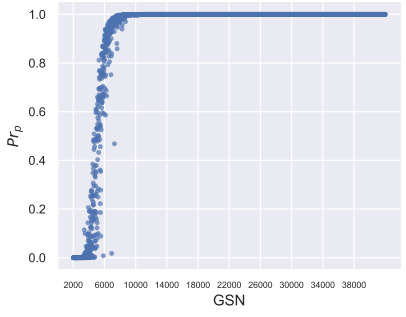
Figure 3: Pr_p against GSN with $Pr_{int} = 0$, varying the number of processes

From the figures 3a, 3b, 3c, one can observe that there is no change in the slope. In all the experiments, keeping the n variable, does not make any difference to the rate of pr_p. This shows that changing the number of processes do not affect the positive values given by the Bloom Clock, given other parameters are constant.

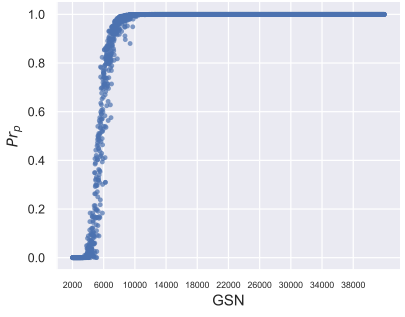
3. When, m is varied ($[0.1n, 0.2n, 0.3n]$), keeping the other components the same that is, $(n, k, Pr_{int}) = (100, 2, 0)$.



(a) $(n, k, m) = [200, 4, 20]$



(b) $(n, k, m) = [200, 4, 40]$

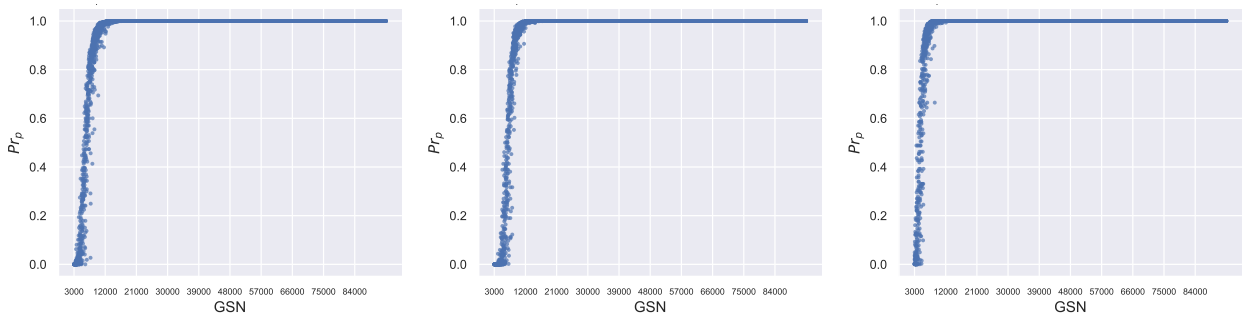


(c) $(n, k, m) = [200, 4, 80]$

Figure 4: Pr_p against GSN with $Pr_{int} = 0$, varying the counter size

From the figures 4a, 4b, 4c one can observe there is a shift in the graph. That is, the z values closer to y have identical probability values. This is because, as m changes, there are more counter variables in the bloom clock to increment, therefore, the binomial relation will be closer to 1. The probability of positives stays close to 0. It starts to increase once at least all the m counters in B_z differ from B_y.

4. When, k ($[2, 3, 4]$) is varied, keeping the other components the same that is, $(n, m, Pr_{int}) = (300, 0.1n, 0)$.

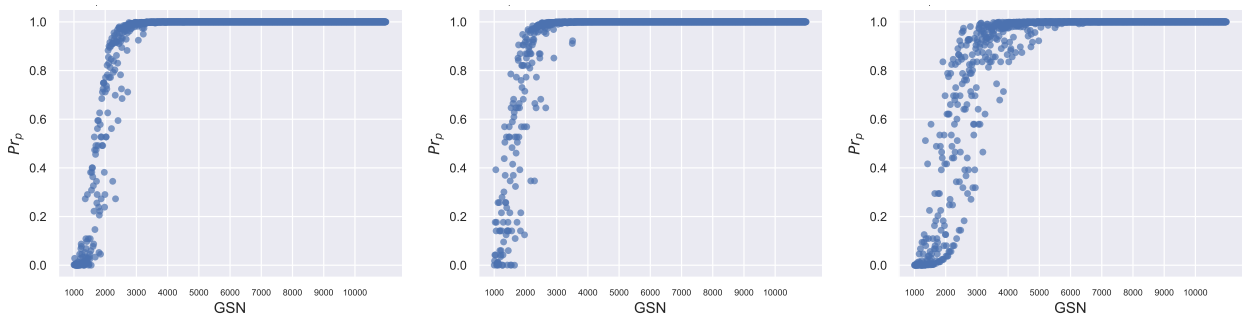


(a) $(n, k, m) = [300, 2, 30]$ (b) $(n, k, m) = [300, 3, 30]$ (c) $(n, k, m) = [300, 4, 30]$

Figure 5: Pr_p against GSN with $Pr_{int} = 0$, varying the number of hash functions

From the figures 5a, 5b, 5c one can observe, The graph is dense in the lower part $m = 3$ and it tapers as m increases. Although the value of m does not directly affect the nature of the graph, more counters are filled in one event as m is increased. This makes the probability of positive for an event z more, as the B_z^{sum} increases m counts at each event. To be more precise, the pr_p value for same GSN would be a higher as m is increased.

5. When pr_{int} is varied, and keeping (n, k, m) is kept constant.



(a) $(n, k, m) = [100, 2, 10]$ (b) $(n, k, m) = [100, 2, 10]$ (c) $(n, k, m) = [100, 2, 10]$

Figure 6: Pr_p against GSN with $Pr_{int} = (0, 0.5, 0.9)$, respectively, varying the probability

From the figures 6a, 6b, 6c, if a probability of internal message is changed, as internal messages are increased, a processes may not be causally related to another for a long time, therefore, only a small amount of counters will be incremented at each process event (k) if it is an internal event. Therefore pr_p remains constant for multiple GSN as a process may just be updating it's own clock and the Bz value will be increasing by a factor of k at most of the times. Only when a receive event comes to that process, the values of B_y are added to the B_z clock. Therefore, there is more scatter in the plots as internal events are increased.

3.2 Analysis of Probability of False Positives

A false positive occurs when Bloom clock predicts a causality between two events when there is none, in reality i.e.,

$$pr_{fp} = pr(y \not\to z).pr(B_z \geq B_y) = (1 - pr_p) \cdot pr_p \tag{2}$$

We can also use,

$$pr_{fp} = pr(B_z \geq B_y) = (1 - pr_p) \cdot pr_{\delta(p)} \tag{3}$$

Where $pr_{\delta(p)}$ is 1 if $(B_z \geq B_y)$, else 0 . The steps to process the data were as follows.

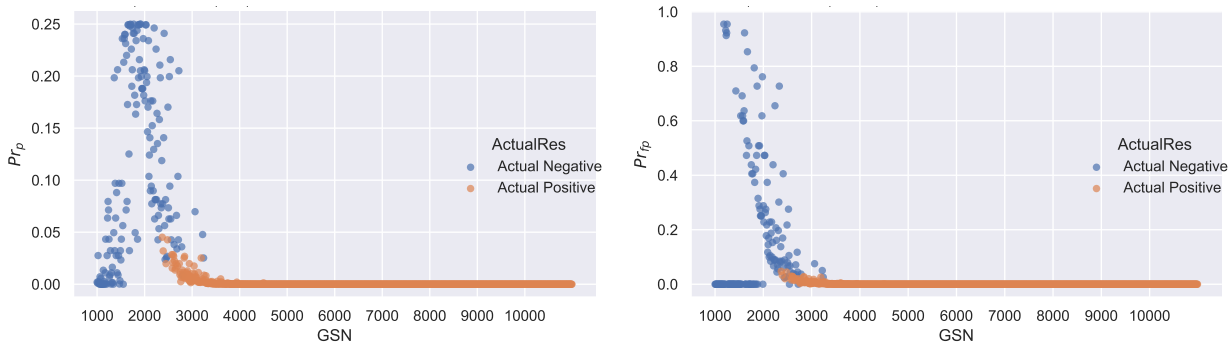
3.2.1 Data Setup

1. The value of y is fixed for the event with $GSN = 10n$. B_y is the respective Bloom Clock value.
2. The value of z is all events in range $[10n + 1, n^2 + 10n]$. B_z are the respective Bloom Clock values.
3. The z slice ensures that all processes are causally related to one another at least once.
4. Data was obtained by varying the number of processes $n = [100, 200, 300]$, hash functions $k = [2, 3, 4]$, and size of the Bloom Counter, denoted by $m = [0.1n, 0.2n, 0.3n]$, and probability of internal message $pr_{int} = [0, 0.5, 0.9]$
5. A graph representing pr_{fp} vs GSN was plotted *(y-axis vs x-axis)*, using Eq(2) and Eq(3)
6. $pr_{\delta(p)}$ is calculated using the Bloom clock values. To find the actual causality for Eq(2), the vector clocks are used. $y \rightarrow z$ if $\exists i | (V_y[i] < V_z[i])$

3.2.2 Observations

Some observations are displayed below.

1. General graphs for Pr_{fp} will be of two forms based on the type of pr_p value. If the probabilistic pr_p is taken, there is a bell curve, the range would be $[0, 0.25]$. If the step function is considered, the first half of the bell-curve is cut, and a similar graph as the first one is seen and the range would be $[0, 1]$. This is mainly because of the step function, which does not become 1 until $B_z \geq B_y$.

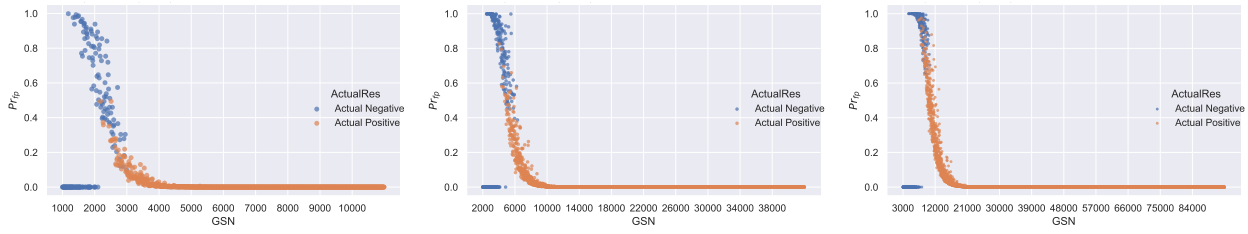


(a) pr_{fp} using Eq(2)$(n, k, m) = [100, 2, 10]$ (b) pr_{fp} using Eq(2) $(n, k, m) = [100, 2, 10]$

Figure 7: Pr_{fp} against GSN with $Pr_{int} = 0$

Figure 7a and 7b have actual positives representing the Bloom clock correctly predicting the causality.The false positive rate is high at the beginning as there are lesser number of processes between y and z. As the gap increases, there would be some sort of causality established between y and z. Henceforth, only the graph of Eq(3) will be shown.

2. When, n ($[100, 200, 300]$) is varied, keeping the other components the same that is, $(k, m, Pr_{int}) = (2, 0.1n, 0)$.

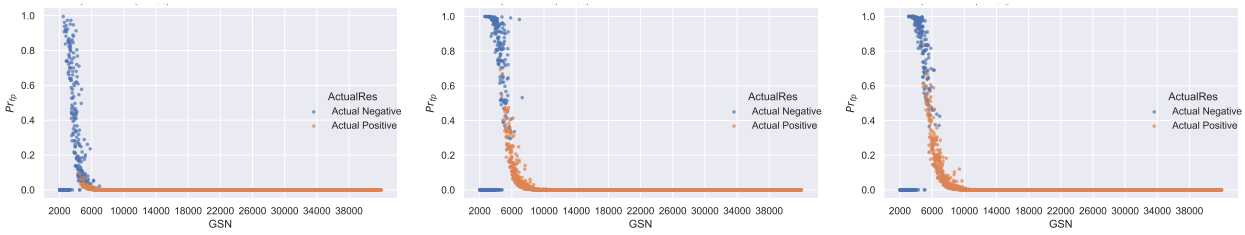


(a) $(n, k, m) = [100, 2, 30]$ (b) $(n, k, m) = [200, 2, 20]$ (c) $(n, k, m) = [300, 2, 30]$

Figure 8: Pr_{fp} against GSN with $Pr_{int} = 0$, varying the number of processes

From the figures 8a, 8b, 8c, one can observe that there is a significant change in the false positive numbers. As the process are increased, the false positives decreases.

3. When, m is varied ($[0.1n, 0.2n, 0.3n]$), keeping the other components the same that is, $(n, k, Pr_{int}) = (100, 2, 0)$.

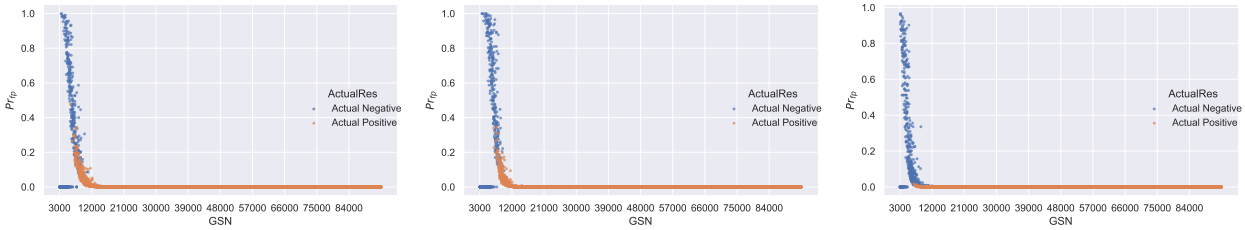


(a) $(n, k, m) = [200, 4, 20]$ (b) $(n, k, m) = [200, 4, 40]$ (c) $(n, k, m) = [200, 4, 80]$

Figure 9: Pr_{fp} against GSN with $Pr_{int} = 0$, varying the counter size

From the figures 9a, 9b, 9c one observe that the rate of false positives predicted decreases as m is increased. As m increases, the probability of same bit getting incremented decreases. Therefore, effectively decreasing false positive rate of the filter.

4. When, k ($[2, 3, 4]$) is varied, keeping the other components the same that is, $(n, m, Pr_{int}) = (300, 0.1n, 0)$.

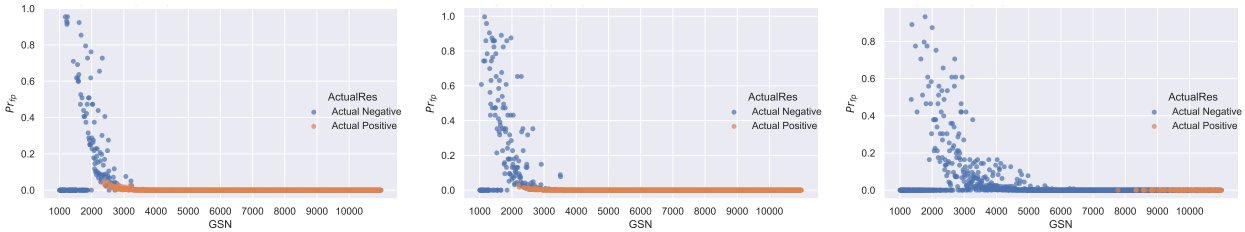


(a) $(n, k, m) = [300, 2, 30]$ (b) $(n, k, m) = [300, 3, 30]$ (c) $(n, k, m) = [300, 4, 30]$

Figure 10: Pr_p against GSN with $Pr_{int} = 0$, varying the number of hash functions

From the figures 10a, 10b, 10c. The false positive rate increases up to a certain decreases in k and then increases. So choosing an optimal value is essential.

5. When pr_{int} is varied, and keeping (n, k, m) is kept constant.



(a) $(n, k, m) = [100, 2, 10]$ (b) $(n, k, m) = [100, 2, 10]$ (c) $(n, k, m) = [100, 2, 10]$

Figure 11: Pr_p against GSN with $Pr_{int} = (0, 0.5, 0.9)$, respectively, varying the probability

From the figures 11a, 11b, 11c, if a probability of internal message is changed, as internal messages are increased, a processes may not be causally related to another for a long time, but, due to the property of bloom filter, the same counter for another event might get incremented and increases the false positive rate. Therefore establishing a causality increases the probability of false positives. It is ideal to have a good mix of send and internal messages.

3.3 Analysis of Accuracy, Precision, False Positive Rate

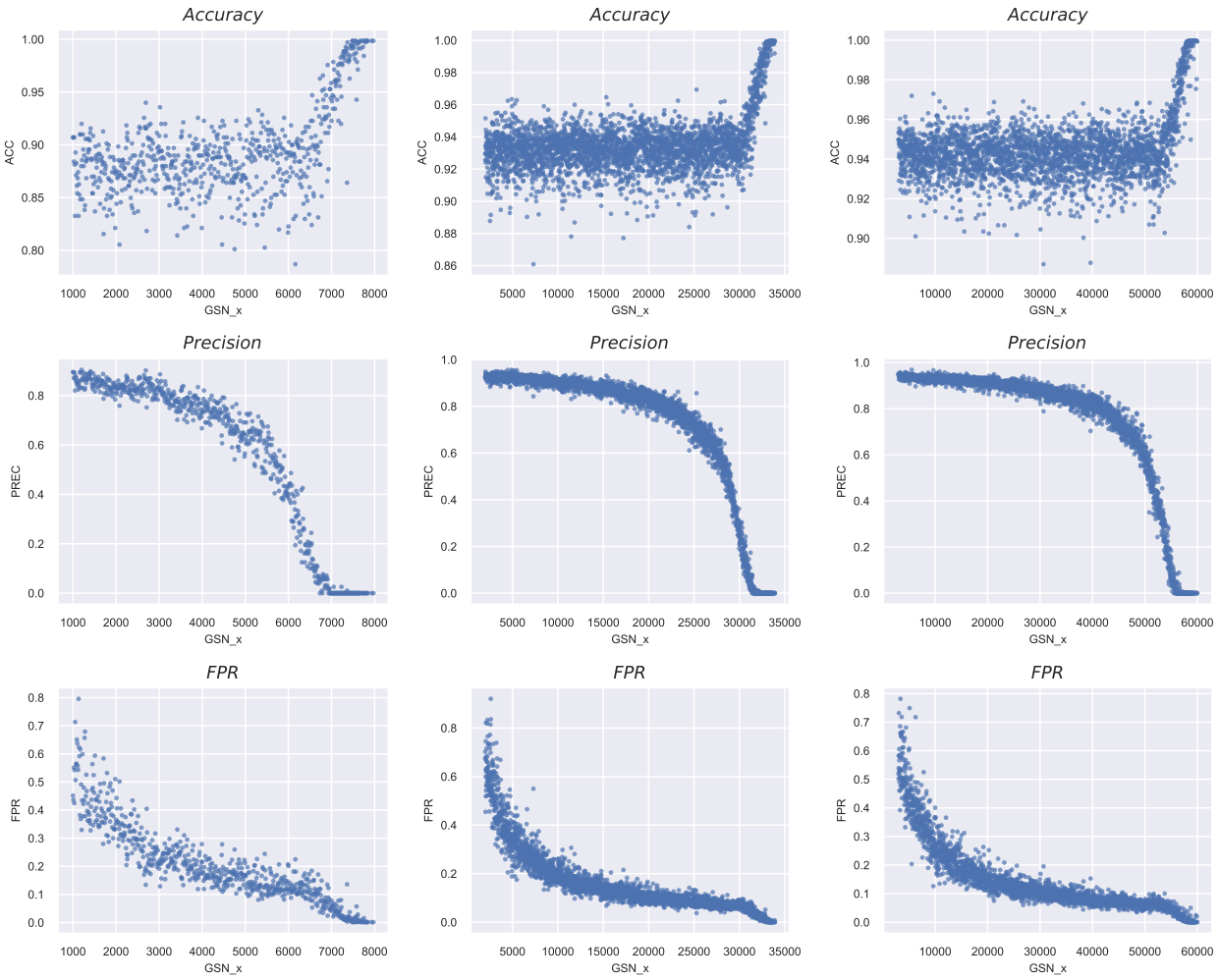
These are the general metrics to measure any probabilistic model. The True Positive, False Positive, True Negative, and False Negative values are computed for a particular slice.

3.3.1 Data Setup

1. The value of y is fixed for the event with $GSN = 10n$. B_y is the respective Bloom Clock value.
2. The value of z is all events in range $[10n + 1, n^2 + 10n]$. B_z are the respective Bloom Clock values.
3. The z slice ensures that all processes are causally related to one another at least once.
4. Data was obtained by varying the number of processes $n = [100, 200, 300]$, hash functions $k = [2, 3, 4]$, and size of the Bloom Counter, denoted by $m = [0.1n, 0.2n, 0.3n]$, and probability of internal message $pr_{int} = [0, 0.5, 0.9]$
5. Each event z id paired with every other z, Thus effectively having ${}^nP_2$ pairs. Consider them to be x, x' For each pair, the TP, TN, FN, FP causality values are calculated using their respective bloom clocks and vector clocks. FN is always 0.
6. To calculate the accuracy of each Bloom clock event at any GSN, the computed table is grouped by x then totalled. An average of that group gives the precision, accuracy and FPR of the Bloom Clock event.

3.3.2 Observation

1. A general trend, as the GSN increases, the accuracy stays constant and it rapidly increases at the end of the slice. Precision stays fairly constant and at the end of the slice shoots down. The FPR, decreases at a constant rate until some point, then it tapers towards 0.
2. When, n ($[100, 200, 300]$) is varied, keeping the other components the same that is, $(k, m, Pr_{int}) = (2, 0.1n, 0)$.



(a) $(n, k, m) = [100, 2, 30]$ (b) $(n, k, m) = [200, 2, 20]$ (c) $(n, k, m) = [300, 2, 30]$

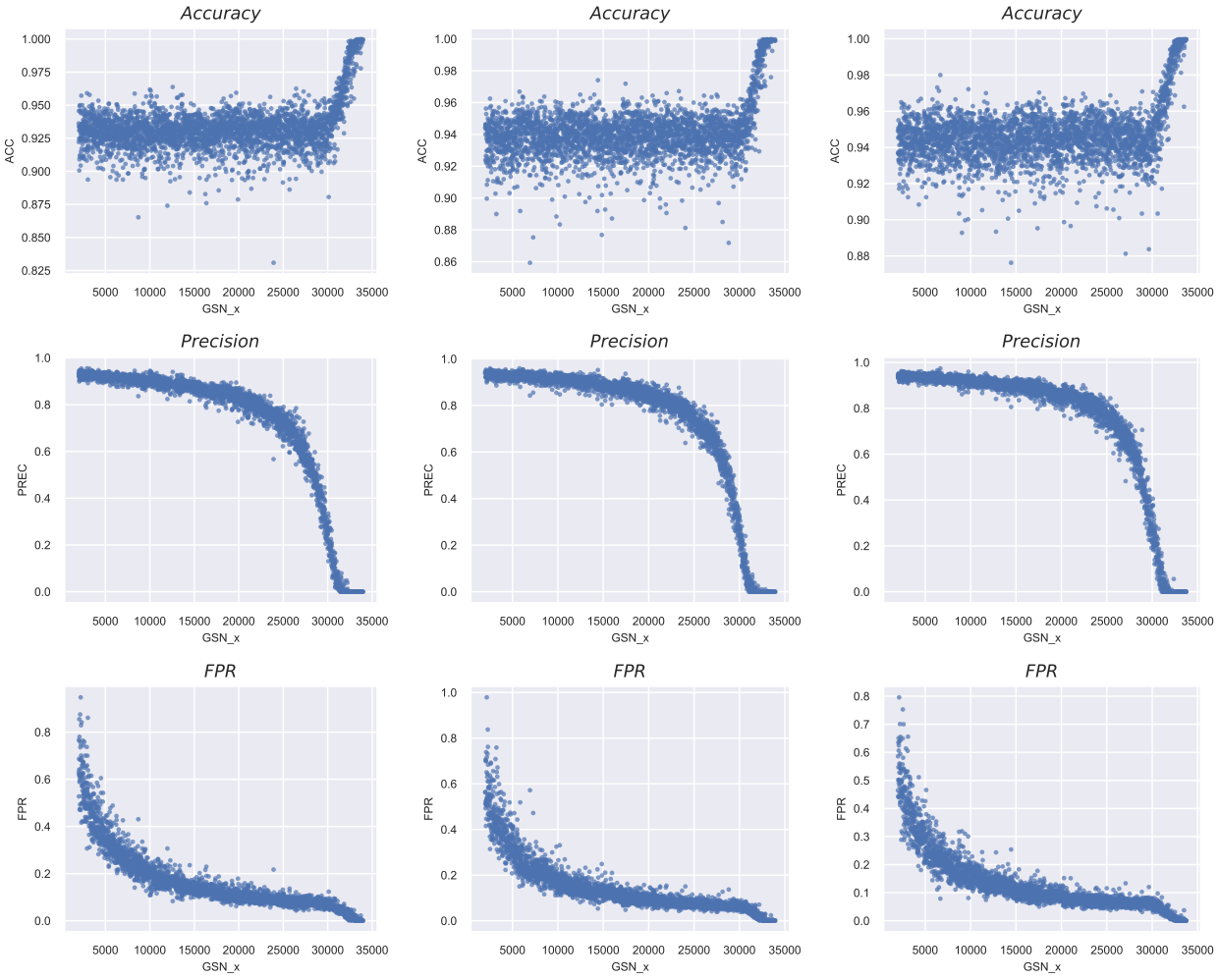
Figure 12: Pr_{fp} against GSN with $Pr_{int} = 0$, varying the number of processes

n, k, m	ACC	PREC	FPR
100, 2, 30	0.89	0.57	0.20
200, 2, 20	0.93	0.72	0.16
300, 2, 30	0.94	0.75	0.14

Table 1: Mean values of Figure 12

From the figures 12a, 12b, 12c and Table 1, as the number of process increase, the accuracy increases and the false positive rate decreases.

3. When, m is varied ($[0.1n, 0.2n, 0.3n]$), keeping the other components the same that is, $(n, k, Pr_{int}) = (100, 2, 0)$.



(a) $(n, k, m) = [200, 4, 20]$ (b) $(n, k, m) = [200, 4, 40]$ (c) $(n, k, m) = [200, 4, 80]$

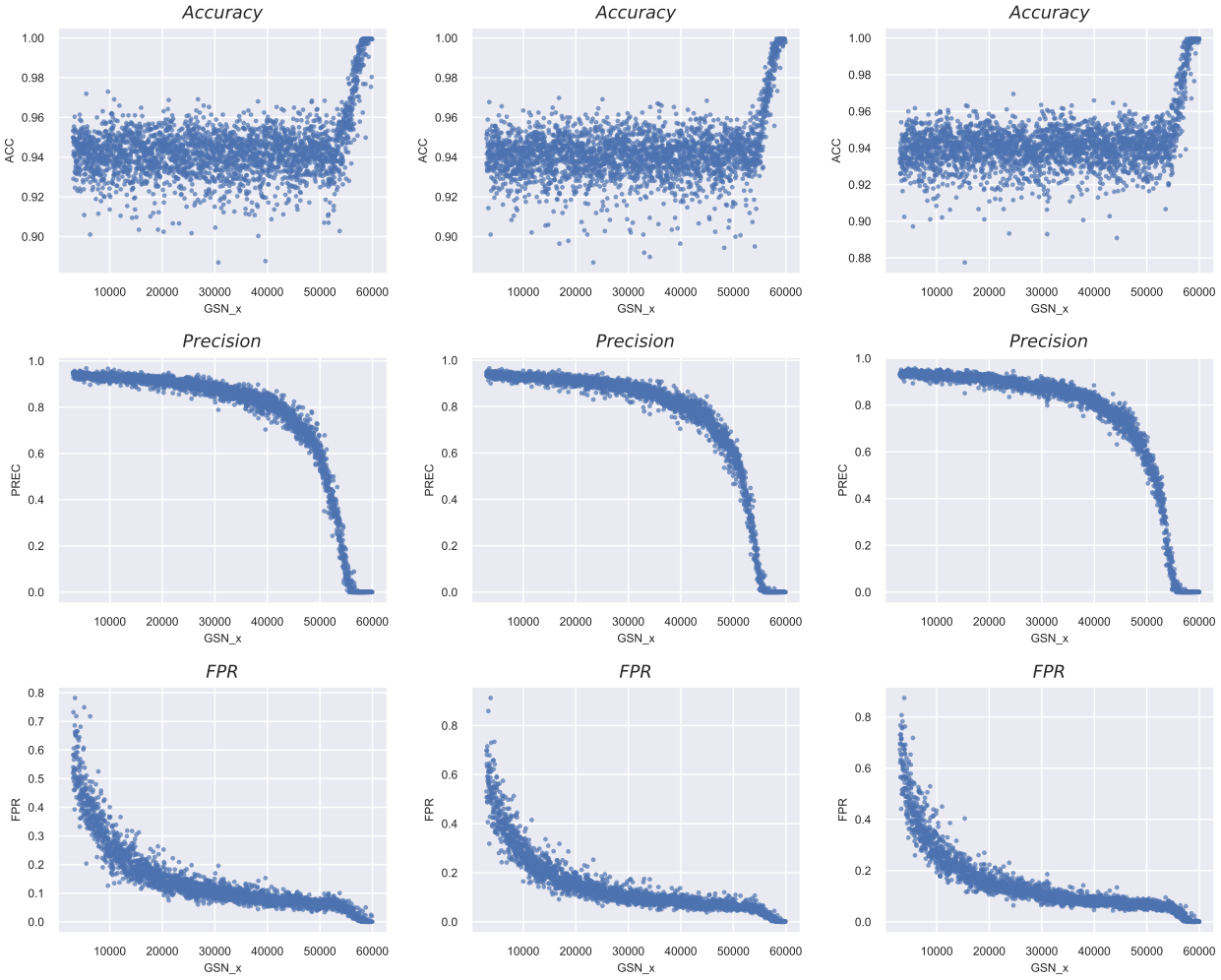
Figure 13: Metrics against GSN with $Pr_{int} = 0$, varying the counter size

n, k, m	ACC	PREC	FPR
200, 4, 20	0.93	0.72	0.16
200, 4, 40	0.94	0.74	0.14
200, 4, 60	0.95	0.75	0.13

Table 2: Mean values of Figure 13

From the figures 13a, 13b, 13c and Table 2 one observe that the rate of false positives predicted decreases as m is increased. As m increases, the probability of same bit getting incremented decreases. Therefore, effectively decreasing false positive rate of the filter.

4. When, k ($[2, 3, 4]$) is varied, keeping the other components the same that is, $(n, m, Pr_{int}) = (300, 0.1n, 0)$.



(a) $(n, k, m) = [300, 2, 30]$ (b) $(n, k, m) = [300, 3, 30]$ (c) $(n, k, m) = [300, 4, 30]$

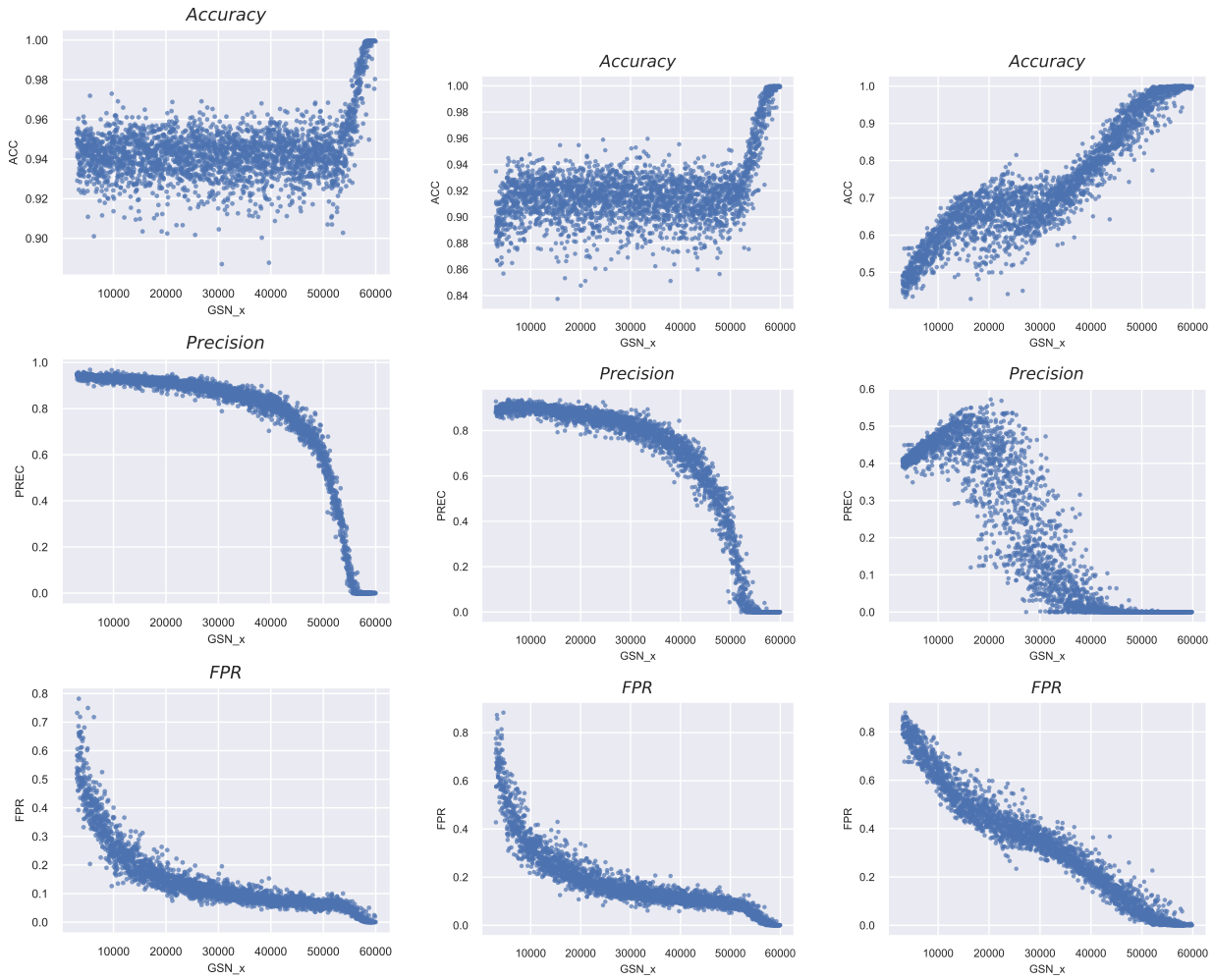
Figure 14: Metrics against GSN with $Pr_{int} = 0$, varying the number of hash functions

n, k, m	ACC	PREC	FPR
300, 2, 30	0.944	0.754	0.142
300, 3, 30	0.943	0.752	0.145
300, 4, 30	0.952	0.751	0.148

Table 3: Mean values of Figure 14

From the figures 14a, 14b, 14c and Table 3. The false positive rate decreases up to a certain extent and then increases in k. The FPR starts at a lower rate for $k = 3$. For $k = 4$ the accuracy values have a lesser deviation . So choosing an optimal value is essential.

5. When pr_{int} is varied, and keeping (n, k, m) is kept constant.



(a) $(n, k, m) = [300, 2, 30]$ (b) $(n, k, m) = [300, 2, 30]$ (c) $(n, k, m) = [300, 2, 30]$

Figure 15: Metrics against GSN with $Pr_{int} = (0, 0.5, 0.9)$

Pr_{int}	**ACC**	**PREC**	**FPR**
0	0.944	0.754	0.142
0.5	0.919	0.667	0.181
0.9	0.733	0.204	0.339

Table 4: Mean values of Figure $n, k, m = (300, 2, 30)$ referring to figure 15

From the figures 15a, 15b, 15c, if a probability of internal message is changed, as internal messages are increased, a processes may not be causally related to another for a long time, but, due to the property of bloom filter, the same counter for another event might get incremented and increases the false positive rate. The accuracy of the bloom clock with most internal messages is lower than others, and the accuracy of only send instances is the highest, therefore, bloom clocks behave well for systems with lot of inter process communication.

4 Conclusion

This project demonstrates a simulation of Bloom clocks. Some observations were demonstrated on the metrics such as accuracy, precision, and false positive rates of bloom clock for detection of causality in distributed systems. The trade-off between accuracy and space is demonstrated, with various other scenarios. Choosing optimal values for k, m is imperative when the number of processes are known. Design decisions also have to be considered as bloom clocks are not very effective in certain distributed scenarios, as demonstrated, that is, when there is very less inter process communication. The trade-offs to be considered while designing distributed

systems keeping bloom clocks in picture make for a great research topic and would greatly benefit the modern computing systems.

5 Future Work

1. An estimation of metrics just by using bloom clock values needs to be performed.
2. Simulation of message failures that occur in real life systems and how bloom clock can handle them.
3. Simulation on a larger number of process using a more powerful workbench.

6 References

References

[1] Ajay D. Kshemkalyani and Anshuman Misra. The bloom clock to characterize causality in distributed systems. In Leonard Barolli, Kin Fun Li, Tomoya Enokido, and Makoto Takizawa, editors, *Advances in Networked-Based Information Systems*, pages 269–279, Cham, 2021. Springer International Publishing.