UNIVERSITY OF TARTU
Institute of Computer Science
Cybersecurity Curriculum

Sander Mikelsaar

Empirical Study of Asynchronous Batch Codes

Master's Thesis (30 ECTS)

Supervisor: Vitaly Skachek, PhD
Co-supervisor: Eldho K. Thomas, PhD

Tartu 2019

Empirical Study of Asynchronous Batch Codes

Abstract: Batch codes were introduced by Y. Ishai, E. Kushilevitz, R. Ostrovsky and A. Sahai in 2004 for load balancing in distributed storage systems. As it is observed in a paper by A.-E. Riet, V. Skachek and E. K. Thomas, varying service times for the user requests could cause long waiting times in the system based on batch code, thus leading to a suboptimal performance. The asynchronous batch code model was introduced as a solution to this problem. In this thesis, to compare the two models, a system prototype was developed which was used to estimate the performance of these models. The constructed system model is described in detail. The thesis introduces a new parameter to the asynchronous system model, which is called "skip distance". The system performance can be improved by optimizing the value of this parameter. The results of the simulations are visualized and explained in detail.

Keywords:
Coding theory, storage system, load balancing, batch codes, asynchronous batch codes, system simulations

CERCS: P170, Computer science, numerical analysis, systems, control

Asünkroonsete Partiikoodide Empiiriline Uuring

Lühikokkuvõte: Partiikoodid esitleti Y. Ishai, E. Kushilevitz, R. Ostrovsky ja A. Sahai poolt aastal 2004 hajusfailisüsteemide koormusjaotuseks. A.-E. Riet, V. Skachek ja E. K. Thomas demonstreerisid artiklis, et erinevate kestvusega päringud partiikoode kasutavale failisüsteemile põhjustavad ooteaegu. Samas artiklis esitleti ooteaegade probleemi lahendamiseks asünkroonse partiikoodi mudel. Käesoleva lõputöö raames loodi kahe partiikoodimudeli võrdlemiseks süsteemi prototüüp, mille abil simuleeriti mõlemat mudelit kasutavaid failisüsteeme. Simulatsioonide töökäiku kirjeldatakse detailselt. Lõputöös võeti kasutusele asünkroonsel süsteemimudelil uus parameeter, mille abil suurendati süsteemi poolt saavutatavat täidetavat päringumahtu. Teostatud simulatsioonide tulemused visualiseeriti ning tulemusi kirjeldatakse detailselt.

Võtmesõnad:
Kodeerimisteooria, failisüsteem, koormusjaotus, partiikoodid, asünkroonsed partiikoodid, süsteemisimulatsioonid

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Batch codes were first introduced by Ishai, Kushilevitz, Ostrovsky and Sahai in [1] for the purpose of load-balancing in storage applications and amortizing computational costs in cryptographic protocols, such as private information retrieval (PIR) protocols. A class of batch codes - linear (computational) batch codes, studied in [2], [3], [4], [5], led to the introduction of asynchronous batch codes in [6].

In a large database of k items, using regular batch codes to distribute the items among n devices, a user chooses a batch of t distinct items to be retrieved from the storage devices. For this reason, however, the model has some limitations, as it always requires a user to choose a full batch of t items to be served. In most storage scenarios, databases contain a large number of files with users only being interested in retrieving a small number of them. For these scenarios, multiset batch codes provide a solution by creating a batch of t items by combining the queries of t users. Whereas regular multiset batch codes would require users to wait until a full batch of t items is requested from the database before serving the requests, as well as all of the requests from a previous batch to be finished before serving the next batch, the asynchronous model would be able to serve an incoming request for any of the k items in the database, with up to a total of t concurrent requests, as soon as a request has been served, without waiting for an entire batch to finish serving. This change to the batch code model can offer increased service rates if the requests served by a storage system have varied durations.

In this thesis, the asynchronous batch code model defined in [6] is studied and compared to the original batch code model. The main focus of the thesis is the comparison of batch codes, formed by using simplex codes, and asynchronous batch codes formed by the same codes. For the purpose of comparing the batch code models, a system prototype was developed which was used to run simulations of both models under varying conditions, such as different distributions of the requested items and the durations of the requests. The system prototype is described in the thesis as the methodology of studying the problem. To improve the service rates achieved by the asynchronous model, a new parameter, skip distance, is introduced to the model. Comparisons are also made to replication based storage models. Finally, the results and findings are presented and visualized, accompanied by the analysis of the applicability of the different system models.

2 Background

In this section, different batch code models are defined and described, along with definitions for simplex codes and statistical distributions used in the implementation of the system prototype used to run the simulations of batch codes using those models.

2.1 Notation

A finite (Galois) field over q is denoted as $\mathbb{F}_q$.
A space of vectors of length n over $\mathbb{F}_q$ is denoted as $\mathbb{F}_q^n$.
Definitions of basic notions in coding theory, as defined in [12]:

Definition 2.1. An (n, M) *code* over a finite alphabet Σ is a nonempty subset $\mathcal{C}$ of size M of Σ^n. The parameter n is called the *code length* and M is the code *size*.

Definition 2.2. An (n, M, d) code $\mathcal{C}$ over a field $\mathbb{F}_q$ is called *linear* if $\mathcal{C}$ is a linear subspace of $\mathbb{F}_q^n$.

Definition 2.3. A *generator matrix* of a linear $[n, k, d]$ code $\mathcal{C}$ is a $k \times n$ matrix whose rows form a basis of the code.

2.2 Batch Codes

Definition of batch codes, as given in [6]:

Definition 2.4. An (n, k, t) *batch code* $\mathcal{C}$ over a finite alphabet Σ is defined by an encoding mapping $C : \Sigma^k \longrightarrow \Sigma^n$ and a decoding mapping D such that:

- For any $\mathbf{x} \in \Sigma^k$ and $i_1, i_2, ..., i_t \in [k]$,

$$D(\mathbf{y} = C(\mathbf{x}), i_1, i_2, ..., i_t) = (x_{i_1}, x_{i_2}, ..., x_{i_t})$$

- The symbols in the query $(x_{i_1}, x_{i_2}, ..., x_{i_t})$ can be reconstructed from t respective pairwise disjoint recovery sets of symbols of $\mathbf{y}$ (the symbol x_{i_ℓ} is reconstructed from the ℓ-th recovery set for each ℓ, $1 \leq \ell \leq t$).

2.3 Linear Batch Codes

Definition of linear batch codes, as given in [3]:

Definition 2.5. An (n, k, t) batch code is *linear*, if the content of each server is a linear combination of original database elements.

Definition 2.6. A *linear combination* of n vectors $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$ over the field $\mathbb{F}_q$ is any vector of the form $a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + ... + a_n\mathbf{v}_n$, where the scalars $a_i \in \mathbb{F}_q$.

2.4 Asynchronous Batch Codes

Definition of asynchronous batch codes, as given in [6]:

Definition 2.7. An *asynchronous* (linear primitive multiset) $[n, k, t]$ *batch code* $\mathcal{C}$ is a (linear primitive multiset) batch code with the additional property that for any legal query $(x_{\ell_1}, x_{\ell_2}, ..., x_{\ell_t})$, for all $\ell_i \in [k]$, it is always possible to replace x_{ℓ_j} by some $x_{\ell_{t+1}}, \ell_{t+1} \in [k]$, such that $x_{\ell_{t+1}}$ is retrieved from the servers not used for retrieval of $(x_{\ell_1}, x_{\ell_2}, ..., x_{\ell_{j-1}}, x_{\ell_{j+1}}, ..., x_{\ell_t})$ without reading more than one symbol from each server.

Example 2.1. To demonstrate the differences between the asynchronous and regular batch code models, the following example from [6] could be used:

Consider the systematic $[8, 4, 3]_2$ batch code $\mathcal{C}$ generated by the following matrix:

$$\mathbf{G} = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \end{array} \begin{array}{c} \begin{array}{cccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \end{array} \\ \left[\begin{array}{cccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right] \end{array}$$

The query (f_1, f_1, f_1) can be retrieved from the following disjoint sets of symbols:

Batch (f_1, f_1, f_1)	
f_1	(s_1)
f_1	$(s_2 \oplus s_5)$
f_1	$(s_3 \oplus s_7)$

Assume that the first queried file f_1 has been retrieved, while the last two queries are still being served, and a new query for the file f_2 has arrived. Then, in the asynchronous model, the recovery set $f_2 = (s_4 \oplus s_8)$ could be used to serve the incoming request for the file f_2 immediately after the first queried file f_1 had finished, without affecting the recovery sets of the other two remaining active queries f_1, f_1.

While the example 2.1 demonstrates how the code given by the matrix $\mathbf{G}$ could be used to achieve the same amount of concurrent requests using the asynchronous batch code model as the regular batch code model in the given specific situation, using $\mathbf{G}$ to generate an asynchronous batch code does not result in a $[8, 4, 3]_2$ asynchronous batch code. It can be shown, however, that for any initial selection of the recovery sets, and for any finished query and new incoming query, there is always a way to select disjoint recovery sets for a total of two concurrent requests, meaning that the code $\mathcal{C}$ is an asynchronous $[8, 4, 2]_2$ batch code.

From the example though, the difference between the regular and asynchronous batch code models can be clearly seen. If a storage system using the code $\mathcal{C}$ was operating

under the regular batch code model, the new incoming request for f_2 would have to wait until the two remaining active requests f_1, f_1 also finished serving, and a new batch of three requests could be served. Using the asynchronous model of the code $\mathcal{C}$, however, would reduce the time that f_2 would have to wait by serving it immediately after the first request for f_1 was finished.

2.5 Simplex Codes

Definition of simplex codes, as given in [11]:

Definition 2.8. A $[2^k - 1, k]$ *simplex code* is a linear code of length $n = 2^k - 1$ and dimension k whose $k \times n$ generator matrix $\mathbf{G}$ contains each nonzero column vector $\mathbf{z}$ of length r exactly once as a column.

Example 2.2. Generator matrix $\mathbf{G}$ of a $[7, 3]$ simplex code:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Example 2.3. As it is shown in [6] that the code $\mathcal{C}$ formed by the matrix $\mathbf{G}$ given in example 2.2 is a $[7, 3, 4]_2$ batch code, it is not a $[7, 3, 4]_2$ asynchronous batch code. In [6] it is shown by the following example:

Assume that the query (f_1, f_1, f_1, f_1) was submitted by the users. Then, one copy of f_1 is retrieved from s_1, and for each of the remaining three copies of f_1, at least two symbols of $\mathbf{s}$ have to be used. If the next query f_2 arrives, it is impossible to serve it without accessing one of the servers containing $s_2, ..., s_7$ at least twice. Therefore, $\mathcal{C}$ is not an asynchronous $[7, 3, 4]_2$ batch code.

As this example proves that using simplex codes to construct asynchronous batch codes does not guarantee the same amount of concurrent requests supported by the two models, the *skip distance* parameter, as defined below, is introduced to improve the performance of the asynchronous batch code model.

Definition 2.9. Assume that a storage system has a request queue $\mathbf{X}$ (with $|\mathbf{X}| \geq d$) of requests submitted by users waiting to be served. Then, the *skip distance* d defines the size of the subset $\mathbf{X}' \subseteq \mathbf{X}$ containing the first d requests of $\mathbf{X}$, which will be attempted to be served by the storage system if the first request in the queue $\mathbf{X}$ does not have an available recovery set.

While it was shown in example 2.3 that using the $[7, 3]$ simplex matrix to generate asynchronous and regular batch codes $\mathcal{C}_a$ and $\mathcal{C}_b$ does not yield the same amount of guaranteed concurrent requests for $\mathcal{C}_a$ and $\mathcal{C}_b$ ($t_a \neq t_b$), the main object of study in this

thesis is the comparison of regular and asynchronous batch codes generated by various $[n, k]$ simplex codes.

By introducing the skip distance d, it is shown in simulations that the mean amount of concurrent requests for $\mathcal{C}_a$ will approach the batch size t_a of $\mathcal{C}_b$ for the codes $\mathcal{C}_a$ and $\mathcal{C}_b$ generated by the same $[n, k]$ simplex code matrix $\mathbf{G}$. In addition to this, it is shown that even when not using the addition of skip distance, the asynchronous model will outperform the regular batch code model in most simulated scenarios.

2.6 Statistical Distributions

In this thesis, for the purpose of analyzing differences between the regular and asynchronous batch code models, various statistical distributions were used. Below, definitions and descriptions of statistical distributions are provided, as given in [10]:

Definition 2.10. The set of possible outcomes of a probabilistic experiment is called the *sample*, *event*, or *possibility space*.

Definition 2.11. A *random variable* is a function that maps events defined on a sample space into a set of values.

Definition 2.12. A *variate* is a generalization of the idea of a random variable and has similar probabilistic properties but is defined without reference to a particular type of probabilistic experiment. A variate is the set of all random variables that obey a given probabilistic law.

Definition 2.13. Let $\mathbf{X}$ denote a variate and let $R_{\mathbf{X}}$ be the set of all (real number) values that the variate can take. The set $R_{\mathbf{X}}$ is the *range* of $\mathbf{X}$

Definition 2.14. Let $\mathbf{X} = x$ mean "the value realized by the variate $\mathbf{X}$ is x". Let the *probability statement* $Pr[\mathbf{X} \leq x]$ mean "the probability that the value realized by the variate $\mathbf{X}$ is less than or equal to x".

Definition 2.15. The *distribution function* F (or more specifically $F_{\mathbf{X}}$) associated with a variate $\mathbf{X}$ maps from the range $R_{\mathbf{X}}$ into the probability domain $R_{\mathbf{X}}^{\alpha}$ or $[0, 1]$ and is such that:

$$F(x) = Pr[\mathbf{X} \leq x] = \alpha, x \in R_{\mathbf{X}}, \alpha \in R_{\mathbf{X}}^{\alpha} \tag{1}$$

Definition 2.16. A *probability density function*, $f(x)$, is the first derivative coefficient of a distribution function, $F_{\mathbf{X}}$, with respect to x (where this derivative exists):

$$f(x) = \frac{d(F(x))}{dx} \tag{2}$$

A discrete variate takes discrete values x with finite probabilities $f(x)$. In this case $f(x)$ is the *probability function*, also called the *probability mass function*.

2.6.1 Poisson Distribution

A discrete probability distribution, used, for example to represent the number of arrivals in a specific interval.
Range $0 \leq x < \infty$, where x is an integer
Parameter $\lambda > 0$
The parameter λ is also the mean and the variance of the distribution.
Distribution function $F(x) = \sum_{i=1}^{x} \lambda^i \frac{\exp(-\lambda)}{i!}$
Probability function $f(x) = \lambda^x \frac{\exp(-\lambda)}{x!}$

2.6.2 Exponential Distribution

A continuous distribution, used, for example in queue theory.
Range $0 \leq x < \infty$
Scale parameter $b > 0$
Alternative parameter $\lambda = \frac{1}{b}$
The parameter b is also the mean of the distribution.
Distribution function $F(x) = 1 - \exp(\frac{-x}{b})$
Probability distribution function $f(x) = \frac{1}{b} \exp(\frac{-x}{b}) = \lambda \exp(-\lambda x)$

2.6.3 Normal (Gaussian) Distribution

A continuous distribution.
Range $-\infty < x < \infty$
Location parameter μ
Scale parameter $\sigma > 0$
The parameter μ is also the mean, and σ the standard deviation of the distribution
Probability density function $f(x) = \frac{\sqrt{2\pi\sigma}}{\exp(\frac{-(x-\mu)^2}{2\sigma^2})}$

3 Implementation

For analyzing the differences in performance between the asynchronous and regular batch code models, a system prototype was built which was used to run simulations of the code models under varying setups. The simulations were run with millisecond accuracy to provide results as accurate as possible while keeping the option of running multiple simultaneous simulations. The general process of running simulations can be seen in figures 1 and 2 depicting storage systems using both the regular, and asynchronous batch code models defined by the $[7, 3]$ simplex matrix $\mathbf{G}$ given in example 2.2.

While the main objective of the system prototype was to evaluate performance of systems based on regular and asynchronous batch codes, yet this prototype can be used for testing systems employing a wide range of codes. For example, to simulate a replication based storage system, where $k = 3$ files are stored on a total of $n = 6$ servers, with a total of 2 servers storing each file f_1, f_2, f_3, the matrix given in the following example could be used:

Example 3.1. Matrix used to generate a code emulating a replication based storage system with $k = 3$ files and $n = 6$ servers

$$\mathbf{G} = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{cccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

In figures 1 and 2 the process of running simulations of both batch code models can be seen. Both storage systems in the figures are defined by the $[7, 3]$ simplex matrix, resulting in 7 servers distributing 3 files as shown in the figures. The differences in running simulations of the two batch code models can be seen by the regular batch code model in fig. 1 using the batch table (given in table 2) to assign servers to a batch of $t = 4$ requests each time a previous batch is finished serving, and the asynchronous model in fig. 2 using recovery sets (table 1) to assign servers to any request that can be served from the first d requests in the queue as soon as a request is finished serving. In both figures 1 and 2, the index i of each request x_i in the request queue $\mathbf{X}$ represents the index of the requested file f_i.

In this section, key algorithms and the overall design of the system prototype are presented by providing code examples, accompanied by more detailed explanations of the design decisions.

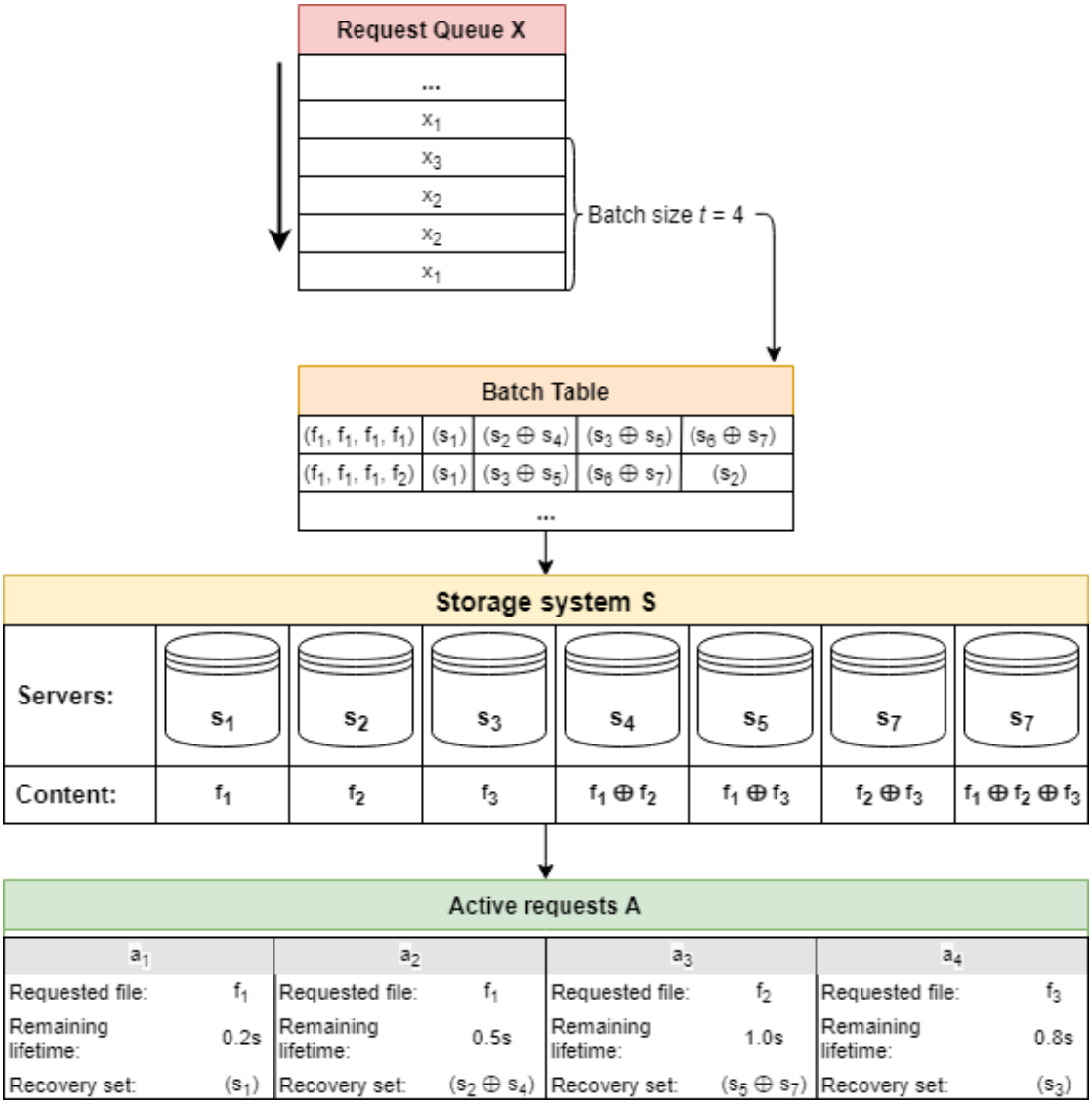


Figure 1. System overview for the regular batch code model

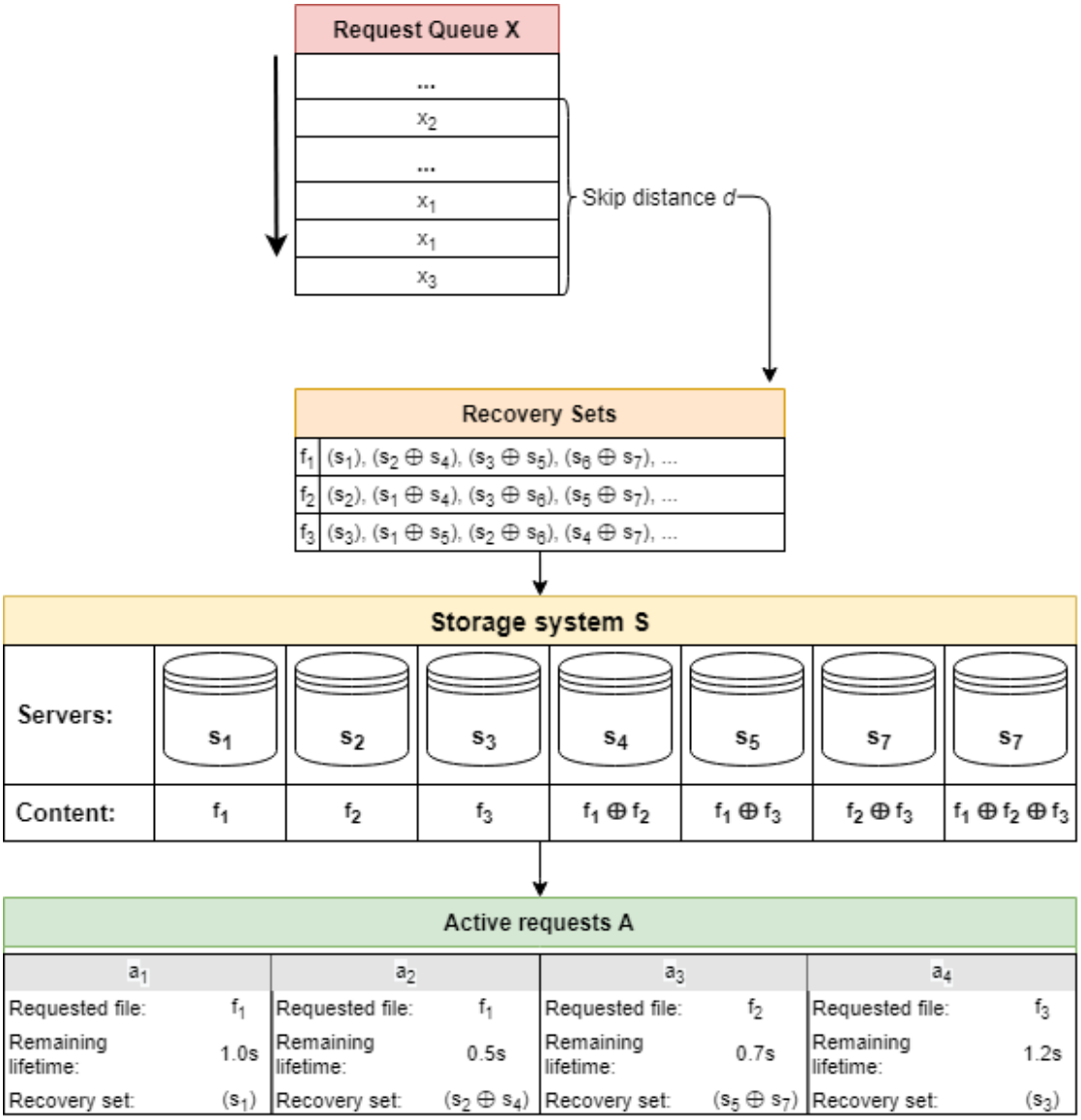


Figure 2. System overview for the asynchronous batch code model

3.1 Codes

The system models (hereafter referred to as codes) used to run the simulations were defined by the following key variables:

1. matrix
2. server combinations
3. batch table
4. request lifetime distribution
5. request gap scales
6. server capacity
7. skip distance

3.1.1 Matrix

A generator matrix **G** was used to define the amount of servers and files for running the simulations, as well as the files or the combinations of files that any server would serve. A $k \times n$ matrix would result in a code, in which n servers were used to host a total of k files. For each server, the corresponding column in the matrix would define what data is stored in the corresponding server. For example, a server server corresponding to the column $(1, 0, 0)^\top$ would only contain f_1, whereas the column $(1, 1, 0)^\top$ would result in the server containing $f_1 \oplus f_2$ (the bitwise XOR of f_1 and f_2).

3.1.2 Recovery Sets

In order to run the simulations, it was necessary to calculate all the possible recovery sets (combinations of servers) that could be used to retrieve any of the k files. For a code defined by a $k \times n$ matrix **G**, this would be done using the following algorithm:

```
# Setup:
servers = [x for x in range(n)]
files = [x for x in range(k)]

files_as_integers = { }                                          #(1)
for f in files:
  file_binary = [0] * k
  file_binary[f] = 1
  files_as_integers[f] = binary_to_int(file_binary)

servers_as_integers = { }                                        #(2)
for s in servers:
  server_binary = G.column(s)
  servers_as_integers[s] = binary_to_int(server_binary)
```

```
combinations = { }
for depth in [x for x in range(k+1)]:                                    #(3)
  for f in files:
    combinations[f].append(recursive_search(f, depth, combinations, servers,
        files_as_integers, servers_as_integers, [ ], 0))

# The recursive function to find combinations up to a depth limit        #(4)
def recursive_search(f, depth_limit, existing_combinations, servers, files_as_integers,
     servers_as_integers, used_servers, current_state):
  for combination in existing_combinations:
    if combination.issubset(used_servers):
      return [ ]

  if current_state == files_as_integers[f]:
    return [used_servers]

  combinations = [ ]
  if depth_limit != len(used_servers):
    for s in servers:
      new_state = XOR(current_state, servers_as_integers[s])
      new_servers = servers.copy()
      new_servers.remove(s)
      new_used_servers = used_servers + [s]
      combinations += recursive_search(f, depth_limit, existing_combinations,
          new_servers, files_as_integers, servers_as_integers, new_used_servers,
          new_state)

  else:
    return [ ]
  return combinations
```

(1–2) As the total number on recursive function calls is relatively high, it is important to make each call as efficient as possible. For this reason, the files and servers are associated with unique integer numbers to make comparisons between the recursion states and calculations of new states more efficient. For this reason, each file f_i for $i \in [1, k]$ is represented as the integer value of a binary string where only the i-th bit is equal to 1. Similarly, each server s_j with $j \in [1, n]$ is represented as the integer value of a binary string equal to the j-th column of the code-defining generator matrix $\mathbf{G}$.

(3) To minimize the maximum recursion depth, where the depth can take any value in the range $[1, k]$. This can be done as any set of $k + 1$ columns of the matrix $\mathbf{G}$ would be linearly dependent and thus makes generating server combinations of size larger than k redundant.

The recursion depth needs to be iterated over an increasing depth limit because it is necessary to always find the smallest possible combinations of servers capable of serving any file f. As the recursive function terminates the recursion branch when a smaller subset of its current server combinations is already a recognized server combination for finding the given file, it is essential that the search for the combinations is conducted breadth-first and in increasing size.

(4) The recursive function itself is a rather basic loop over all possible server combinations, limited by the depth limit which is iterated over in the main algorithm. The function checks whether its given server combination has a smaller subset already defined as a viable server combination for its given file and terminates if true, returns the given server combination as a viable combination if its given state matches the given file, or continues the recursion for all possible remaining servers if the depth limit has not been reached.

For each of the depth limits iterated over in the main algorithm, the recursion starts from the beginning, while terminating any of the recursion branches as soon as they prove to be useless. While it would be possible to save the current states of the recursion when the function terminates due to the depth limit being reached, so that on the next iteration, when the depth limit is increased, could be resumed without repeating any of the steps performed by the recursive function in a previous iteration, this proved to be very expensive in its memory usage.

Example 3.2. Generation of recovery sets for a batch code defined by the matrix **G** of a $[7, 3]$ simplex code:

To start off, both the files and servers need to be assigned integer values. This is done as described above in the algorithm description (3.1.2) steps 1 and 2. The resulting integer representations are:

f_1	f_2	f_3
4	2	1

s_1	s_2	s_3	s_4	s_5	s_6	s_7
4	2	1	6	5	3	7

To find all the server combinations capable of serving the files f_1, f_2, f_3, sets of size 1 (single servers) are looked at first. This is done simply by comparing the integer values of the servers s_i to the values assigned to files f_j and if the values match, the s_i is added as a valid server combination for f_j for all $i \in [1, 7]$ and $j \in 1, 2, 3$:

Size 1	
f_1	(s_1)
f_2	(s_2)
f_3	(s_3)

Next, to find server combinations of size 2, for example, for f_1, all possible pairs of servers that do not include s_1 are tried. Pairs including s_1 are excluded as there already exists a smaller subset (s_1 by itself) capable of serving f_1. Bitwise *XOR* is then applied to the integer values of all the other server pairs and if the resulting value equals the

integer value of f_1, the pair is added as a valid server combination for f_1. This search is then repeated for f_2 and f_3, resulting in the following server combinations being added to the set of valid combinations:

	Size 2
f_1	$(s_2 \oplus s_4), (s_3 \oplus s_5), (s_6 \oplus s_7)$
f_2	$(s_1 \oplus s_4), (s_3 \oplus s_6), (s_5 \oplus s_7)$
f_3	$(s_1 \oplus s_5), (s_2 \oplus s_6), (s_4 \oplus s_7)$

Finally, combinations of size 3 are searched for and added in a similar fashion to what was done previously for sets of size 2, resulting in the complete server combination table for the $[7, 3]$ simplex code given in table 1

	Recovery sets
f_1	$(s_1), (s_2 \oplus s_4), (s_3 \oplus s_5), (s_6 \oplus s_7), (s_2 \oplus s_3 \oplus s_7), (s_2 \oplus s_5 \oplus s_6), (s_3 \oplus s_4 \oplus s_6), (s_4 \oplus s_5 \oplus s_7)$
f_2	$(s_2), (s_1 \oplus s_4), (s_3 \oplus s_6), (s_5 \oplus s_7), (s_1 \oplus s_3 \oplus s_7), (s_1 \oplus s_5 \oplus s_6), (s_3 \oplus s_4 \oplus s_5), (s_4 \oplus s_6 \oplus s_7)$
f_3	$(s_3), (s_1 \oplus s_5), (s_2 \oplus s_6), (s_4 \oplus s_7), (s_1 \oplus s_2 \oplus s_7), (s_1 \oplus s_4 \oplus s_6), (s_2 \oplus s_4 \oplus s_5), (s_5 \oplus s_6 \oplus s_7)$

Table 1. Recovery sets for the $[7, 3]$ simplex code

3.1.3 Batch Table

For running codes as regular batch codes, the most efficient solution is to generate all possible unique combinations of t requests, where t is the batch size of the code. For a $k \times n$ simplex code, the batch size t is given as follows [7]:

$$t = 2^{k-1}$$

The batch table is used to define which servers are used to serve each of the incoming requests when simulating regular batch codes. The table consists of all unique sets of length t of k files and the server combinations used to serve each file in the request batch.

For generating the batch table, the following algorithm was used:

```
t = 2**(k-1)
unique_sets = combinations_with_replacement([x for x in range(k)], t)   #(1)
server_combinations = code.server_combinations                          #(2)
batch_table = { }

for uset in unique_sets:
  success = False
  while not success:
    used_servers = [ ]
    combinations = [ ]

    for f in uset:
      combination = get_server_combination(used_servers, f,  server_combinations)
      if combination:
        for s in combination:
```

```
          used_servers += [s]
          combinations += [combination]
      else:
        shuffle(set)                                                    #(3)

    if len(combinations) == k:
      success = True
      batch_table[set] = combinations

def get_server_combination(used_servers, f,  server_combinations):
  for combination in server_combinations[f]:
    valid = True
    for s in combination:
      if s in used_servers:
        valid = False
    if valid:
      return combination
    else:
      return False
```

1. To generate all possible unique sets of length t for k files, combinations with replacement from the Python itertools library, as given in [8], was used.

2. Server combinations is the dictionary of all the possible recovery sets able to serve any of the k files, as given by the algorithm defined in section 3.1.2.

3. As the sets generated by combinations with replacement are emitted in lexicographic sort order, it can happen that by trying to assign servers to files of the requests in the sorted order, using the first unused server combination for the files, as given by the algorithm defined in section 3.1.2, a successful combination for the entire batch cannot be found. If this happens, shuffling the order of the batch and retrying until a successful order is found, proved to be a much more efficient approach than using a recursive function to find a successful combination without shuffling.

Example 3.3. Batch table generation for a batch code defined by the matrix $\mathbf{G}$ of a $[7, 3]$ simplex code, using the recovery sets from example 3.2.

First of all, the batch size k must be calculated:

$$t = 2^{k-1} = 2^2 = 4$$

Next, all unique sets of size $t = 4$ of the files f_1, f_2, f_3 are found, of which there is a total of 15. Then, for each of the sets, servers are assigned based on the server combinations. For example, the batch (f_1, f_1, f_2, f_3) is assigned the following servers:

Batch (f_1, f_1, f_2, f_3)	
f_1	(s_1)
f_1	$(s_3 \oplus s_5)$
f_2	(s_2)
f_3	$(s_4 \oplus s_7)$

When assigning server combinations to the files in a batch, the combinations are chosen in the order that they were defined in the *recovery sets* table 1. While this means that smaller sets are prioritized, it does not take into account the servers of the files yet to be assigned, meaning that not all sets can be assigned valid server combinations using this method of selecting combinations for the files. This can be seen using the batch (f_1, f_1, f_1, f_2):

Batch (f_1, f_1, f_1, f_2)	
f_1	(s_1)
f_1	$(s_2 \oplus s_4)$
f_1	$(s_3 \oplus s_5)$
f_2	$None$

This problem was solved by randomizing the order (shuffling) of the batch, as it proved to be a much more efficient approach than "intelligently" assigning server combinations to files in a batch, by taking into account all the files that follow in the batch, especially when generating batch tables for codes defined by larger simplex codes, for example the $[31, 5]$ simplex code. For an example of the shuffling, the batch (f_1, f_1, f_1, f_2) can be reordered as (f_2, f_1, f_1, f_1), which yields the following server combination assignment using the same method as before:

Batch (f_2, f_1, f_1, f_1)	
f_2	(s_2)
f_1	(s_1)
f_1	$(s_3 \oplus s_5)$
f_1	$(s_6 \oplus s_7)$

This method of assigning server combinations is repeated for all the 15 unique batches, resulting in the batch table given in table 2.

Example 3.4. The exact sizes of the recovery set and batch tables for various batch codes defined by $[n, k]$ simplex codes can be seen in the table 3.

$[n, k]$	Total recovery sets	Sets per file	Batch size	Unique batches
$[3, 2]$	4	2	2	3
$[7, 3]$	24	8	4	15
$[15, 4]$	368	92	8	165
$[31, 5]$	18420	3684	16	4845

Table 3. Sizes of batch and recovery set tables for $[n, k]$ simplex codes

Batch table for the $[7, 3]$ simplex code				
(f_i, f_j, f_k, f_l)	f_i	f_j	f_k	f_l
(f_1, f_1, f_1, f_1)	(s_1)	$(s_2 \oplus s_4)$	$(s_3 \oplus s_5)$	$(s_6 \oplus s_7)$
(f_1, f_1, f_1, f_2)	(s_1)	$(s_3 \oplus s_5)$	$(s_6 \oplus s_7)$	(s_2)
(f_1, f_1, f_1, f_3)	(s_1)	$(s_2 \oplus s_4)$	$(s_6 \oplus s_7)$	(s_3)
(f_1, f_1, f_2, f_2)	(s_1)	$(s_2 \oplus s_4)$	$(s_3 \oplus s_6)$	$(s_5 \oplus s_7)$
(f_1, f_1, f_2, f_3)	(s_1)	$(s_2 \oplus s_4)$	$(s_5 \oplus s_7)$	(s_3)
(f_1, f_1, f_3, f_3)	(s_1)	$(s_2 \oplus s_4)$	(s_3)	$(s_6 \oplus s_5 \oplus s_7)$
(f_1, f_2, f_2, f_2)	(s_1)	(s_2)	$(s_3 \oplus s_6)$	$(s_5 \oplus s_7)$
(f_1, f_2, f_2, f_3)	(s_1)	(s_2)	$(s_3 \oplus s_6)$	$(s_4 \oplus s_7)$
(f_1, f_2, f_3, f_3)	(s_1)	(s_2)	(s_3)	$(s_4 \oplus s_7)$
(f_1, f_3, f_3, f_3)	(s_1)	(s_3)	$(s_2 \oplus s_6)$	$(s_4 \oplus s_7)$
(f_2, f_2, f_2, f_2)	(s_2)	$(s_1 \oplus s_4)$	$(s_3 \oplus s_6)$	$(s_5 \oplus s_7)$
(f_2, f_2, f_2, f_3)	(s_2)	$(s_1 \oplus s_4)$	$(s_5 \oplus s_7)$	(s_3)
(f_2, f_2, f_3, f_3)	(s_2)	$(s_1 \oplus s_4)$	(s_3)	$(s_6 \oplus s_5 \oplus s_7)$
(f_2, f_3, f_3, f_3)	(s_2)	(s_3)	$(s_1 \oplus s_5)$	$(s_4 \oplus s_7)$
(f_3, f_3, f_3, f_3)	(s_3)	$(s_1 \oplus s_5)$	$(s_2 \oplus s_6)$	$(s_4 \oplus s_7)$

Table 2. Batch table for the $[7, 3]$ simplex code

3.1.4 Request Lifetime and Gap Distributions

One of the key differences studied when comparing the asynchronous and regular batch code models was how they handled requests of varying durations. For this reason, a defining parameter for running a code was the distribution of the lifetime (duration) of the requests. The system prototype supports the use of the following statistical distributions for defining the lifetimes:

- uniform(min, max)
- exponential(b)
- Poisson(λ)
- normal(μ, σ) (Gaussian distribution)

The distributions used were imported from the numpy.random library of Python, described in [9].

To determine the time between incoming requests, gap scales $(b_{f_1}, b_{f_2}, ..., b_{f_k})$ were used. Each file f_i for $i \in [1, k]$ in the code simulation was assigned an exponential distribution variate b (scale) b_{f_i}, using which a gap duration was generated after each incoming request for the requested file f_i. The duration of the gap determined how long

it would take for another request for the file f_i to arrive. This, in turn, determined the total incoming rate $r_{inc_{f_i}}$ for each file in the simulation, with a mean rate $r_{inc_{f_i}} = \frac{1}{b_{f_i}}$ for each file f_i for $i \in [1, k]$ and a total incoming request rate $r_{inc_t} = \sum_{i=1}^{k} r_{inc_{f_i}}$

3.2 Requests

The incoming requests of the simulations were defined by Python dictionaries, which contained the following:

Requested file defined by an integer $f \in [1, k]$

Lifetime duration of the request, generated using one of the distributions listed in section 3.1.4

Assigned servers an array containing the recovery set used to serve the request, assigned when the request is added to active requests from the queue

Time in queue time the request has spent in the queue, used for statistics when analyzing code performance

3.3 Running Simulations

For running the simulations, each code object has a number of variables which are used to store information necessary for the simulation:

Request queue a deque object from the collections Python standard library

Current requests an array of active requests

Completed requests a counter keeping track of the total number of requests completed during the simulation

Servers a dictionary of servers and their current load

Remaining gap times a dictionary of files and the remaining time until a new request for any of the files will be generated

Running time a counter used to keep track of how long the simulation has been running

Statistics a collection of data collected about the performance of the simulated code:

- request queue size

- amount of active requests
- completed requests
- requests per second (mean)
- servers in use
- mean queue time
- maximum queue time

The simulations of running codes were performed in 1 millisecond steps. In each step, the following operations are performed:

1. Add 1ms to code running time
2. Terminate any request which has been active for its predetermined duration (life-time)
3. If able, add new requests from the request queue to active requests
4. Calculate statistics for the code and write them to file
5. Subtract 1ms from remaining gap times for each file and if the request gap time for any file was reached, generate a new request

3.4 System Prototype and Methods for Analysis

The system prototype designed to run the simulations offers information about the current state of the running simulations, such as graphs displaying any of the collected statistics for all the running codes for the last minute, specific statistics values for the running code simulations, as well as information about the request queue, active requests and state of the servers in the simulation. These functionalities were implemented to provide an overview of the testing process and to quickly test hypothesis of how the defining variables affect running simulations.

Further analysis of the system model was performed using the collected statistical data. For achieving this, an automated process of testing large numbers of codes with a specific range of defining variables based on the test was set up. These automated processes were repeated until a large enough sample size, sufficient to provide a meaningful overview of the specific model under the conditions being tested, was achieved.

Example 3.5. To provide an overview of the complete process of running the simulations, assume that an asynchronous code $\mathcal{C}_a$ and a regular batch code $\mathcal{C}_b$ are being simulated. Both $\mathcal{C}_a$ and $\mathcal{C}_b$ are defined by the matrix $\mathbf{G}$ of a $[7, 3]$ simplex code:

$$\mathbf{G} = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

While the regular batch code $\mathcal{C}_b$ is using the batch table, as given in table 2, to assign server combinations to incoming requests, the asynchronous code $\mathcal{C}_a$ only uses the recovery sets as given in example 3.2.

Assume that both of the codes $\mathcal{C}_a$ and $\mathcal{C}_b$ have the request lifetimes defined by the exponential distribution variate $b = 1$, meaning that the average request lifetime is one second, and gap scales $(b_{f_1}, b_{f_2}, b_{f_3}) = (0.75, 0.75, 0.75)$. These gap scales result in an incoming rate $r_{inc_{f_i}} = \frac{1}{b_{f_i}} = \frac{1}{0.75} = 1\frac{1}{3}$ for each file f_i for $i \in 1, 2, 3$ and a total incoming request rate $r_{inc_{total}} = \sum_{i=1}^{3} r_{inc_{f_i}} = 4$. As the gap times are defined generated using the exponential distribution, the total number of incoming requests per second will follow the Poisson distribution with the variate $\lambda = r_{inc_t} = 4$.

In addition to the code-defining parameters above, the system simulation of the asynchronous code $\mathcal{C}_a$ has a skip distance parameter set as $d = 2$.

For simplicity, assume that both $\mathcal{C}_a$ and $\mathcal{C}_b$ have matching incoming request queues $\mathbf{X}_a = \mathbf{X}_b = [x_1, x_2, x_3, x_4]$ where the incoming request objects x_i have the following values:

Request queues $\mathbf{X}_a$ and $\mathbf{X}_b$		
x_i	Requested file	Lifetime
x_1	f_2	$0.5s$
x_2	f_1	$1.2s$
x_3	f_1	$1.0s$
x_4	f_1	$0.9s$

Also, for simplicity, assume that the codes $\mathcal{C}_a$ and $\mathcal{C}_b$ have matching values for their remaining gap times $(g_{f_1}, g_{f_2}, g_{f_3}) = (1.1s, 0.8s, 1.2s)$, and have matching active requests $\mathbf{A}_a = \mathbf{A}_b = [a_1, a_2, a_3, a_4]$ such that:

Active requests $\mathbf{A}_a$ and $\mathbf{A}_b$			
a_i	Requested file	Remaining lifetime	Recovery set
a_1	f_1	$0.5s$	(s_1)
a_2	f_1	$0.7s$	$(s_2 \oplus s_4)$
a_3	f_1	$1.0s$	$(s_3 \oplus s_5)$
a_4	f_1	$1.0s$	$(s_6 \oplus s_7)$

After $0.5s$ of the simulation has passed, the request a_1 is finished serving, meaning that the server s_1 is no longer in use. While the regular batch code $\mathcal{C}_b$ takes no action here, the simulation of $\mathcal{C}_a$ attempts to serve the first request in the queue $\mathbf{X}_a$. As the request x_1 can not be served, as there is no available recovery set for f_2, a subset $\mathbf{X}_a^{'}$ of size d of the first $d = 2$ requests in the request queue $\mathbf{X}_a$ is searched for a servable request. The second request x_2 is looked at and, as its requested file f_1 can be served, it is added to the list of active requests $\mathbf{A}_a$.

After another $0.2s$ of the simulation pass, the request a_2 is completed, freeing up the servers s_2 and s_4. While the code $\mathcal{C}_b$ still takes no action, $\mathcal{C}_a$ adds the request x_1 for f_2, which was skipped over during the previous attempt, to $\mathbf{A}_a$, as the server s_2 is a possible recovery set for f_2.

A further $0.1s$ into the simulation, the gap times $(g_{f_1}, g_{f_2}, g_{f_3}) = (0.3s, 0.0s, 0.4s)$. As the gap time $g_{f_2} = 0$, a new request x_5 for f_3, with a lifetime of, for example $1.6s$, is added to the request queues and a new gap time is generated (using the exponential variable b_{f_2}) for g_{f_2}, equal to $1.0s$. At this point, the states of the codes are as follows:

Request queue $\mathbf{X}_a$		
x_i	Requested file	Lifetime
x_3	f_1	$1.0s$
x_4	f_1	$0.9s$
x_5	f_3	$1.6s$

Request queue $\mathbf{X}_b$		
x_i	Requested file	Lifetime
x_1	f_2	$0.5s$
x_2	f_1	$1.2s$
x_3	f_1	$1.0s$
x_4	f_1	$0.9s$
x_5	f_3	$1.6s$

Active requests $\mathbf{A}_a$			
a_i	Requested file	Remaining lifetime	Recovery set
a_3	f_1	$0.2s$	$(s_3 \oplus s_5)$
a_4	f_1	$0.2s$	$(s_6 \oplus s_7)$
a_5	f_1	$0.9s$	(s_1)
a_6	f_2	$0.4s$	(s_2)

Active requests $\mathbf{A}_b$			
a_i	Requested file	Remaining lifetime	Recovery set
a_3	f_1	$0.2s$	$(s_3 \oplus s_5)$
a_4	f_1	$0.2s$	$(s_6 \oplus s_7)$

After another $0.2s$ of the simulations has passed, the requests a_3 and a_4 are complete. As now $|\mathbf{A}_b| = 0$ (the entire batch has been served), the regular batch code $\mathcal{C}_b$ creates

a new batch of four requests from the request queue $\mathbf{X}_b$. This results in a batch of requests for the files (f_2, f_1, f_1, f_1), which is then sorted by increasing indices of the requested files and for which the recovery sets are taken from the batch table (given in table 2). As a result, the batch is served using the recovery sets for each of the files f_i: $f_1 = (s_1), f_1 = (s_3 \oplus s_5), f_1 = (s_6 \oplus s_7), f_2 = (s_2)$.

At the same time, the code $\mathcal{C}_a$ also adds the requests x_3 and x_4 from $\mathbf{X}_a$ to $\mathbf{A}_a$, resulting in the following final state of this example:

Request queue $\mathbf{X}_a$		
x_i	Requested file	Lifetime
x_5	f_3	$1.6s$

Request queue $\mathbf{X}_b$		
x_i	Requested file	Lifetime
x_5	f_3	$1.6s$

Active requests $\mathbf{A}_a$			
a_i	Requested file	Remaining lifetime	Recovery set
a_5	f_1	$0.7s$	(s_1)
a_6	f_2	$0.2s$	(s_2)
a_7	f_1	$1.0s$	$(s_3 \oplus s_5)$
a_8	f_1	$0.9s$	$(s_6 \oplus s_7)$

Active requests $\mathbf{A}_b$			
a_i	Requested file	Remaining lifetime	Recovery set
a_5	f_1	$1.2s$	(s_1)
a_6	f_1	$1.0s$	$(s_3 \oplus s_5)$
a_7	f_1	$0.9s$	$(s_6 \oplus s_7)$
a_8	f_2	$0.5s$	(s_2)

4 Results

To provide a comprehensive overview of differences in performance of the regular and asynchronous batch code models, simulations were run using a wide range of defining variables for the codes described in section 3.1. Throughout this section, the following notation is used to represent the statistics collected from the simulations:

Definition 4.1. The *service rate* r_s is the amount of requests a system model serves in one second and is calculated as the mean amount of requests completed by the system each second.

Definition 4.2. *Concurrent requests* c_r is the amount of active requests being served by a system model, calculated as the mean amount of active requests for each (one millisecond) step of the simulation.

Definition 4.3. *Queue time* t_q is the mean time a request spends in queue, waiting to be served by a system model. During each step i of a simulation, a (local) mean queue time t_{q_i} is calculated from the time spent in queue by each request during the step i of the simulation. The (global) *queue time* t_q is the mean of all t_{q_i}.

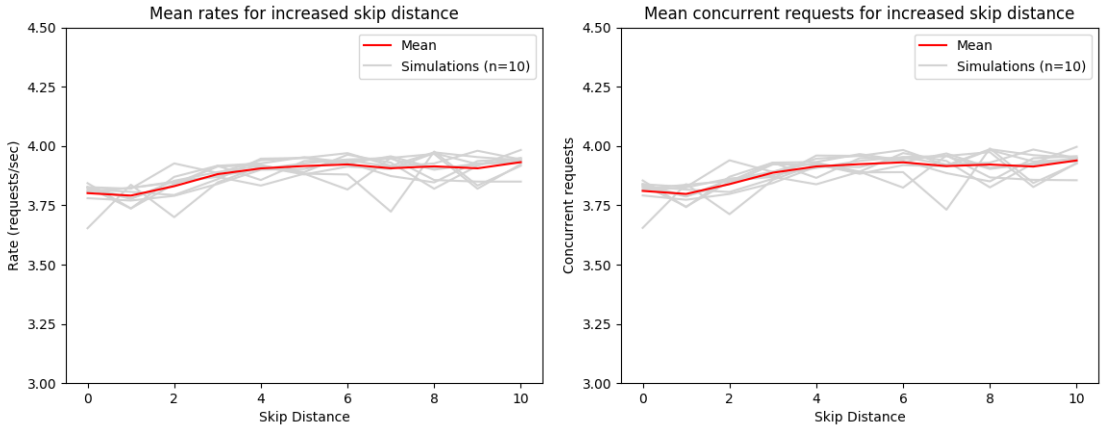
4.1 Skip Distance

In this section, the asynchronous model is analyzed in terms of its ability to serve an incoming request for any file as soon as an active request is finished. In the theoretical model this is stated to be the defining property of the model. As became apparent when generating the batch tables for regular batch codes defined by simplex codes in section 3.1.3, there are some orderings of batches, when assigned servers using server combinations in the order of increasing size, for which a valid set of server combinations could not be found. In section 3.1.3, the solution proposed for finding a valid combination was to shuffle the order of the batch and retry the server assignment.

As a similar issue of blocking request configurations can happen when running codes in the asynchronous model, the skip distance parameter was introduced. Skip distance d is used to define the depth of the queue up to which servable requests are searched for when trying to add new active requests during a code step.

To demonstrate the effects of an increased skip distance d, the following simulations for the asynchronous code $\mathcal{C}_a$ defined by the $[7, 3]$ simplex matrix were made. The simulations used a constant lifetime $l = 1$ for all the requests, uniformly distributed amounts of incoming files f_1, f_2, f_3 defined by the incoming request gap times $(g_{f_1}, g_{f_2}, g_{f_3}) = (0.75s, 0.75s, 0.75s)$, resulting in a total incoming request rate $r_{inc_{total}} = 4$. A total of 10 simulations were performed for each skip distance $d \in 0, 1, ..., 10$ using these parameters, results of which are visualized in *gray* on the figures 3a and 3b, and the mean values of

the simulations for the collected statistics, visualized in *red*, were calculated. Each of the simulations had a duration of 5 minutes.



(a) Service rate r_s (b) Concurrent requests c_r

Figure 3. Mean service rate r_s and mean concurrent requests c_r using constant lifetime with increased skip distance d

The mean values for the service rate r_s, concurrent requests c_r and request queue times t_q can be seen in table 4.

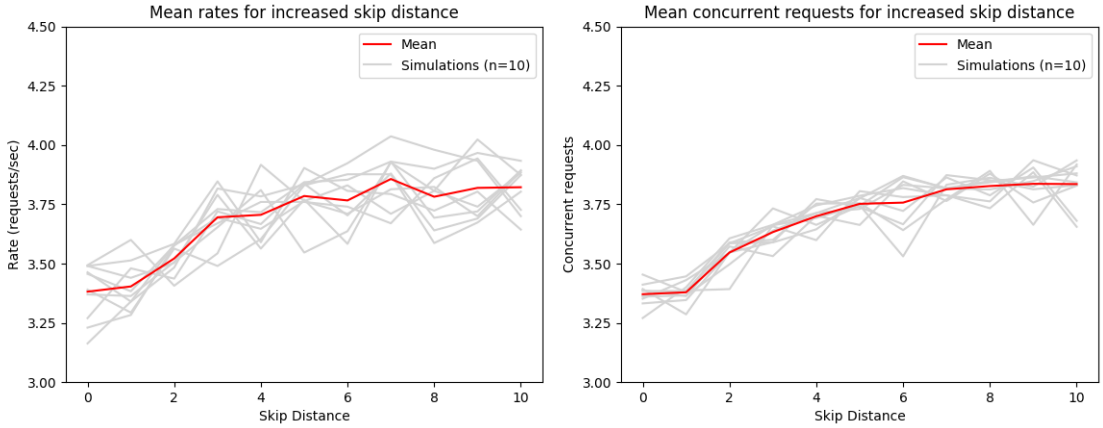
d	r_s	c_r	t_q
0	3.801	3.811	5.118
1	3.791	3.798	4.627
2	3.831	3.839	3.364
3	3.881	3.888	3.062
4	3.906	3.913	3.329
5	3.916	3.924	4.150
6	3.922	3.931	2.980
7	3.906	3.915	2.897
8	3.914	3.922	2.477
9	3.906	3.913	2.874
10	3.932	3.939	3.389

Table 4. Mean r_s, c_r and t_q for increased skip distance d using constant request lifetime

As can be seen from fig. 3 and table 4, using constant request lifetimes when running the simulations, the addition of a skip distance d does provide a small increase in the

performance of the code in terms of mean service rate r_s, mean concurrent requests c_r and the mean queue time t_s for the requests.

To more clearly demonstrate the increase in performance, simulations were performed again for the asynchronous code $\mathcal{C}_a$ defined by the $[7, 3]$ simplex matrix with $r_{inc_{total}} = 4$ and incoming request gap scales $(g_{f_1}, g_{f_2}, g_{f_3}) = (0.75s, 0.75s, 0.75s)$. Instead of using a constant lifetime for the incoming requests, as in the simulations of fig. 3, request lifetimes were determined by the exponential distribution using $b = 1$. A total of 10 simulations with a duration of 5 minutes each were performed using skip distances $d \in 0, 1, ..., 10$. The results of these simulations are given in fig. 4 and table 5.



(a) Service rate r_s

(b) Concurrent requests c_r

Figure 4. Mean service rate r_s and mean concurrent requests c_r using exponential lifetime with increased skip distance d

The mean values for the service rate r_s, concurrent requests c_r and request queue times t_q for the simulations from fig. 4 can be seen in table 5.

d	r_s	c_r	t_q
0	3.382	3.371	12.385
1	3.404	3.379	12.852
2	3.522	3.547	9.340
3	3.695	3.633	6.579
4	3.706	3.670	5.834
5	3.785	3.752	4.498
6	3.766	3.757	4.434
7	3.857	3.813	4.275
8	3.782	3.827	4.950
9	3.819	3.836	4.772
10	3.822	3.835	4.811

Table 5. Mean r_s, c_r and t_q for increased skip distance d with exponential request lifetime

The effect of using a different distribution to generate incoming request lifetimes is clear when comparing the graphs of fig. 3 and 4. While the graphs in figures 3a and 3b using a constant request lifetime $l = 1$ are nearly identical, the increased volatility caused by the large variance of the exponential distribution is clearly reflected in the service rates visualized in fig. 4a.

From the results of the second round of simulations (fig. 4), the positive effects of increased skip distance d can more easily be seen. While using constant lifetimes of requests, adding a skip distance $d = 10$ only offered a 3.4% increase in the service rate r_s compared to a skip distance of $d = 0$. In the second round of simulations, simulating exponential request lifetimes with $b = 1$, skip distance $d = 10$ resulted in a 13.0% increase in the mean service rate compared to $d = 0$.

The difference in increased performance provided by the skip distance d in the two rounds of simulations (fig. 3 and 4) is likely caused by the beneficial effects of using constant request lifetimes in first round of simulations. To explain this, simulations using both incoming request types were observed. If request lifetimes are set to $l = 1$ second, the served requests naturally tend to form batches, and as a result, are finished serving simultaneously. Then, when assigning recovery sets to new requests, smaller recovery sets can be used, resulting in higher amounts of concurrent requests. In comparison, when using exponential lifetimes, as in fig. 4, requests are finished serving at different times, resulting in higher chances of recovery sets of size three being assigned to new requests. As this is the case, the probability of blocking incoming request configurations increases, which leads to situations where the amount of concurrent requests drops to two. In comparison, with constant request lifetimes, the minimum amount of concurrent requests observed was three.

As the addition of skip distance d offered a clear improvement to the performance of

the asynchronous model in terms of the service rate r_s and concurrent requests c_r, all of the following simulations of asynchronous batch codes used a skip distance of $d = 8$. This value was chosen, as from the results of this section, it was apparent that $d = 8$ was sufficient to provide an increase in service rates for the asynchronous model, while requiring low enough simulation time.

4.2 Constant Request Lifetime

The following simulations were run with request lifetimes $l = 1$ second, with no variance, for both the regular and asynchronous batch code models. Under these conditions, every request in a batch is finished serving at the same time, resulting in no waiting for requests with longer duration to be finished until a new batch can be served for the regular batch code model. In table 6, the mean results of 10 simulations for each code, with a duration of 5 minutes, can be seen. Each code in the simulations uses a constant lifetime $l = 1$ of the requests. All of the simulations have uniformly distributed rates of incoming files $f_1, ..., f_k$, with a total incoming request rate $r_{inc_{total}} = t$, where $t = 2^{k-1}$ is the batch size of a regular batch code $\mathcal{C}_b$ generated by the $[n, k]$ generator matrix of a simplex code, resulting in gap scales $g_{f_i} = \frac{1}{t}k$ for $i \in 1, ..., k$.

The simulations were performed for both the regular and asynchronous batch code models using 3 different simplex code matrices to generate the codes. Simulations for the asynchronous models also used a skip distance property of $d = 8$, as described in section 4.1.

The mean values for the service rate r_s, concurrent requests c_r and request queue times t_q for each $[n, k]$ matrix used to generate the codes can be seen in table 6.

	Regular batch codes			Asynchronous batch codes		
$[n, k]$	r_s	c_r	t_q	r_s	c_r	t_q
$[7, 3]$	3.879	3.887	1.892	3.918	3.924	3.291
$[15, 4]$	7.896	7.907	2.491	7.742	7.760	4.657
$[31, 5]$	15.840	15.859	1.826	14.944	14.978	6.230

Table 6. Mean r_s, c_r and t_q for regular and asynchronous batch codes using constant request lifetimes

While the asynchronous batch code model does outperform the regular batch code when using the $[7, 3]$ simplex code matrix to generate the codes in these simulations, this is likely due to the high variance of gap times between incoming requests, which follow the exponential distribution using the gap scales $(g_{f_1}, g_{f_2}, g_{f_3})$. While the exponential distribution is the most realistic distribution to use when simulating arrival gaps between requests, it results in occasions where the simulated regular batch code has to wait for additional requests to arrive before it can start serving a batch of requests.

For the other two sizes of codes however, the regular batch code model is able to achieve higher service rates compared to the asynchronous model. For the codes generated using the $[15, 4]$ simplex matrix, the regular batch code had a 2% greater service rate r_s and the $[31, 5]$ code had a 6% greater r_s, when compared to the corresponding asynchronous codes.

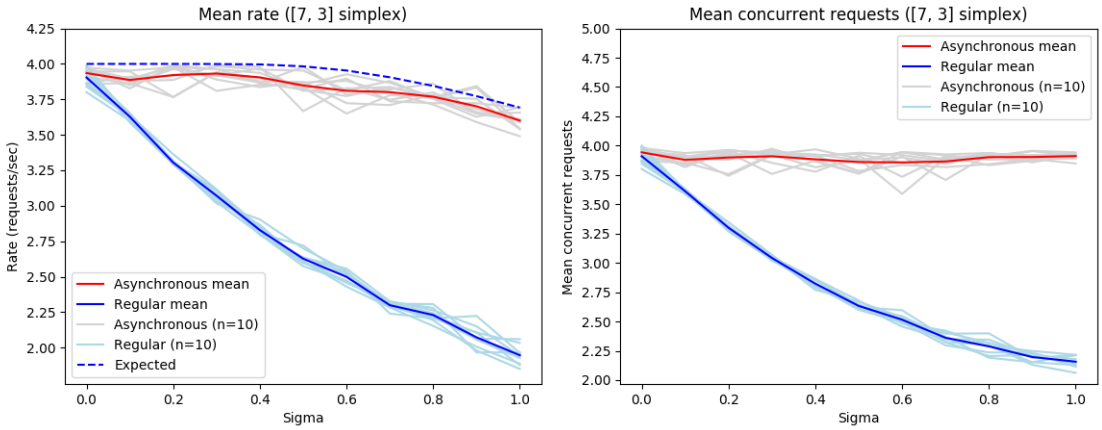
4.3 Increased Variance of Request Lifetimes

While the regular batch code model works well in the idealized setup described in section 4.2, these are not indicative of a real-world storage system, on which the models would applied. To study the effects of request incoming request lifetime variance on both models, simulations were run using the normal (Gaussian) distribution with increasing variance. In this round of simulations, request lifetimes were generated using the normal distribution with $\mu = 1$ and $\sigma \in 0, 0.1, ..., 1.0$ for both models. It should be noted that using $\mu = 1$ and $\sigma > 0.3$ to generate the lifetimes of the requests can result in negative values. In the system model, the requests assigned with negative lifetime values are converted into positive values of one millisecond.

To validate the trends shown in the simulations, codes generated by both the $[7, 3]$ and $[15, 4]$ simplex code matrices were tested. The incoming request rates for each file were uniformly distributed with a total incoming request rate $r_{inc_{total}} = t$, where $t = 2^{k-1}$, for both codes.

The graphs in figure 5 show the mean service rates r_s and concurrent requests c_r of 10 simulations for each $\sigma \in 0, 0.1, ..., 1.0$, with a duration of 5 minutes for each simulation. Both the regular and asynchronous batch code models were generated by the $[7, 3]$ simplex matrix, using gap time scales $g_{f_i} = 0.75$ for $i \in 1, 2, 3$, resulting in a total incoming request rate $r_{inc_{total}} = t = 2^{k-1} = 4$. The asynchronous models in the simulations had a skip distance of $d = 8$.

While it can be seen from figure 5 that the regular batch code model had a significant drop in its service rate r_{s_b}, which was to be expected due to some requests in a batch finishing sooner than others, the asynchronous model also displayed a drop in the mean rate of requests served per second r_{s_a}. As previously stated, using the normal distribution to generate lifetimes can result in negative lifetime values when $\sigma > 0.3$. This, in turn, effects the real mean lifetime of the requests, shifting it from $\mu = 1$, which should be the mean using normal distribution, to a larger value, as noted by "Mean lifetime" in table 7. This directly affects the rate of requests served that the models could achieve if they supported mean concurrent requests equal to the total rate of incoming requests ($c_r = r_{inc_{total}}$). This theoretical limit can be seen in the figure 5a and table 7, denoted as "Expected". The increased performance in service rates for the asynchronous model compared to the regular batch code model is denoted in table 7 as "Ratio", calculated as $\frac{r_{s_a}}{r_{s_b}}$.



(a) Service rate r_s

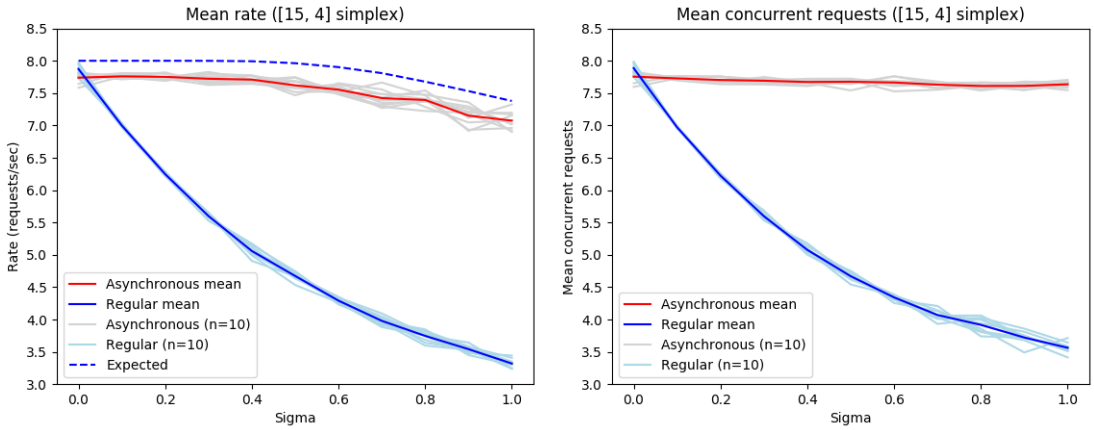
(b) Concurrent requests c_r

Figure 5. Mean service rate r_s and mean concurrent requests c_r with increased σ

σ	Mean lifetime	Expected	r_{s_a}	r_{s_b}	Ratio (%)
0.0	1.0	4.0	3.935	3.904	101%
0.1	1.0	4.0	3.887	3.628	107%
0.2	1.0	4.0	3.921	3.305	119%
0.3	1.0	4.0	3.932	3.071	128%
0.4	1.0007	3.997	3.905	2.829	138%
0.5	1.0045	3.982	3.848	2.627	146%
0.6	1.0121	3.952	3.810	2.499	152%
0.7	1.025	3.902	3.801	2.299	165%
0.8	1.0415	3.841	3.769	2.230	169%
0.9	1.0622	3.766	3.701	2.073	179%
1.0	1.0856	3.685	3.601	1.946	185%

Table 7. Service rates per increased σ for both models generated by the $[7, 3]$ simplex matrix

The same simulations were repeated using the $[15, 4]$ simplex matrix to generate the codes, with a gap time scales $g_{f_i} = 0.5$ for $i \in 1, 2, 3, 4$, resulting in a total incoming rate $r_{inc_{total}} = 8$ for the requests. The asynchronous also had an assigned a skip distance of $d = 8$. A total of 10 simulations with a duration of 5 minutes each were run again using the normal distribution with $\mu = 1$ for each $\sigma \in 0, 0.1, ..., 1.0$ to generate request lifetimes.



(a) Service rate r_s

(b) Concurrent requests c_r

Figure 6. Mean service rate r_s and mean concurrent requests c_r using constant lifetime with increased σ

σ	Mean lifetime	Expected	r_{s_a}	r_{s_b}	Ratio (%)
0.0	1.0	8.0	7.739	7.872	98%
0.1	1.0	8.0	7.759	6.998	111%
0.2	1.0	8.0	7.752	6.247	124%
0.3	1.0	8.0	7.722	5.599	138%
0.4	1.0007	7.994	7.709	5.059	152%
0.5	1.0045	7.964	7.620	4.673	163%
0.6	1.0121	7.904	7.554	4.291	176%
0.7	1.025	7.805	7.423	3.979	187%
0.8	1.0415	7.681	7.394	3.747	197%
0.9	1.0622	7.532	7.152	3.542	202%
1.0	1.0856	7.369	7.075	3.319	213%

Table 8. Service rates per increased σ for both models generated by the $[15, 4]$ simplex matrix

As can be seen from the graphs (figures 5 and 6), when the variance of incoming request lifetimes increases, the asynchronous model greatly outperforms the regular batch code model. This is most obvious when comparing the amount of mean concurrent requests c_{r_a} and c_{r_b} supported by the models. While the mean values of c_{r_a} do not show a significant change as the σ increases, the concurrent requests c_{r_b} for the regular batch codes show a significant drop. The exact values for c_{r_a} and c_{r_b} and the ratio $\frac{c_{r_a}}{c_{r_b}}$ for both rounds of simulations are given in table 9.

	$[7,3]$			$[15,4]$		
σ	c_{r_a}	c_{r_b}	Ratio (%)	c_{r_a}	c_{r_b}	Ratio (%)
0.0	3.943	3.910	101%	7.755	7.885	98%
0.1	3.878	3.612	107%	7.728	6.972	111%
0.2	3.899	3.300	118%	7.703	6.225	124%
0.3	3.909	3.042	129%	7.691	5.597	137%
0.4	3.884	2.823	138%	7.672	5.077	151%
0.5	3.860	2.633	147%	7.676	4.670	164%
0.6	3.857	2.515	153%	7.664	4.342	177%
0.7	3.865	2.361	164%	7.630	4.070	187%
0.8	3.902	2.288	171%	7.611	3.917	194%
0.9	3.903	2.197	178%	7.612	3.721	205%
1.0	3.911	2.157	181%	7.636	3.563	214%

Table 9. Current requests for both batch code models generated by the $[7,3]$ and $[15,4]$ simplex matrices

4.4 Exponential Distribution

We also simulate a system where the lifetime of the requests is given by the exponential distribution. Such a distribution better describes the real-life behavior of the storage systems. The following simulations were made using the same setups as in section 4.2, with the exception of the request lifetime l, which was generated for each request using the exponential distribution with $b = 1$. As in section 4.2, the simulated codes were generated by $[n,k]$ simplex matrices, used request gap time scales $g_{f_i} = \frac{1}{t}k$ for $i \in 1,...,k$, where $t = 2^{k-1}$, resulting in $r_{inc_{total}} = t$. The asynchronous model also used a skip distance of $d = 8$.

The mean values for the service rate r_s, concurrent requests c_r and request queue times t_q of 10 simulations, with a duration of 5 minutes each, of each code can be seen in table 10.

	Regular batch codes			Asynchronous batch codes		
$[n,k]$	r_s	c_r	t_q	r_s	c_r	t_q
$[7,3]$	1.933	1.915	39.580	3.829	3.820	4.292
$[15,4]$	2.951	2.954	47.937	7.616	7.550	4.445
$[31,5]$	4.789	4.754	52.564	14.183	14.042	8.978

Table 10. Mean r_s, c_r and t_q for both batch code models using exponential request lifetimes

As can be seen from table 10, when generating request lifetimes using the exponential distribution, the difference in performance of the code models is significant. It can be seen that when using the smaller $[7, 3]$ simplex matrix to generate the code models, the asynchronous model offers a 98% increase in the mean service rate r_s compared to the regular batch code model, but using the $[31, 5]$ simplex matrix to generate larger codes, the improvement of service rates increases to 196%. This can be explained by the increased batch size t of the $[31, 5, 16]$ batch code, which increases the probability of having requests with greatly varied lifetimes in the same batch, as can be seen in the following example 4.1.

Example 4.1. The maximum lifetime difference l_Δ of two request lifetimes in a batch, when using the exponential distribution with $b = 1$. Calculated for regular batch codes generated by $[n, k]$ simplex codes as the mean maximum l_Δ of 100000 batches:

$[n, k]$	Batch size	l_Δ
$[7, 3]$	4	$1.8349s$
$[15, 4]$	8	$2.5835s$
$[31, 5]$	16	$3.3203s$

The lifetime difference l_Δ can be used to explain the significant drop in the service rate r_s of the regular batch code model as seen in table 10. l_Δ shows the mean time that is spent waiting for the entire batch to be finished serving while a recovery set, which could be immediately used to serve another request in the asynchronous model, remains unused by the regular batch code model.

While it can also be seen that using larger simplex codes for the asynchronous model causes an increased difference between the mean concurrent requests r_s and the theoretical maximum concurrent requests $t = 2^{k-1}$, this is likely due to the increased amount of blocking configurations of requests. The difference $t - r_s$ could possibly be reduced by using a skip distance $d > 8$, which was used for these simulations.

4.5 Requested File Distribution

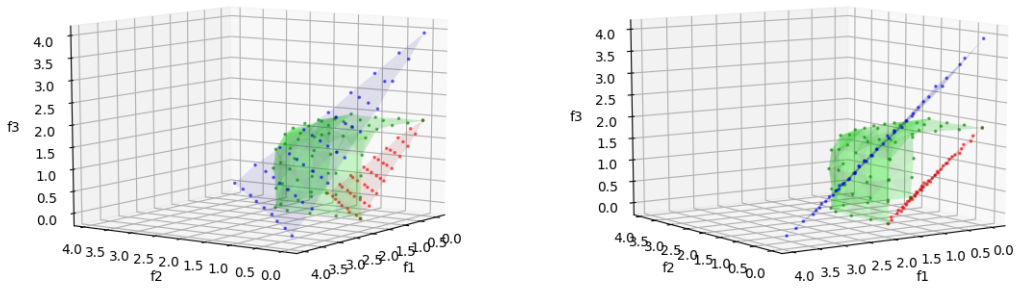
To visualize the differences between both the regular and asynchronous batch code models and the regular replication based storage models, as well as test the effects of all possible distributions of incoming request rates per file, simulations of the regular and asynchronous batch codes generated by the $[7, 3]$ simplex code matrix were made. The batch code models were compared to a replication based storage model, defined by the matrix $\mathbf{G}$ given in example 3.1, which uses a total of $n = 6$ servers to store $k = 3$ files by replicating each file on two servers.

For both batch code models, the total incoming rate of requests is $r_{inc_{total}} = 4$ with individual incoming rates $r_{inc_{f_i}}$ for $i \in 1, 2, 3$ taken from the range $0, 0.5, 1, ..., 4$. In

other words, all individual incoming rate configurations per file $r_{inc_{f_i}}$, which add up to a total incoming rate $r_{inc_{total}} = 4$, were simulated for both the batch code models. As the replication based storage model can support up to 2 concurrent requests for any file f_1, f_2, f_3, the incoming rates $r_{inc_{f_i}}$ for each file in the replication model are taken from the range $0, 0.5, ..., 2$, with a total incoming rate $r_{inc_{total}} \in 2, 2.5, ..., 6$.

For all of the simulations, the request lifetimes were generated using the exponential distribution with $b = 1$, as this is most indicative of a real-world scenario.

In the following figures 7a and 7b, each axis represents the individual service rates $r_{s_{f_i}}$ for each file, and the position (p_1, p_2, p_3) of each dot is calculated as $p_i = r_{inc_{f_i}} \cdot r_e$ for $i \in 1, 2, 3$ for each code model, where r_e is the service efficiency rate $r_e = \frac{r_s}{r_{inc_{total}}}$. The values for (p_1, p_2, p_3) are calculated using the mean rates of a total of 10 simulations, with a duration of 5 minutes, for each incoming file rate configuration for each code and are colored based on the model they represent, $blue$ for the asynchronous, red for regular batch and $green$ for the replication model.



(a) Angled view

(b) Side view

Figure 7. Requested file distributions for the replication (*green*), asynchronous (*blue*) and regular batch code (*red*) models

It can be seen in the figure 7 that the distribution of incoming request rates per file does not have any significant effect on the performance of either batch code model in terms of the service rate r_s. This is apparent from figure 7b, as the dots representing the models form nearly perfectly flat plains in the graph. If there were certain configurations of incoming request rates $r_{inc_{f_i}}$ that caused a drop in the service rate r_s, regions matching those configurations would display some curvature in the figure 7b.

As the position of each dot in the graphs (fig. 7) is determined by the service efficiency rate r_e, the distance of any dot from the point $(0, 0, 0)$ is indicative of the service rate r_s

that the codes achieve using their given incoming request rates per file $r_{inc_{f_i}}$. As is visible from the graphs, using exponential request lifetimes for the simulations, the replication model outperforms the regular batch code model in each configuration of the incoming request rates $r_{inc_{f_i}}$. This is due to the replication model always using one server to serve a request, unlike the batch code models, and not suffering from the performance drop caused by the variance of request lifetimes, as the regular batch code model does.

Similarly, the positions of the two plains connecting the dots which represent both the batch code models shows that the 98% increase in performance in favor of the asynchronous model, as shown in section 4.4, holds for all configurations of incoming request rates.

Comparing the replication model to the asynchronous batch code model indicates that if the rates $r_{inc_{f_i}}$ are more uniformly distributed, then the replication based model does offer an increased service rate r_s compared to the asynchronous model while using one less server for the storage system. However it can be seen from the figures 7a, 7b that there are large regions depicting areas where the incoming rates are in the range $4 \geq r_{inc_{f_i}} > 2$ for any file f_1, f_2, f_3, where the asynchronous model outperforms the replication based storage model.

4.6 Summary of the Analysis

As it was observed in [6], varied lifetimes of requests to storage systems using the regular batch code model will lead to a suboptimal performance of the storage system. This is demonstrated by the drops in the mean service rates r_s, concurrent requests c_r and increased mean queue times t_q for the incoming requests as shown in section 4.3. The only two cases in which the regular batch code model displayed an increased service rate r_s compared to the corresponding asynchronous models, were the codes generated by the $[15, 4]$ and $[31, 5]$ simplex matrices with a constant incoming request lifetime of $l = 1$, as demonstrated in section 4.2. However, as constant request lifetimes are not indicative of a real-world storage system, the results of section 4.4, where request lifetimes follow the exponential distribution using $b = 1$, should be used to compare the two batch code models. From these results it can be seen that in a more realistic scenario, the asynchronous batch code model will always be, by a vast margin, superior to the regular model in regards to the service rates achieved by the two models.

From the results of section 4.5 it can be seen from the nearly perfectly flat plains representing the regular and asynchronous batch code models generated by the $[7, 3]$ simplex matrix in the graph in fig. 7b, that the distribution of incoming request rates $r_{inc_{f_i}}$ for the files f_1, f_2, f_3 does not affect the service rates of either batch code model.

As noted in section 4.5, if the incoming rates are in the range $4 \geq r_{inc_{f_i}} > 2$ for any file f_1, f_2, f_3, the asynchronous batch code model has an increased service rate r_s compared to the replication model used in the simulations of section 4.5. As this is the case, it can be said that if a storage system experiences periods of increased incoming

rates for any of the stored files, the asynchronous batch code model could also be a superior alternative to the replication based storage model used in the simulations.

5 Conclusion

The asynchronous batch code model for storage systems was introduced as an variation of the regular batch code model, which allows for parallel recovery of items from a coded database in an asynchronous manner, increasing the service rate of the system when dealing with requests of varying duration. In this thesis, to study the differences in performance of the two models, a system prototype was developed which was used to run simulations of both models. A detailed description of the prototype is provided in the thesis, including algorithms and examples of the simulation processes, highlighting the differences of the models.

To compare the two models, simulations of storage systems generated by the same $[n, k]$ simplex matrices using either the regular or asynchronous batch code model were performed. The models were analyzed using multiple distributions of request service time, as well as varying rates for incoming requests. In addition, to improve the achievable service rates by increasing the average amount of concurrent requests achieved by the asynchronous model, the skip distance parameter was introduced. The effects on performance of this modification of the model were analyzed.

Finally, comparisons are made to a non-coded, replication based storage model, providing examples of conditions, under which the asynchronous model would be an advantageous alternative to the replication based storage system.

6 Acknowledgements

First and foremost, I would like to thank my supervisors Vitaly Skachek and Eldho Kuppamala Puthenpurayil Thomas for their patience, support and guidance throughout the making of this thesis.

The work on this thesis was supported by the Estonian Research Council grant PRG49 and the European Regional Development Fund through Mobilitas Pluss grant MOBJD246. I am also grateful to the IT Academy for the specialization stipend which was incredibly helpful throughout my Master's studies.

References

[1] Y. Ishai, E. Kushilevitz, R. Ostrovsky, A. Sahai "Batch codes and their applications," in *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*, June 2004, Chicago, IL.

[2] V. Skachek "Batch and PIR Codes and Their Connections to Locally Repairable Codes" in Network Coding and Subspace Designs, Editors: M. Greferath, M. Pavcevic, M.A. Vazquez-Castro, N. Silberstein, Springer 2018.

[3] H. Lipmaa, V. Skachek "Linear Batch Codes" in Proceedings 4th International Castle Meeting on Coding Theory and Applications (ICMCTA), Palmela, Portugal, Sept. 2014

[4] H. Zhang, V. Skachek "Bounds for Batch Codes with Restricted Query Size" in Proceedings IEEE International Symposium on Information Theory (ISIT), Barcelona, Spain, July 2016

[5] A. Vardy, E. Yaakobi "Constructions of batch codes with near-optimal redundancy" in IEEE International Symposium on Information Theory, Barcelona, Spain, Aug. 2016

[6] A.-E. Riet, V. Skachek, E. K. Thomas "Asynchronous Batch and PIR Codes from Hypergraphs" in Proceedings IEEE Information Theory Workshop (ITW), Guangzhou, China, Sept. 2018

[7] Z. Wang, H. M. Kiah, Y. Cassuto "Optimal Binary Switch Codes with Small Query Size" in IEEE International Symposium on Information Theory (ISIT), Hong Kong, China, Oct. 2015

[8] The Python Standard Library Documentation itertools `https://docs.python.org/2/library/itertools.html#itertools.combinations_with_replacement`, Aug. 2019

[9] NumPy Python Library Documentation numpy.random `https://docs.scipy.org/doc/numpy-1.14.1/reference/routines.random.html`, Aug. 2019

[10] C. Forbes, M. Evans, N. Hastings, B. Peacock "Statistical Distributions Fourth Edition" , May 2011

[11] Y. Zhang, E. Yaakobi, T. Etzion "Bounds on the Length of Functional PIR and Batch codes" arXiv:1901.01605v2, Apr. 2019

[12] R. M. Roth "Introduction to Coding Theory " Cambridge University Press, March 2006

7 Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sander Mikelsaar**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Empirical Study of Asynchronous Batch Codes,

   supervised by Vitaly Skachek and Eldho Kuppamala Puthenpurayil Thomas.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Sander Mikelsaar ***14/08/2019***