Synchronization in High-Performance Large-Scale Distributed Systems

Sverre Krohn Tennøe

Master Thesis

May 2, 2008
Abstract

This master thesis proposes a design for distributing the *generation controller*, a component in the FAST search architecture. The purpose of the generation controller is to ensure that the system maintains a consistent index while introducing new content. The content is processed, indexed and queried in a fully distributed and parallel manner, which makes the need for some synchronization scheme apparent. Today’s design and implementation of the generation controller builds on two-phase commit. The current generation controller is a single-node solution, and for large installations, requiring fault tolerance as well as scaling for performance issues, it is regarded insufficient.

The most important finding in this thesis is Binomial Two-Phase Commit. It is a commit protocol intended for scenarios where there are a large number of participants involved with each transaction and the overhead of receiving votes from each participant is too big for a single coordinator to handle. The coordinator only needs to receive votes from a subset of the participants and the size of the subset is independent of the total number of participants. As a result, it scales with O(1) as number of participants increase. It accomplishes this by trading off the possibility to *guarantee* that a transaction is committed atomically.
# Contents

1 Introduction ........................................ 15
   1.1 Background and Motivation ......................... 15
   1.2 Problem Statement .................................. 17
   1.3 Method ............................................ 19
   1.4 Contributions ..................................... 20
   1.5 Outline ........................................... 20

2 FAST Architecture Overview .......................... 21
   2.1 Search Engines: The Basics ......................... 22
   2.2 Content Submission Service (CSS) .................. 23
      2.2.1 Purpose ........................................ 23
      2.2.2 CSS composition ................................. 24
      2.2.3 Operation ...................................... 24
   2.3 Content Processing Subsystem (CPS) ................. 26
      2.3.1 Purpose ........................................ 26
      2.3.2 CPS composition ................................. 26
## CONTENTS

2.3.3 Operation .............................................. 26

2.4 Content Distribution Service (CDS) .......................... 27
2.4.1 Purpose .................................................. 27
2.4.2 CDS composition ......................................... 28
2.4.3 Operation .................................................. 28

2.5 Content Target Service (CTS) ................................. 29
2.5.1 Purpose .................................................. 29
2.5.2 CTS composition .......................................... 30
2.5.3 Operation .................................................. 32

2.6 The Generation Controller (GC) ............................... 32
2.6.1 Purpose .................................................. 32
2.6.2 GC composition .......................................... 33
2.6.3 Operation .................................................. 34

2.7 Query Result Servers (QR) ................................... 36
2.7.1 Purpose .................................................. 37
2.7.2 QR composition .......................................... 37
2.7.3 Operation .................................................. 37

2.8 Summary ..................................................... 38

3 Theory and general issues ........................................ 41
3.1 Asynchronous versus Synchronous Distributed Systems .... 41
3.2 Fault Tolerance .............................................. 42
3.3 Timing and Clocks ............................................ 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Failures and Failure Detection</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Failures</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Failure Detectors</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>Consensus and Master Elections</td>
<td>46</td>
</tr>
<tr>
<td>3.6</td>
<td>Distributed Transactions</td>
<td>47</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Two-phase commit protocols</td>
<td>48</td>
</tr>
<tr>
<td>3.7</td>
<td>Statistics and Probability</td>
<td>50</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Probability</td>
<td>50</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Distributions and Confidence Intervals</td>
<td>51</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Combinatorics</td>
<td>53</td>
</tr>
<tr>
<td>3.8</td>
<td>Summary</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>Design</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Replacing 2PC</td>
<td>58</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Binomial Two-phase Commit (B2PC)</td>
<td>58</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Using B2PC in Mars</td>
<td>62</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Estimating thresholds</td>
<td>63</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Other alternatives to 2PC and B2PC</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Distributing the Generation Controller</td>
<td>66</td>
</tr>
<tr>
<td>4.2.1</td>
<td>One generation concept</td>
<td>68</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Multiple generation concepts</td>
<td>73</td>
</tr>
<tr>
<td>4.3</td>
<td>Complete generation controller design</td>
<td>75</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Purpose</td>
<td>76</td>
</tr>
</tbody>
</table>
4.3.2 GC Composition ............................... 76
4.3.3 Operation ................................. 78
4.4 Summary .................................. 79

5 Analysis ................................. 81
5.1 Performance analysis of commit protocols .......... 81
  5.1.1 2PC .................................. 81
  5.1.2 B2PC ................................. 82
  5.1.3 Ring-based 2PC ......................... 84
  5.1.4 Hierarchal 2PC ......................... 85
  5.1.5 Summary of commit protocols ............... 86
5.2 Sample size in estimation of confidence intervals ...... 86
5.3 Thresholds ................................ 88
5.4 Accuracy of the estimation technique ................ 96
5.5 Performance of Generation Coordination ............. 101
  5.5.1 Token-based .......................... 102
  5.5.2 Master-based ......................... 102
  5.5.3 Multiple generation concepts .............. 103
5.6 Summary .................................. 104

6 Discussion .................................. 107
6.1 Commit protocols ................................ 107
6.2 Token-based generation coordination ............... 110
6.3 Summary .................................. 111
CONTENTS

7 Conclusion

7.1 Conclusion .......................................................... 113
7.2 Future work ......................................................... 115
7.3 Acknowledgements ................................................. 116
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overview of the relevant services in FAST’s search platform</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>CSS composition. The CSS is made up of several CSS nodes working in parallel.</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>CDS composition. CSS instances relate to CDS instances. The CDS instances relate to all CTS instances and the generation controller.</td>
<td>28</td>
</tr>
<tr>
<td>2.4</td>
<td>CDS instances route annotations to correct partitions and notify the generation controller when a full content batch has been distributed.</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>CTS instances organized in rows and columns.</td>
<td>31</td>
</tr>
<tr>
<td>2.6</td>
<td>The generation controller receives batch identifiers and schedules new generations.</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>The generation controller receives batch identifiers and initiates a new generation using two-phase commit. When the generation has committed, the QR servers are notified of the new generation, and the content batches in the generation are acknowledged to the content distributor.</td>
<td>35</td>
</tr>
<tr>
<td>2.8</td>
<td>Clients send queries to QR servers. The QR servers routes them to the partitions of the CTS.</td>
<td>38</td>
</tr>
<tr>
<td>2.9</td>
<td>QR server annotates queries with current generation, if no specific generation was provided by client.</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>State diagram for coordinator in B2PC.</td>
<td>60</td>
</tr>
</tbody>
</table>
4.2 State diagram for participants in B2PC. .......................... 61
4.3 State diagram for generation controller using B2PC. .............. 66
4.4 Generation Controllers organized in a ring. One is elected master and is responsible for keeping the circle consistent in case of failures. The token is passed round the ring and the token counter is increased for each step. 70
4.5 Master generation controller initiates a new token, as the failed generation controller held the token as it failed. 1) The GC requests state information from the CTS instances. 2) the GC decides if there is need to commit or abort a pending generation and sends the decision to the CTS instances. 3) The QR servers are notified of what the newest generation is. 4) The GC acknowledges batches to the CDS instances, and/or asks them to resubmit unacknowledged batches to another generation controller instance. 5) A new token is initiated using the knowledge of the last committed generation. .......................... 72
4.6 Sequence diagram for multiple generation concepts showing the problem of non-sequential index evolvement. .......................... 74
4.7 System overview with new generation controller design. .......... 76
4.8 Sequence diagram for generation controller interaction. .......... 78
4.9 Sequence diagram for B2PC part of figure 4.8. ........................ 79
5.1 Total number of messages from the generation controllers point of view, as a function of number of CTS instances. .......................... 83
5.2 The number of participants needed to make sound estimations, with respect to percentage of commitability. .......................... 87
5.3 Partition fail frequency for 2 partitions of 5000 CTS instances each. 91
5.4 Close up of partition fail frequency for 2 partitions of 5000 CTS instances each. .......................... 92
5.5 Partition fail frequency for 5000 partitions of 2 CTS instances each. 92
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Partition fail frequency for 5000 partitions of 2 CTS instances each with interval of 1 failure.</td>
<td>93</td>
</tr>
<tr>
<td>5.7</td>
<td>Partition fail frequency for 100 partitions of 100 CTS instances each.</td>
<td>95</td>
</tr>
<tr>
<td>5.8</td>
<td>Close up of fail frequency for 100 partitions of 100 CTS instances each.</td>
<td>96</td>
</tr>
<tr>
<td>5.9</td>
<td>Partition fail frequency for AllTheWeb example between 0 and 50 failed nodes.</td>
<td>100</td>
</tr>
<tr>
<td>5.10</td>
<td>A thought example of how the number of live nodes evolve over time.</td>
<td>101</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In this chapter the background and motivation behind our work with the generation controller is presented. The generation controller is a component in Fast Search and Transfer's (FAST) search architecture. The internal FAST related information in this chapter is in large a result of discussions with Torgeir Hovden and Rolf Michelsen, which were external supervisors for this thesis. A problem statement is reached, and the approach to solving it is given, as well as our contributions and finally outline the rest of the thesis.

1.1 Background and Motivation

The revolution of information technology over the last 50 years, with the Internet as the crown jewel, allows people around the globe to create, publish and access vast amount of information. With the new trend of self-publication, everyone can upload whatever they want. And they do; A quick search for "*" on Youtube returns more than 72 Million videos (24 Feb, 2008). Blogs, Facebook, Youtube and forums are all means of self-publication and the amount of information out there is ever increasing. Ironically, making all this information available could make the information you are actually looking for hard to find.

Search engines are today used to organize haystacks of information enabling
information seekers to find their needles. Relevant information is drowning in an ocean of irrelevant information, and a good search engine should identify and bring relevant information to the surface. The importance of search is becoming more and more apparent. Microsoft are currently (spring 2008) acquiring FAST, and this shows that no information technology company can dismiss the importance of search. Future search engines may have to relate to unthinkable amounts of information, so building scalable solutions today will likely turn out to be a huge advantage in the future. On the opposite, not including scalability as a non-functional property today, will in most likelihood haunt you at some point in the future as the fundamental search engine design approaches the limits of its capacity.

The perhaps best known participants in the search engine market are Google and Yahoo!. Most users would recognize their web search engines, but there is also a market for enterprise search engines. These search engines are intended for use within businesses\(^1\). FAST delivers enterprise search installations to companies all over the world.

FAST’s search platform is called FAST ESP (FAST Enterprise Search Platform). It has been developed since the beginning of FAST, 1997. In those days, web search was the main focus area, and some of the principles applied bear the mark of the web context. Web content 10 years ago was fundamentally different from the content indexed in today’s enterprise search installations. The change is first and foremost that content changes rapidly. As content changes frequently, a "snapshot" of the index will not be valid for long. To give the user sensible search results, the index needs to be refreshed continuously. This has some interesting implications on design principles for the search engine and affects performance, consistency and latency of queries and indexing of new content.

There is today ongoing development of a new search core, more suitable for rapidly changing content. In the old search core document processing, indexing and the querying was done sequentially. Building the index was done as a batch job, meaning that no search queries was processed during the indexing. To offer availability during indexing, the search engine consisted of more than one version of the index. Each version were rebuilt in turns, and the versions not being rebuilt processed queries. Each time content was

\(^1\)Often referred to as "behind the firewall".
made searchable, it involved rebuilding the index. As the index needed to be refreshed more often and the amount of content increased, indexing became a huge, time-consuming job. To illustrate the impact of this design, a full rebuild of FAST’s old web search engine — AllTheWeb.com — lasted as long as 12 days, possibly implying that your search results would be 12 days old\textsuperscript{2}! In those days, according to FAST, this was one of the faster web search engines in terms of indexing and FAST’s slogan was “freshness matters!”.

Today, FAST offer search for businesses. Content changes more frequently, and this change in the nature of content motivated the need for a new search core design using an updateable index, where updates and queries can be executed in concurrent fashion. One of the goals of the updateable index is to provide sub-second indexing, meaning that a document should be visible in the index in less than a second after it was submitted to the search engine. While the updateable index design is suitable for rapidly changing content, a plethora of issues, not relevant in the old index, arise. This thesis will address some of those concerns.

\section*{1.2 Problem Statement}

Fast uses the notion of generations to identify different versions, or snapshots, of the search index. New content is made available through a new generation. A generation identifier is associated with the content, and this identifier should incrementally grow as new content is indexed. A generation controller is responsible for assigning these identifiers and coordinating the creation of new generations. A detailed description of how this component functions is given in Section 2.

The importance of having generations is first and foremost in order to offer clients a way to do search on consistent data sets. Clients should be able to trust that search results represent the indexed documents correctly. More precisely, the search results should not be influenced by failures within the search engine or be a result of how the system is organized internally. For instance, if a batch of documents that should have been made visible in the index at the same time, is only partially included in queries, the client will

\footnote{but the query would not take 12 days to process, a subtle distinction.}
see inconsistent search results. In addition, the client needs to be able to do a series of queries on the same data set. This is also accomplished through generations. All queries on the same generation are conducted on the same version of the index — i.e. content part of newer generations are excluded, even though the generation they are part of is searchable. Generations are also a tool for optimizing indexing efficiency. The synchronization related to adding content to the index introduces some overhead. By including more content in each generation, the synchronization overhead can be reduced, trading off indexing latency.

The generation controller is the component allocating and initiating new generations. It synchronizes the distributed index through a commit protocol, and it is a single-node solution. Though having a single, centralized node solves many issues, it introduces others. For one, scalability becomes a problem. Imagine a football field size data center of servers that should be organized by a single running generation controller. At some point, the centralized node will simply not be able to cope. The lesson learnt from the old AllTheWeb-days was that when a single node needs to communicate regularly with more than 200 other nodes or so, the network interface will easily begin to struggle. In our scenario, each generation involves communication with all search nodes in the system, and in combination with sub-second indexing, the single-node generation controller will soon turn out to be a bottleneck.

In the old search core, scalability was achieved at the expense of indexing latency, as the 12-day indexing example illustrates. With the updateable index, sub-second indexing is a crucial property, and another way to achieve scalability not affecting indexing latency is required.

If our design today, offers no possibility to scale further than 200 nodes, when the time comes to deploy such a system, the fundamental design for doing generation coordination must be altered. The effects of changing a fundamental component like the generation controller, would probably affect an unforeseeable amount of other components and their design.

The fundamental design should strive to offer scalability as a non-functional property. It is a classic mistake in the world of software development to assume that present needs will remain valid in the future. Internet Protocol version 4 addressing is one example, the issues with Y2K is another.
Fault tolerance is also important. The single-node generation controller serves as a single point of failure. If it fails, no content will be indexed. Having a centralized generation controller solves search engine consistency, but scalability and fault tolerance become problems.

To summarize, the need to guarantee one property often have an effect on a different property. It seems, there are no "perfect" solutions for all — even any — scenarios. Trade-offs must be identified and considered, and finding the right balance between consistency, performance, scalability, implementability, fault tolerance etc. is definitely not trivial. Finding this balance is the goal of this thesis.

The problem statement is defined as follows:

*How can a distributed generation controller, handling failures gracefully, achieving index consistency effectively without compromising search performance nor indexing latency and scales with respect to content load, number of nodes and query load be designed?*

### 1.3 Method

In this section the approach to solving the problem statement is described. A solution to the problem statement will be designed. Its properties are examined, through an in-depth analysis, and experiments provided where needed. To actually implement the solution, and running meaningful tests will involve deploying a system possibly consisting of *thousands* of nodes. Implementing a running solution is beyond the scope of this paper. Running simulations was considered, but simulations would only be valid under the chosen assumptions of the simulation environment, and those assumptions may not be valid in a real environment. Another problem with simulations in a large-scale environment is that the accuracy becomes questionable as some fraction of inaccuracy is introduced for each simulated component. Thus, an in-depth analysis and some experiments are provided where appropriate.
1.4 Contributions

In short, the solution deals with two problems. The first is scaling of Two-Phase Commit (2PC) with respect to participants. As the number of participants grow, the coordinator becomes a bottleneck, as it has to manage votes from each participant. Binomial Two-Phase Commit is proposed as a replacement for 2PC. The semantics of 2PC has somewhat changed, so readers should be aware that it does not offer the same degree of consistency. It does however, scale extremely well, as number of participants is not a factor in its operation.

The second problem explored is the need for a synchronization technique that scales with respect to the amount of data to be synchronized. In our case, it means that as the amount of content to be indexed increases, a single node coordinating the content into new generations might become overloaded. This is different than the problem of synchronizing a big number of nodes. The amount of content to be synchronized is a result of the number of documents to be indexed per time unit, not the number of nodes needed to hold the search index. Token-based coordination is proposed, where a set of nodes in turn, are allowed to add content to the index. The solution is able to handle a larger amount of content, at the expense of indexing latency.

1.5 Outline

In this chapter, the problem at hand was introduced. The reader should now have a basic understanding of the problem this thesis aim to solve, and be prepared for a detailed description of the system and its components, presented in Chapter 2. Relevant literature is addressed and further explanations to the problems faced are given. Then, in Chapter 4 a different solution to today's generation coordination is proposed. We will show how our solution solves the problems, and argue its strengths and weaknesses. In Chapter 5 the properties of the design and the algorithms used are explored. A discussion and comparison with the existing solution follows in Chapter 6, before we deduce a conclusion from our work in Chapter 7.
Chapter 2

FAST Architecture Overview

This chapter will give a detailed description of the relevant components discussed through this paper. In order to present a good solution to the problem at hand it is important to fully understand the assumptions and guarantees offered by the system. A good amount of effort will be spent at describing the components at hand. The system is approached from a content point of view, assessing each component a document fed to the search engine encounters. The chain of components a document encounters is referred to as the processing pipeline. Finally a look at the system from a querying point of view is given. The description of the components is based on internal FAST design documents [26] [25].

FAST are currently developing a new search core from scratch. This project is known internally as Mars. The motivations behind the new search core are diverse. The old core have through the years grown to become a complex platform. As the old platform grew with demanded functionality, its increased complexity made it harder to maintain. It requires much knowledge from the developers today to see the complications and effects of changing it. Also, FAST found that a new core could provide more flexibility, meaning that new requirements could be easier included using new and modern design principles.

The responsibilities of the search core are all the fundamental functionalities needed in a search engine. This is amongst others a reversing process of
CHAPTER 2. FAST ARCHITECTURE OVERVIEW

documents, referred to as document processing and making the processed documents — referred to as content — part of the search index. Finally, the core must offer some interface for sending queries to the index. In the next chapters these functionalities are explored and FAST’s search architecture is outlined.

2.1 Search Engines: The Basics

The basic operation of a search engine is now exemplified. A very simplified example of what a search engine actually do is given. The example does in large, give an overview of the most important search engine functionality.

A search engine is really a simple thing. Imagine a normal document like the one you are reading right now. It contains words. If you read this document, it points you to the words within. The search engine reverses this structure, creating a data structure where words point to documents. This way, you can ask the search engine for documents having occurrences of specific words, as opposed to the words are kept in specific documents. The data structure the search engine keeps this information in is called an inverted index. A common example of an inverted index is the index at the end of many scientific books, used to look up specific words of interest and their position in the book.

The inverted index is built from a set of, what FAST refers to as, annotations. One annotation can be thought of as a single word-to-document pointer. From this document, there would typically be one annotation for each unique word, pointing back to this document. The inverted index can be asked to return a list of all documents containing some word. Basically, this is it! Ask a web search engine for web sites containing a specific word, the web search engine looks up pointers to the documents in the inverted index and returns the results. There are usually a lot of added functionality like ranking of search results, filtering, where the search engine notifies the client if documents matching the query are indexed, but all of these are made possible through the reversed data structure mentioned. In addition, search these days are much more than simple keyword matching, but this is not to our concern.

Each component relevant for indexing and querying in FAST’s search plat-
form, is now presented. In figure 2.1 an overview over the services discussed is given. The Content Submission Service receives documents for inclusion in the index. The Content Submission Service uses a subsystem responsible for producing the annotations added to the index. The Content Distribution Service’s responsibility is correct routing of annotations to the Content Target Service. The Content Target Service is where the inverted index is situated. The Query Result Servers are responsible for receiving and routing queries correctly and assemble search results before they are handed back to the client.

In addition, there is another component coordinating the inclusion of new content in the index. This component is called the generation controller and it is addressed in section 2.6.

2.2 **Content Submission Service (CSS)**

2.2.1 **Purpose**

The CSS is the connection point for submitting content to the search engine. In a web search engine there is typically web crawlers pushing websites to
the search engine, but in enterprise search there is a range of different components delivering content to the search engine. In enterprise search, content submitters can be web crawlers, file traversers or even database connectors. The purpose of the CSS is to offer an interface for the range of content submitters adding content to index. This interface also supports tracking of documents, in order for content submitters to monitor the indexing.

### 2.2.2 CSS composition

The CSS is made up from several CSS instances. The CSS instances work in parallel providing fault tolerance and increased submission capacity. Each instance is assigned an unique identifier namely a `serviceinstanceid`, all shown in Figure 2.2.

![CSS Composition Diagram](filename.png)

**Figure 2.2:** CSS composition. The CSS is made up of several CSS nodes working in parallel.

### 2.2.3 Operation

A client opens a session with one CSS instance and feeds the documents that should be indexed. Within the session it is possible to feed documents in *content batches*. All documents within a batch should appear simultaneously in
the index. For each content batch a content policy is defined. All documents in a batch have the same content policy. Properties of the policy include:

**Latency:** The maximum latency until the batch should be visible in the index. The value 0 indicates that the processing should be done as fast as possible.

**Priority:** Integer between 0 and 31 specifying the priority the batch should have in the feeding process.

**Causality:** Specifies if the batches within a session should become visible in the order they were fed to the CSS.

**Failure action:** Indicates action when the feeding process fails.

Each session within a CSS instance is assigned an unique identifier, a `sessionid`. One CSS instance can serve multiple clients. The clients each opens a session. Furthermore, within each session, batches are assigned identifiers, called `batchids`. Batch identifiers are unique within each session, and assigned incrementally. No holes are permitted, as holes are used to discover missing batches. The documents in a batch are assigned document identifiers, called `documentids`. These identifiers are also assigned incrementally, from 1 and upwards. Using these identifiers, each document are uniquely identified in the processing pipeline. The global identifier looks like:

<serviceinstanceid>.<sessionid>.<batchid>.<documentid>

Documents may also be split into parts using the same semantics as for document identifiers. These identifiers look like:

<serviceinstanceid>.<sessionid>.<batchid>.<documentid>.<partid>

The requirement that content batches should appear simultaneously in the index is of significant importance. This requirement imposes the need for a synchronization scheme, as will soon become apparent. The need comes from the fact that a content batch can consist of multiple documents that are routed to different parts of the search index. More on this follows in 2.4 and 2.5.
Equally important is perhaps the requirement of causality. The system should be able to guarantee causality between content batches within a session. This implies that the client should be able to trust that if it submits a series of content batches in succession, these will appear in the order they were fed in the index.

### 2.3 Content Processing Subsystem (CPS)

#### 2.3.1 Purpose

The CPS is responsible for processing documents into annotations. A document is processed into a set of annotations representing the inverted structure where words point to documents.

#### 2.3.2 CPS composition

The CPS is a sub-system of the Content Submission Service. It is considered part of the CSS. There are typically a set of stateless CPS servers, and the CSS can send documents to any one of them.

#### 2.3.3 Operation

The CPS offers a service interface to the CSS. The CSS submits documents for processing into annotations. The CPS forwards the processed annotations to a CDS instance.

As an example of what annotations are, the poem "When Roses cease to bloom, Sir" by Emily Dickinson, is presented.
When Roses cease to bloom, Sir,
And Violets are done –
When Bumblebees in solemn flight
Have passed beyond the Sun –
The hand that paused to gather
Upon this Summer’s day
Will idle lie – in Auburn –
Then take my flowers – pray!

When processed, the annotations would look something like:

<table>
<thead>
<tr>
<th>Word</th>
<th>Document Identifier</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>When</td>
<td>When Roses cease to bloom, Sir</td>
<td>1</td>
</tr>
<tr>
<td>Roses</td>
<td>When Roses cease to bloom, Sir</td>
<td>2</td>
</tr>
<tr>
<td>cease</td>
<td>When Roses cease to bloom, Sir</td>
<td>3</td>
</tr>
<tr>
<td>to</td>
<td>When Roses cease to bloom, Sir</td>
<td>4</td>
</tr>
<tr>
<td>bloom</td>
<td>When Roses cease to bloom, Sir</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Each row in the table represents one annotation. This example have a simplified view of what annotations are, but illustrates the reversing process documents go through, and keep in mind; annotations are indexed, not documents.

2.4 Content Distribution Service (CDS)

2.4.1 Purpose

The purpose of the CDS is to route annotations correctly into the distributed index. The index is partitioned amongst a number of nodes (see Section 2.5 for partitioning and replication of the index). As documents change and need to be re-indexed, the annotations from the documents must be routed to the correct partition of the index, and making sure the routing is done correctly is the responsibility of the CDS.
2.4.2 CDS composition

The CDS consist of a set of CDS instances, working in parallel to provide fault tolerance and increased routing capacity. A Content Submission Service instance normally relates to a single CDS instance. As seen in Figure 2.3, the CDS instances relate to all CTS instances and the generation controller (see Section 2.5 and 2.6 for descriptions).

Figure 2.3: CDS composition. CSS instances relate to CDS instances. The CDS instances relate to all CTS instances and the generation controller.

2.4.3 Operation

Figure 2.4 shows how the CDS instances are fed annotations from the Content Submission Service instances. Annotations from the same session are routed to a single CDS instance. The CDS instance needs to collect all annotations generated from a specific content batch. It uses the identification scheme presented in Section 2.2.3 to accomplish this. The last annotation in a batch is identified by a marking in the identifier. The CDS then routes annotations to the correct partitions (see Section 2.5.2) of CTS instances. A hash function is used to decide the partition an annotation belongs to. If a static property of the original document the annotation belongs to is used as input to the hash function, annotations will be routed correctly and the lookup is simple. The document URL is normally used as input to the hash function. Note that
annotations from different documents in a content batch will most probably be routed to different partitions as a result of the document URL being used for routing. The content distributor then informs the generation controller that a content batch is ready for indexing and where the annotations were routed to.

Figure 2.4: CDS instances route annotations to correct partitions and notify the generation controller when a full content batch has been distributed.

2.5 Content Target Service (CTS)

2.5.1 Purpose

The CTS is home of the search index. It adds new annotations to the index and also processes search queries retrieving search results from the index.

As mentioned previously, Mars is aimed at developing a new search core. Within this core, the concept of an updateable index is central. As new documents are pushed to the search engine at high rates, documents are indexed in less than a second. In order to do indexing without compromising search query efficiency or consistency, queries are processed and annotations are added to the index in parallel.
As new generations are added, annotations are added to the inverted index structure in a way allowing query processing on different generations. The index structure allows queries on different generations to be processed, not only on the newest generation it has added. For guaranteeing consistency, the index should in theory, be able to process queries on all generations which ever were. In practice, old generations are eventually "forgotten", as it grows more unlikely that any clients are interested in those.

2.5.2 CTS composition

A CTS instance can only hold a limited amount of content. If the search engine needs to index more content than a single CTS instance can hold, the index is partitioned between two or more CTS instances. The number of CTS instances will vary from installation to installation. The old FAST web search engine — AllTheWeb.com — consisted of 750 CTS instances, partitioned in 250 partitions. This was however many years ago. Speculators claim that Google now have more than 30’000 machines [17], some even claim they already exceeded 100’000 machines [4]. The largest installation FAST runs today is 200 CTS instances.

Partitioning has some implications. First of all, a mechanism for deciding how the content is partitioned is needed. This mechanism must ensure that updates on already indexed documents are routed to the correct partition. These updates could be that the document has been modified, or even removed all together. As mentioned, CDS instances route annotations by hashing on a document identifier, typically the URL of the document. This identifier stays constant regardless of the CDS instance, session or batch the document belongs to. This routing scheme ensures that annotations are consistently routed to the correct partition. This scheme makes lookups very cheap, and correctness is guaranteed if all CDS instances use the same hash-function. On the downside, it does not allow the CDS to take into consideration the state of the CTS instances. For instance, if a new, empty partition is added to a system, most of the new content should be routed to it, in order to exploit the free capacity. This is however tricky. As a result of hashing, the distribution is done evenly across the partitions. One way to exploit the empty partition, is to change the hash function, and do a full redistribution of the indexed content. This is no cheap task, as it would be
time consuming and cumbersome. During such a redistribution the system query capacity would be reduced. Routing policies are subject to research in itself. Partitioning allows scaling of the search index with respect to content, and in this thesis the hash-based routing policy is assumed.

For large systems, *query processing* must also be distributed. A single CTS instance will only be able to handle a certain amount of queries per time unit, so the CTS instances will have to be replicated. Replicated CTS instances offer equivalent views of its partition of the index. Queries can be sent to any of the replicas in a partition. Each CTS replica in a partition returns identical result sets to identical queries. This mechanism allows scaling the system with respect to *query load*.

Another important effect of replication is fault tolerance. By replicating the CTS instances in a partition, increased availability of the partition is achieved, making the index more tolerant to failures.

Partitioning and replication gives scalability of content- and query capacity. To describe a system FAST organizes the CTS instances in rows and columns. All CTS instances in a column are replicas, offering the same partition of the search index. A complete search index is offered by each row. To query the complete search index, a query must be sent to one CTS instance in each column, or all instances in a row if you wish.

![Figure 2.5: CTS instances organized in rows and columns.](image-url)

Figure 2.5 shows a logical layout of CTS instances in rows and columns. All
instances in a column are part of the same partition, while a complete row offers the complete indexed content. A query is routed to one CTS instance in each partition in order to retrieve results from the complete index. Note that the layout is only logical, a query can be routed to any instance in a column. In other words, one row may include any of the CTS instances in each column. This implies that if one of the CTS instances in a column fail, the logical row it belongs to does not fail, but the queries must simply be routed to another member of the same column.

2.5.3 Operation

The CTS instances receive annotations from the Content Distributor Service. These annotations are not added to the index directly, but buffered while waiting for the generation controller to initiate a commit protocol resulting in the annotations being added to the index as part of a generation. The generation controller is discussed in the next section, and a more detailed view of the interaction between CTS instances, CDS instances and the generation controller is presented.

2.6 The Generation Controller (GC)

2.6.1 Purpose

The purpose of the generation controller is to map content into new generations. This mapping must be done in a consistent way, meaning that all CTS instances map the correct content to the correct generation. The reason for adding new content as part of generations is to ensure that all documents in a content batch is made visible atomically, providing the searching clients a way to access the inverted index in a consistent manner.

Assignment of generation identifiers must be done in a consistent way, meaning that all CTS instances map a generation identifier to the correct content. When it comes to replicated CTS instances — all CTS instances within a column — queries are routed to one of the instances. The result should not
be dependent on the CTS instance the query is routed to. In other words, all CTS replicas in a partition must include the same annotations in a generation.

As for the partitions — the different columns — the generation controller needs to make sure that a generation spanning across the columns satisfy the atomicity and causality property of content batches. This implies that the decision of what content to include in each generation must be made in a way that includes complete content batches, as well as not breaking any causality.

Making a new generation introduces overhead, so by including more content, the total overhead of the generation concept is reduced. However, by including more content the latency of the indexing process increases, as there will be longer intervals between each generation. The rate at which the generation controller initiates a new generation will be a configurable property of each deployed installation. As quick indexing is one of the key concepts in the new search core, a new generation will in many installations be initiated in intervals between 10-50 milliseconds. There are also installations where the requirements to indexing latency is not as strict, and these installations typically initiate new generations in 30 second intervals.

### 2.6.2 GC composition

The existing solution for coordination of new generations is a single node solution. As for fault tolerance, no new content may be added to the index if it fails. A new instance must be restarted and recover its predecessors state. The generation controller keeps a batch identifier queue for each CDS instance, and a scheduler is responsible for deciding what content to include in the next generation.

As seen in figure 2.6, messages from the CDS instances are queued in separate queues at the GC and a scheduler selects content batches for inclusion in the next generation. The decision to make a new generation is either made by a time-out or by having a certain amount of content to index. This allows the GC to choose when to initiate a new generation with respect to latency and
2.6.3 Operation

The GC receives batch identifiers from CDS instances together with a list of the CTS instances the annotations were routed to. The content policy of the batch is also included. These messages are sent for each content batch distributed to the Content Target Service. Annotations are sent to a number of CTS instances, but the GC is a single node. The messages are small in size compared to the actual annotations sent to the CTS instances, but they are numerous.

The GC does not know that a batch consist of documents. It only sees batch identifiers, and it is all it need to know. The atomicity property is ensured.
by including entire batches in new generations, as the generation controller cannot make individual documents in a batch part of a generation. The generation controller needs to make sure that all partitions are capable of making the content batch searchable, as the documents within could have been — and probably will be — routed to different partitions by the CDS. If part of the content policy, the scheduler will assure that the causality requirement is not broken, and the scheduler may also delay a batch, if the content policy states that the batch does not need to be indexed straight away.

In Figure 2.7, the GC decides to initiate a new generation. It uses the two-phase commit protocol (see Section 3.6.1) known from distributed databases to commit a new generation at the CTS instances. It ensures that all CTS instances can make the new generation searchable, across partitions and replicas. Thus, if the generation is committed, all partitions and CTS instances can use the new generation and the atomicity property is ensured. The generation controller sends a message to all CTS instances informing them of the generation identifier and what content should be included. The CTS instances add the annotations to the log — making them persistent, and acknowledges that it is capable of making the new generation searchable. By sending this reply, it guarantees to the GC that it will be able to make the
CHAPTER 2. FAST ARCHITECTURE OVERVIEW

generation searchable in case of any failures, provided that it survives. This typically means that if it looses power, crashes etc. it will be able to make the generation searchable when it is restarted. If something goes wrong, it will notify the GC that it can not make the generation searchable. Such an error could for instance be that the disk is full, or anything that locally prevents it from committing the generation.

The GC waits for replies from all CTS instances. If all have acknowledged the new generation, it sends a new message to all the CTS instances, asking them to go ahead and actually make the generation searchable. If any of the CTS instances were not able to make the new generation searchable, or have failed to reply, the GC tells all CTS instances to abort the generation. It will also tell the CDS that something went wrong.

When the generation has committed, the generation controller sends notifications about the new generation to the Query Result servers and they can start annotating queries with the new generation identifier.

**Failure of the generation controller** If the generation controller fails, a new instance would have to be started. This instance needs to recover the state of the failed generation controller. This is done by gathering state from the CTS instances, as well as asking the CDS instances to resend unacknowledged content batch identifiers. If the reader is not familiar with the different states in the participant part of the two-phase commit, the reader is advised to look at Section 3.6.1 and specifically [3]. The CTS instances can be in state ”Prepare”, ”Ready”, ”Commit” or ”Abort”. If *all* the CTS instances are in uncertain state, the generation controller may assume that noone decided to abort the generation, and hence commit the generation. If anyone has committed or aborted the generation, that decision can be sent to all the CTS instances. It won’t occur that anyone aborted the generation if someone else committed it, as this is one of the properties of 2PC.

2.7 Query Result Servers (QR)

In the above sections the document feeding process has been described in detail. The reader should understand how documents are fed to the search
2.7. QUER Y RESULT SERVERS (QR)

engine, processed and made part of the search index. This section outlines the execution of queries and what clients should expect of the results.

2.7.1 Purpose

The purpose of the QR servers is to offer an interface for querying the index. The QR servers mask away the internal index composition, meaning that the clients does not have to relate to how the index is partitioned. The QR servers are responsible for routing queries to the different partitions of the index and assemble the results before returning them to the client.

2.7.2 QR composition

The QR servers work in parallel without any interaction themselves in between. By increasing the number of QR servers, the system provides higher availability to clients, and increase the query receive capacity of the system.

Figure 2.8 shows how the QR servers relate to the CTS instances, the generation controller and its clients. Clients send queries to a search interface offered by a QR server. A large system will consist of many QR servers that are responsible for routing queries to the correct CTS instances. All partitions must be represented in the result set. If a CTS instance is unable to reply to a certain generation the query must be routed to another CTS instance in the same partition.

2.7.3 Operation

As shown in Figure 2.9, when receiving a query, the QR server annotates the query with a generation identifier. If the client has specifically annotated its query with the generation of interest, the QR servers simply routes the query to a row of CTS instances. The QR servers also receives updates regarding the current generation from the generation controller.

Clients might issue a series of queries, expecting to get results from the same indexed content for all queries. This is achieved by requiring the client to
Figure 2.8: Clients send queries to QR servers. The QR servers routes them to the partitions of the CTS.

annotate the query with the generation identifier of interest. If the clients interaction with the search index as is seen as a transaction, and the search index as a shared resource, this functionality would normally prevent the shared resource from being updated until the client has completed its transaction. By offering multiple generations at the same time, inconsistency of search result from having an updateable index is avoided.

2.8 Summary

In this chapter the components relevant for indexing in the Mars core were described. Content is submitted to the Content Submission Service and processed by the Content Processing Subsystem. The annotations are routed to the Content Target Service instances by the Content Distribution Service. The Content Distributor Service informs the generation controller that
Figure 2.9: QR server annotates queries with current generation, if no specific generation was provided by client.

content batches are available for indexing. The generation controller runs two-phase commit with the CTS instances as participants and informs the Query Result servers that a new generation is searchable.

The system today relies on a single node to decide what content to include in each generation, and to coordinate the CTS instances allowing the system to guarantee consistent indexing and consistent search results to the clients. The main problem with this solution is scalability and fault tolerance. If the generation controller fails, no new content is indexed. Also, the generation controller will prove to be a bottleneck as the system grows, both with respect to the number of CTS instances and the amount of content to be indexed. As the number of CTS instances grow, the generation controller needs to receive one vote from each CTS instances for each generation. As the number of CDS instances grow, and the amount of content increases, the generation controller needs to receive a large number of content batch identifiers. The reader might imagine the strain sub-second indexing will impose on the generation controller in large-scale installations, as it receives content information and synchronizes a large number of CTS instances for each generation. For these reasons, a generation controller design providing scalability and better fault tolerance is desired.
Chapter 3

Theory and general issues

In this chapter an overview over relevant theory and the environment the system is operating in is given. The theory is in large based on [7] and [1], but other references are provided where appropriate. The subject of combinatorics and binomial distributions will also referred, as it applies to the new generation controller design.

3.1 Asynchronous versus Synchronous Distributed Systems

In a synchronous system, it is possible to predict the time consumed by any event. In effect, it is possible to predict how much time any task will need to complete. This property makes it possible to deterministically distinguish a failed event from a delayed one. In real life, things are never quite so simple. In general, asynchronous systems can not deterministically distinguish a failed event from a delayed one. Any event can be delayed indefinitely, and in theory it is impossible to guarantee that it has failed.

Why is this important? In a variety of scenarios a system wish to take some action in case of a failure. A common case is for instance for a process to assume the responsibilities of another process if it should fail. In the FAST architecture, the generation controller is one such example. In theory, it is
impossible to guarantee that the other process actually has failed, and the system could end up with two processes assuming the same responsibilities. In certain systems, it is crucial that some task is performed by a single process, and having two processes doing the same task might have severe consequences. Many of the concepts described in this chapter are a direct result of operating in an asynchronous environment.

3.2 Fault Tolerance

In a distributed environment, there are a lot of issues which designers of single-node systems can dismiss. Firstly, resource location must be considered. In a local domain all resources are accessible through the local name space. There is a order of magnitude difference in cost associated with accessing resources, dependent on whether it resides in-memory, on disc etc. In a distributed system, the resource of interest could reside on a remote machine, and accessing it might involve accessing the network, increasing overhead substantially. Furthermore, there are no guarantees that the machine holding our resource of interest is actually running, nor that it is reachable through the network. As distributed systems grow, i.e. number of nodes increase, so does the chance of failures. Failures include broken hardware, power outage, malfunctioning software and network issues. It is important to understand which failures could occur, and what action should be taken when they occur. The appropriate action is potentially different from system to system, and it is absolutely crucial for designers of distributed systems to identify them.

Locally running systems typically halts when failures occurs. For instance, if your processor stops, so does your locally running system. In distributed systems, such a failure should most often not bring the system to a halt, but the system should most likely continue to run, perhaps at reduced capacity. This failure model is often refered to as partial failure. The amount of failures tolerated, or the robustness is often referred to as the level of fault tolerance. Fault tolerance does not come for free, and in a large scale environment there is a need to provide it. It is easy to assume happy days scenarios when designing systems, but in large scale environments, failures are the common case. In the FAST architecture, as the number of CTS instances grow, the
probability that some CTS instance cannot commit the generation increases. As a result of using Two-Phase Commit, the probability of a generation to commit will decrease as the number of CTS instances grow. This indexing should not halt as a result of partial failures, but if possible, continue to run at reduced capacity.

One important and interesting impact of the requirement of good fault tolerance is the idea of no single point of failures. It implies that no single failure should bring the system to halt, and the basic solution is to assure that no resource is only available through a single component. For instance, large data centers typically have multiple power sources. If one power source fails, a second will provide power. It is also common to have multiple network connections, ensuring that the system does not rely on a single network router to be available. Components also include software. No single process should be required to run in order for the system to function.

In the Mars project the generation controller is one example of a single point of failure. In order to index new content, the generation controller needs to be running and accessible through the network.

### 3.3 Timing and Clocks

All computers are equipped with a clock. Clocks tick at a certain rate, and can be adjusted to a certain time. The problem, is that we have no guarantee that two computer clocks tick at the same rate. The difference in the tick rate is referred to as drift rate. As a result, two clocks will over time become less synchronized. Computer clocks in asynchronous system can be synchronized with a bounded limit of accuracy. This limit is in large decided by the network delay. As a consequence, it is impossible to accurately synchronize two events at different computers using the clocks. For this reason, it is generally a good idea to avoid dependence of clocks when designing distributed algorithms.

For instance, any approach where initiating new generations is dependent of each CTS instance doing something a at a specific point in time or anything in that direction is out of the question, it simply will not work.

The drift rates and synchronization issues with clocks also makes it impos-
sible to use computer clocks to correctly decide in what sequence two events at different computers occurs. Leslie Lamport wrote a famous paper [13] describing a scheme for ordering events using logical clocks. In short, some events must occur before other events. For instance, a process replying to a message, must have received the message before sending the reply. Some events can not be ordered. Two independent events at different processes can not be ordered. We refer to this ordering as casual ordering.

3.4 Failures and Failure Detection

Distributed systems are built from a variety of components. These components are subject to human and natural influence. For instance, an operating system is written by humans. It is human to error, and a bug in the operating system can cause your application to crash. Furthermore, careless system administrators yank the wrong network cables, causing your network connection to perish. The hard drive is a mechanical device. The components within experience wear and tear as any other substance on this earth, and will eventually fail. And as a famous bug in Intel’s original pentium floating point unit reminds us [24], the CPU is built by humans and might not execute correctly. Point is, all components are subject to various failures, and designers of distributed systems must realize and take this into account. It is crucial to understand the domain the system operate in, and make sure it handles failures in the desired way. Naturally, all kind of failures cannot be addressed. Designing a system which doesn’t assume that some of the CPUs execute correctly is a tricky one. The most typical failures experienced in traditional distributed systems are now described.

3.4.1 Failures

Host and node related failures A host is considered as a computer and all internal resources needed to run processes on it. It consist of an operating system, processor(s), hard drive(s), memory, etc. These physical devices may break down, effectively crashing the processes running on the host. A node is refered to as a logical entity of software running within one host. Nodes
3.4. Failures and Failure Detection

may crash as a result of software bugs or because of failure of the host they are run on.

It is common to assume fail-stop behavior of nodes. After failure, the node does not execute any instructions. This assumption does not always hold. A software bug could make the application operate arbitrarily, meaning answering requests wrongfully. These kind of failures are known as byzantine failures. It is in general hard to discover and protect a system from such failures, and doing so bears close resemblance to computer security. It usually boils down to being able to trust the service used, which often touches the subjects of authentication etc. In this thesis, correct software and no byzantine failures are assumed.

Network related failures Networks are built using network cables, switches, routers, gateways and a range of different devices all of which may fail. Physical devices break down. A router can get overloaded with work, and simply drop packets in order to cope. Failures in wireless networks are even more common, as data is more easily corrupted. In FAST’s domain, large data centers, wired networks are used.

Hosts communicate over a network, which may fail in various ways. Packets may be duplicated, lost, reordered and even corrupted. These kind of failures are common for Internet-based applications. For communication within a data center, packet loss is usually an indication that there is a fundamental problem with some of the devices.

In networks, temporarily unavailable communication channels are experienced. Hosts in real networks rarely have links to all other hosts. Communication is done via intermittent hosts such as routers, switches etc. If some of these intermittent hosts fail, the effect could be that the network is divided into partitions. Partitions are groups of hosts separated from each other, not being able to communicate between the partitions. This phenomenon is known as network partitioning. Some protocols work well with network partitions, others do not work very well. One of the things that master election protocols should ensure is that after an election only one master exists. Some election protocols does not ensure this in case of network partitioning and each partition end up with one master each.
3.4.2 Failure Detectors

For one node to discover what other nodes are live, the system needs a device for detecting failures. Failure detectors generally work by the failure detector sending alive requests to a set of nodes, which each acknowledges the request. If the request is not acknowledged before some time limit is exceeded, the node is marked as failed, or at least suspected failed. In a synchronous system, it is possible to implement a correct failure detector which can guarantee that nodes that do not reply to live messages have in fact failed. For asynchronous systems however, the failure detector can only suspect that the process has failed, as no guarantees can be made as to how long it should take to acknowledge the alive request.

In asynchronous systems, choosing a clever time-out value is not trivial. If the value is too low, live processes will be suspected unnecessarily. A too high value, will cause unnecessary overhead between the failure of a process and its discovery. If there are big variations in response times, selecting a time-out value is especially hard. Introducing a dynamic time-out value calculated using previous response times is one way of improving the effect of failure detectors.

3.5 Consensus and Master Elections

In distributed systems it is often necessary for nodes to agree on some properties. For instance, it is often useful for one node to play a specific role in the system. All nodes should agree who should assume the responsibilities. Such a role could be to schedule access to some shared resource. Electing a node for this purpose is known as master election. It is crucial that all nodes agree as to which node should be playing this role. Though agreeing on properties at first glance might seem trivial, it can be shown that it is impossible to guarantee consensus in an asynchronous system where nodes may fail [5]. Note that there is a big difference between not being able to guarantee consensus and not being able to reach consensus. There exist a number of algorithms for reaching consensus. The bully algorithm [6] and Paxos [15] [14] are perhaps the most well-known. In Paxos, when some node discovers that the master is missing, it asks the other nodes for its vote. If
votes from more than half the nodes are received, it declares itself master. If multiple nodes are competing, votes are given based on some comparable variable, for instance node identifiers. This algorithm will ensure that there is never more than one master throughout the system, and is considered highly robust. It will even assure one master in case of network partitions as long as one partition includes a majority of the nodes. It functions with up to $N/2 - 1$ failures and is used in amongst others Google’s Chubby Distributed Lock Service [2] and in Chain Replication [23].

3.6 Distributed Transactions

Distributed transactions do operations on several servers, together being an instance of a distributed database. If multiple users access shared data without some sort of synchronization, the data often end up wrong. Two well-known problems with shared access are the lost update problem and inconsistent retrievals. The lost update occurs when some process updates an object which has been previously read by another process. If the object is updated by the latter process, the first update will be overwritten. Inconsistent retrievals occurs when a process reads data which are inconsistent, because of some other process has not committed its transaction.

The ACID properties [9] were introduced in order to allow concurrent access to data, while giving each client an isolated view and keeping data consistent. The following properties must be ensured.

**Atomicity** A transaction either commits or aborts. With distributed transactions, this implies that all servers participating in the transaction either commits or aborts. This requires a commit protocol, discussed in the next section.

**Consistency** A transaction accessesing a consistent database, leaves the database consistent after committing or aborting. This is typically the responsibility of programmer of the transaction, as consistency could be that data conforms to some rules defined outside the database system.
Isolation All clients should get the experience that they are the only one accessing the database. This often involves the use of a locking scheme. Other ways of ensuring isolation also exist. Locking is a pessimistic approach, making sure that there are no conflicts. There are more optimistic schemes, where conflicts may arise and must be handled when they are discovered. These kind of schemes gives less overhead when there are no conflicts, but is more expensive if conflicts are common.

Durability The effects of a committed transaction are never lost. It must be fully recoverable, and survive any failures which the database itself survives.

3.6.1 Two-phase commit protocols

Two-phase commit allows a set of nodes — known as participants — to agree whether to commit or abort a transaction. It is used in distributed databases, where a transaction may have updated objects at a set of different servers. In order to maintain atomicity, it is crucial that the servers agree upon whether to commit or abort the transaction. If any of the participants are unable to commit the transaction, it should be aborted globally. The protocol has two phases. During the first phase a coordinator sends prepare messages to all participants. There are two possible replies to the prepare message. If a participant cannot commit the transaction, it sends abort to the coordinator. If the participant is able to commit, sends commit back to the coordinator. It does not commit the transaction at this point, but waits for a decision from the coordinator. It does however guarantee that it will be able to commit the transaction in case of failures and never abort the generation unless explicitly told to do so. In the second phase, the coordinator based on the replies it has received, makes a decision to commit or abort the transaction. If any participant sent abort or failed to reply to the prepare message, the coordinator sends an abort message to all participants. If commit has been received from all participants, a commit message is sent to all participants.

The protocol is a blocking protocol in the sense that if the coordinator fails between sending prepare messages and sending its decision, the participants remain in an uncertain state, not knowing if they should commit or abort. In
these cases, the participants must wait until the coordinator is restored and can make a decision. The participants can in some cases conclude by asking each other for their state. If any of the participants have received commit from the coordinator, then they should all commit. If some participant have decided to abort, then so can all the others. However, if they are all in uncertain state, they must simply wait.

Two-phase commit have been extensively studied and there exist a large number of variations and improvements. Three-phase commit [19] was introduced to prevent the blocking involved with failure of the coordinator. It uses an extra round of messages to avoid this. During normal operation it is more expensive than two-phase commit. Presumed commit [16] and presumed abort are optimizations where we can reduce the number of forced log writes at the coordinator by only logging committed or aborted transactions. If the coordinator recovers from a failure, and is asked about a transaction which it does not have any stored information of, the transaction is presumed to have committed, or aborted in presumed abort. In traditional 2PC, also known as presumed nothing, all information regarding transactions are stored, and the coordinator can make all the "correct" decisions after recovery. In Circular Two-Phase Commit [12], replication is used for persistence in stead of persistent storage. This may optimize the protocol if accessing the network is faster than accessing your disk. It assumes that the mean time between failures is shorter than the time to recover.

The latency of committing transaction is influenced by the number of disk accesses and the number of communication steps involved with the atomic commit protocol. 2PC ensures that all participants can commit by introducing a voting phase. If the transaction should in most cases be committed, this is tedious and in the general case unnecessary work.

There is not much work focusing on the scalability of 2PC with respect to number of participants. This is probably because there are not many installations in the world today where this is relevant. Many-to-one communication in the scale required by FAST is not very common. Early peer-2-peer systems such as Napster used a centralized component called a "broker", to organize file transfers between peers. Napster peaked in number of users with roughly 13100 users per broker [10]. The broker would only receive look-up requests and chat room messages and such. This number shows an example of a possible limitation for Napster with respect to the number of users one server
could handle. In [10], it is also suggested that Napster tried to link their 90 brokers together, but was unable to as the load between the servers was simply too much to handle.

3.7 Statistics and Probability

This section introduces some statistical concepts concerning distributions and probability estimation. The theory is in large taken from Mathematical Statistics and Data Analysis [18] by John A. Rice.

3.7.1 Probability

When calculating with probabilities, there is a set of rules to follow. The set of possible events is called $\Omega$ and the probability function is denoted $P$. $A_1$ and $A_2$ are disjoint events and subsets of $\Omega$. We have:

\[
P(\Omega) = 1 \\
P(A) \geq 0 \\
P(A_1 \cup A_2) = P(A_1) + P(A_2)
\]

There are also have conditional probabilities. If two events $A_1$ and $A_2$ are depending on each other — not disjoint — the probability of $A_1$, if $A_2$, is

\[
P(A_1|A_2) = \frac{P(A_1 \cap A_2)}{P(A_2)} \quad (3.1)
\]

From this the multiplication rule can be deducted.

\[
P(A_1 \cap A_2) = P(A_1|A_2)P(A_2) \quad (3.2)
\]

An example: There is a basket with 1 blue and 3 red balls. Two balls are picked from the basket. What is the probability that both are red. event $A_1$ is defined as ”pick the first red ball” and event $A_2$ as ”pick the second red ball”. We have from the multiplication law that
3.7. STATISTICS AND PROBABILITY

\[ P(A_1 \cap A_2) = P(A_2|A_1)P(A_1) \] (3.3)

We begin with event \( A_1 \). The probability of picking a red ball \( P(A_1) = \frac{3}{4} \) as 3 of the 4 balls are red. We then calculate \( P(A_2|A_1) \). If \( A_1 \) occurred, there are now 1 blue and 2 red balls left. Thus, \( P(A_2|A_1) = \frac{2}{3} \). Using this in the multiplication rule, we get that

\[ P(A_1 \cap A_2) = \left( \frac{3}{4} \right) \ast \left( \frac{2}{3} \right) \] (3.4)

which equals 1/2. The conclusion is that the chance of picking two red balls is 50%.

3.7.2 Distributions and Confidence Intervals

Two events are \textit{independent} if knowing that one occurred gives no information about whether the other did. A binomial distribution is the probability distribution of the number of successes in a sequence of independent success/failure experiments. Each experiment has the probability \( p \) of success. The number of successes are a binomial distributed random variable \( b(n, p) \) with the number of experiments \( n \) and the probability of success \( p \) as parameters.

Consider a normal 6-sided dice. The chance of getting 6 is 1/6. If the dice is thrown 20 times, the number of 6’s is a binomial random variable \( b(20, 1/6) \).

By the \textit{Central Limit Theorem}, the sum of a large number of independent random variables is approximately normally distributed. From this we deduct that a number of binomial distributed random variables are approximately normally distributed. If the dice is thrown 20 times and the number of 6’s are counted, and this experiment is performed many times, the results will be approximately normally distributed with the mean denoted \( \mu \), approximately 3 as this is the expected outcome.

Also, in a population, people are either below 18 years of age, or above. If a random sample of the population is asked if they are above or below, the result will be a binomial random variable.
In the real world, the true mean is seldom known. However, from a binomial random variable, the true mean can be estimated. The true mean of the distribution is denoted \( \mu \), and \( \sigma \) is the standard deviation. The true mean is unknown, but would in this case exist, because there is a finite number of people below 18, and a finite number above 18. \( \mu \) can be estimated as

\[
\hat{\mu} = \hat{p} \ast n \tag{3.5}
\]

where \( \hat{p} = \text{number of successes} / n \) from the binomial distributed random variable \( b(n, p) \) and

\[
\sigma = n\hat{p}(1 - \hat{p}) \tag{3.6}
\]

We would like to say something about the certainty of the estimation. A confidence interval can be calculated on \( \hat{\mu} \) with some level of confidence. The true mean \( \mu \) should reside within the confidence. We say that confidence interval includes \( \mu \) with a certain degree of confidence.

When calculating a confidence interval, we calculate upper and lower limit values marking the boundaries of the interval. This is for binomial distributions, accomplished by using the formula:

\[
\hat{p} \pm z_{1 - \alpha/2} \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}} \tag{3.7}
\]

where \( z_{1 - \alpha/2} \) is the \( 1 - \alpha/2 \) percentile of the standard normal distribution table (see the Appendix). \( \alpha \) is the desired level of confidence.

When calculating confidence intervals with samples — a subset of our population — the Student’s t-table\(^1\) is used, instead of the standard normal distribution table. The Student’s t-table takes account for the uncertainty of the standard deviation, because of sampling. It says that the z-value is chosen with a certain degrees of freedom, one less than our sample size.

Lets do an example: 101 randomly chosen citizens of Oslo are asked if they are for or against building a new opera house. From these 101, 74 are for and 27 are against. A 95% confidence interval is calculated for the true percentage of citizens who wants the new opera house.

\(^1\)The name Student’s t-table origins from its discoverer, who published it under the pseudonym ”Student”
The observed number of citizens who said yes is called $X$. First we estimate the $\hat{p}$ in favor of the opera house. We have that

$$\hat{p} = \frac{X}{n} \quad (3.8)$$

so $\hat{p} = \frac{74}{101} = 0.7327$. The $z$-value of a 95% confidence interval with 100 degrees of freedom is 1.984. Using the formula for calculating upper and lower bound we get that:

$$\hat{p} \pm 1.984 \sqrt{\frac{\frac{74}{101} \cdot (1 - \frac{74}{101})}{101}} \approx \hat{p} \pm 0.087 \quad (3.9)$$

Thus, with 95% confidence, we can say that the true percentage of citizens in Oslo in favor of the new opera is between 64.6% and 82.0%. Notice that if a larger $n$ was used, the interval would narrow down. Increasing our sample size gives more accurate estimations, and the confidence interval narrows down.

### 3.7.3 Combinatorics

A common task within combinatorics is counting the number of objects that satisfy a certain criteria from some set of possibilities. An example could be: How many different combinations of cards is possible if ten cards are drawn from a 52 card deck? We separate between permutations and combinations. In permutations repetition is allowed — i.e. allowing the same card to be drawn multiple times — as opposed to combinations, where cards may only be drawn once.

The number of combinations to draw $k$ cards from $n$ possible, read ”$n$ choose $k$”, is denoted

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $!$ is the faculty defined as:

$$x! = x \cdot (x-1) \cdot ... \cdot 1$$

An example:
If we want to draw 4 cards from a 52 card deck, what is the chance that
exactly 3 of the 4 cards are spades? In order to solve this, we can combine
combinatorics with conditional probability. First the probability of one com-
bination giving 3 spades and 1 other suit is calculated. One such combination
is to draw: spade, spade, spade, not spade. Using the multiplication rule, we
get that:

\[ P(\text{spade} \cap \text{spade} \cap \text{spade} \cap \text{not spade}) = \frac{13}{52} \times \frac{12}{51} \times \frac{11}{50} \times \frac{39}{49} \]

\[ P(\text{spade} \cap \text{spade} \cap \text{spade} \cap \text{not spade}) = \frac{66924}{6497400} \approx 0.0103 \]

This represents only one way of drawing 3 spades. We need to multiply with
the number of ways we can draw the 3 spades with 4 cards. We then get:

\[ \binom{4}{3} \times P(\text{spade} \cap \text{spade} \cap \text{spade} \cap \text{not spade}) = \frac{4!}{3!(4-3)!} \times 0.0103 \]

\[ \binom{4}{3} \times P(\text{spade} \cap \text{spade} \cap \text{spade} \cap \text{not spade}) = 4 \times 0.0103 = 0.0412 \]

There is approximately a 4% chance of drawing exactly 3 spades when draw-
ing 4 cards from a 52 card deck. If the probability of drawing 0, 1, 2, 3 and
4 spades was calculated, the sum of the probabilities would have been 1, as
it represents all possible outcomes.

It may not be obvious why this approach is correct. To see why, notice that
\( \binom{4}{3} \) is 4. the four combinations are to draw the non spade as the first, second,
third or forth card. The probability for one of those events was calculated,
so we need to take all four events into account. It should be fairly obvious
that if we draw 3 spades and 1 non spade, drawing the non spade as first,
second, third or fourth is equally likely. For sceptics, it is left as an exercise
to calculate the probabilities for all four combinations and see that they are
equal.

Let us complicate things even further. If two poker players draw 5 cards
each. What is the probability that none of the players have more than 2
kings or aces? For instance, 2 kings and no aces satisfy our criteria, but 2
kings and 1 ace or 3 kings and no aces don’t — i.e. we treat kings and aces
equally.

We need to find all combinations where none of the players have 3 or more
kings and aces. We see straight away that the hand of one player affect the
3.7. STATISTICS AND PROBABILITY

hand of the other player. The probabilities for one player to draw less than 3 kings or aces are dependent on what the other player draws.

The approach is to first calculate \( P(0 \cup 1 \cup 2) \), the chance for one player to draw 0, 1, or 2 kings or aces. The probabilities for 3, 4 and 5 will be \( 1 - P(0 \cup 1 \cup 2) \).

The probability for drawing no kings or aces are:

\[
\binom{5}{0} \times \frac{44}{52} \times \frac{43}{51} \times \frac{42}{50} \times \frac{41}{49} \times \frac{40}{48} \approx 0.4179
\]

We list the results for 1, 2 and 3, 4 and 5 without calculations.

<table>
<thead>
<tr>
<th>kings and aces</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.4179</td>
</tr>
<tr>
<td>1</td>
<td>0.4179</td>
</tr>
<tr>
<td>2</td>
<td>0.1427</td>
</tr>
<tr>
<td>3,4,5</td>
<td>0.0216</td>
</tr>
</tbody>
</table>

This is the probability for one player. We now have to calculate the probabilities for the second hand, given the number of kings and aces in the first. We show the calculation, given the first hand drew one of the kings and aces. There are now only 7 kings and aces left, and 47 cards in total. The probability for the second hand to have 2 kings and aces is:

\[
\binom{5}{2} \times \frac{40}{47} \times \frac{39}{46} \times \frac{38}{45} \times \frac{7}{44} \times \frac{6}{43} \approx 0.0135
\]

We give the table with all combined possibilities which meet our criteria. The first hand is represented on each row, while the second hand on each column.

<table>
<thead>
<tr>
<th>Kings and Aces</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1568</td>
<td>0.1792</td>
<td>0.0697</td>
</tr>
<tr>
<td>1</td>
<td>0.1792</td>
<td>0.1742</td>
<td>0.0565</td>
</tr>
<tr>
<td>2</td>
<td>0.0697</td>
<td>0.0565</td>
<td>0.0149</td>
</tr>
</tbody>
</table>
Summing the probabilities gives us a total chance of 95.7\% that no hand has more than 2 kings and aces. Though this example might seem to have little practical value, the reader should note that the table of probabilities will have one dimension for each hand, and each slot requires one calculation. Thus, the calculation has complexity $\text{number of hands} \times \text{cards per hand}$, which will be of relevance in latter stages of this paper.

### 3.8 Summary

This chapter described some concepts in and properties of distributed systems. The statistical theory will be used in the design of Binomial Two-phase Commit described in the next chapter.
Chapter 4

Design

In this chapter a new generation controller design is described. The generation controller assures that content batches are made visible in the search index atomically, by including them in generations and using 2PC to make sure all CTS instances are capable of offering the generation. When all CTS instances are able to offer the generation, the generation is committed and the QR servers are notified of the new generation. The QR servers can now annotate queries with the new generation identifier.

There are three apparent reasons to finding alternative solutions to the current design. Firstly, relying on a single node to be available in order to index new content is very fault intolerant. If it fails, no new content will be indexed. Secondly, The generation controller uses 2PC to assure that CTS instances can commit the generation. This requires the generation controller to handle votes from all the CTS instances, causing scalability issues as the system grows. As mentioned, in AllTheWeb FAST experienced that when there are more than 200 nodes communicating with a single node, network traffic becomes an issue. Finally, all CDS instances will have to forward information regarding the content batches to be indexed to the generation controller, again causing scalability issues as the number of CDS instances and content load increases.

In this chapter, the first concern is replacing 2PC. A description of Binomial Two-Phase Commit (B2PC) is given. The name comes from the fact that
CHAPTER 4. DESIGN

after the prepare phase, there will be an unknown, but set, fraction of the CTS instances capable of committing. This fraction is estimated using the properties of binomial distributions described in section 3.7.2. Using this approach, the fraction of the system capable of committing is estimated, asking only a subset of the CTS instances for their vote. Some other alternatives to 2PC, as Ring-based 2PC and hierarchical 2PC is also addressed.

After offering alternatives to 2PC, ways to create a distributed component are proposed. The goal is to increase fault tolerance and allow scaling of content load. The number of CDS instances should, as the CTS instances, be able to scale in the order of thousands, so the goal is to reach a component able to handle the content load installations of this size would generate.

4.1 Replacing 2PC

The motivation behind replacing 2PC, is to use a cheaper protocol when there are a large number of CTS instances. B2PC is not intended for installations with less than 200 CTS instances, as some portion of nodes is needed to make sensible estimations. Also, the overhead of regular 2PC in systems with less than 200 nodes should be within the capacities of a single node to handle. Another motivation is to allow a degree of failures with our CTS nodes. In environments with possibly thousands of CTS instances, it will rarely be experienced that all are functioning as advertised. In regular 2PC, generations are aborted if one of the nodes fails to reply.

Using Binomial Two-phase Commit allows scaling of participants, but the strong guarantees offered by regular 2PC are not maintained, and guaranteeing that all participants are able to commit is no longer possible. A description of the protocol follows and then some alternatives.

4.1.1 Binomial Two-phase Commit (B2PC)

The very general idea behind B2PC is that if the system don’t need to guarantee that all CTS instances can commit, but rather that there is at least a certain portion of them able to commit. A subset of the CTS instances can
be asked and from these replies an estimation of the total amount of CTS instances able to commit is possible. This can save the overhead of having to relate to every individual "opinion", and the system can just relate to what the general trend in the total population of CTS instances (probably) is. In order to make good estimations, the subset of voters needs to be selected randomly.

Selecting the number of voters is different. There is no simple answer to how many of the CTS instances should be selected as voters. It depends on the accuracy needed in the estimation, and the overhead involved with each vote. We suggest that the number should be at least 100, but the more votes received the more accuracy the estimation will have. For this reason we will for now not say that the protocol requires some certain number of voters, but the reader can for now imagine that there are more than 100 voters.

With 2PC, a generation controller commits if all CTS instances have voted in favor. With B2PC the decision to commit is a bit more complicated. The overall goal is to assure atomicity. In practice, this means that all partitions in the system should be able to offer the new generation. Thus, there should be at least one member of each partition capable of committing. However, a large scale system would most certainly require more than one CTS instance in each partition to commit. Atomicity in itself is not enough, the system also needs to handle the subsequent query load. The approach is to calculate a minimum number of CTS instances able to commit, in order to be fairly sure that atomicity and the capacity to handle queries are maintained. Calculating this threshold is dependent on what the system looks like with respect to number of partitions and replicas. The technique for threshold calculation is given in Section 4.1.3 and some practical examples are presented in Section 5.3.

The protocol begins with the generation controller broadcasting prepare messages to all CTS instances, as with the original 2PC. Multicast possibilities are assumed, for instance ip-multicast or some overlay-multicast. As the application is thought to operate in a closed data center, ip-multicast should be provided. Using multicast, we assume that there is one packet, normally received by all participants.

A generic state diagram for the coordinator in B2PC is now presented. A coordinator is responsible for the estimation. The coordinator is the notion
of the centralized component in B2PC as a standalone commit protocol. The coordinator is the equivalent of the generation controller in our specific system. Participants are the equivalent of CTS instances and are responsible for returning votes in favor or against committing.

In figure 4.1 the coordinator has four different states. The initial state of the coordinator is "Ready". The coordinator sends prepare messages to all participants, asks for votes from a subset of the participants in return and enters "Receive votes". The coordinator waits until it has received votes from all the voters or until timeout, whatever occurs first. It then makes a decision to commit or abort the transaction, and sends its decision to all participants. "Commit" and "Abort" states mark that the protocol has finished.

Even though all participants are asked to prepare the commit, only a subset — referred to as voters — are asked to return a vote. This subset of participants is chosen randomly. Information regarding the identity of the voters may be piggybacked on the prepare message, or two different multicast messages can be sent. Each participant decides whether it is capable to commit the transaction, and the voters return their vote. In traditional 2PC, if a participant decides to abort the transaction, it knows that the transaction will be aborted globally. Thus, it can locally decide to abort the transaction. This no longer holds. We wish to loosen the requirement of all participants being able to commit, and hence may wish to commit the transaction even if there are participants that cannot commit. A new state called
"Failed" is introduced. Participants that cannot commit should enter this state. Participants that fail to commit the transaction, cannot do so because they have some sort of problem, and must locally decide what to do in order to become functional again. This will usually involve some kind of recovery procedures and possibly the need for human intervention. For instance, in our specific system, if a CTS instance is experiencing some hardware failure, an administrator needs to replace the broken hardware.

Figure 4.2: State diagram for participants in B2PC.

Figure 4.2 shows the states for one round of B2PC for participants. There are five states; "Ready", "Prepared", "Failed", "Abort" and "Commit". A participant may locally decide to fail at any time, and this decision does not imply that other participants should abort or fail. If the participant is a voter, it sends its vote to the coordinator. It can vote for "commit" or "abort". If a participant wants to abort, it enters the state "Failed", and must initiate the appropriate recovery procedure in order to resume normal operation.

In the second phase of the protocol, the coordinator uses the received votes to estimate a percentage of the participants capable of committing. This is done using the properties of binomial distributions and confidence intervals discussed in section 3.7.2. A certain percentage of the participants that can commit, can be estimated with a certain degree of confidence. If the estimated percentage of participants able to commit satisfies some threshold, the
coordinator proceeds with multicasting "commit" to all participants. Deciding what the threshold should be is not trivial. An approach is discussed in Section 4.1.3. Participants that prepared the new generation will upon receipt of the commit message go ahead and commit the transaction. Those not able to commit at this stage will if they haven't already, enter the state "Failed".

In regular 2PC, when a participant votes for commit, it also guarantees that it is capable of commiting the transaction, regardless of survivable failures it experiences thereafter. This basically involves storing enough state on persistent storage to commit in case of power outages. The overhead involved with writing to disk is in this case avoidable. The protocol itself does not offer the strong guarantees of 2PC with respect to commitability, so requiring the participants to do so does not really give us anything. As a result of estimation, there will be some uncertainty involved with the decision to commit the transaction anyway.

4.1.2 Using B2PC in Mars

For the generation controller, selecting the number of voters depends different considerations. By selecting more voters, the estimation will be more accurate, but the overhead of votes increases. The frequency of new generations and the amount of votes the generation controller is able to handle per generation is important.

When a CTS node goes into recovery, it needs to get its index up to date. This can be accomplished by retrieving completed generations from its replicas, while receiving new annotations, not yet part of a generation, from the CDS. When it has received all annotations and restored its index to the last completed generation, it can start to vote in favor of new generations, and commit them according to normal protocol behavior. When the generation controller selects voters, it should select from the entire pool of CTS instances, even the failed. CTS instances in recovery should continue to reply to prepare messages, as they are statistically important for the generation controller. To minimize the latency of the voting, CTS instances should if they can, reply even though they are already in failed state. Waiting for timeouts is expensive, so a negative vote is if possible, desired as opposed to
timeouts. If a CTS instance has failed entirely, the vote will time out at the generation controller.

### 4.1.3 Estimating thresholds

When estimating a percentage of the participants capable of committing the generation, the notion of partitions is not taken into account. When selecting a threshold, the number of partitions and replicas must be accounted for. The goal is amongst others, to ensure atomicity. This means that all partitions should have a satisfactory degree of commitability. Therefore, something must be said about the chance for one or more partitions to have a too big share of failures in order for the generation controller to risk committing the generation. To illustrate the point consider the following.

In an imaginary system consisting of 1000 CTS instances organized in 10 partitions, implying 100 CTS instances in each partition, the generation controller multicasts prepare to all partitions and ask 101 of them to return their vote. From these 101 replies, 95 commit-votes and 6 abort-votes are returned.

We chose a 99.95% confidence level. This implies that we have 99.95% confidence in the estimated interval. In practice, once every two thousand estimation, the true number of committable CTS instances will not be included in our interval.

The generation controller can then, using the approach presented in section 3.7.2, estimate the lower boundary for the number of committable CTS instances. The $z$-value for a 99.95% confidence interval with 100 degrees of freedom is 3.390 (see Section 7.3 for table).

\[
\hat{p} - 3.390 \times \sqrt{\frac{95/101 \times (1 - 95/101)}{101}} \approx \hat{p} - 0.080 \quad (4.1)
\]

\[
\hat{p} = 95/101 \approx 0.94 \quad (4.2)
\]
Thus, the estimated fraction of participants capable of committing is approximately 94%, and the lower boundary of our confidence interval is approximately 86%. The generation controller concludes that with 99.95% confidence, the true number of committable CTS instances is above 860.

As an example, let the system require that there is no more than 10 failed CTS instances in each partition. From the estimation above, the generation controller found that there are 860 CTS instances capable of committing the generation. It should then calculate the probability that one of our partitions has more than 10 failed nodes when there are 860 CTS instances capable of committing. When doing this the generation controller needs to assume that nodes fail independently of the partition they belong to. In practice, this is a very strong assumption. It means for instance that entire partitions should not be deployed behind single network switches. If that switch failed, the CTS instances behind it would fail as a result of belonging to that partition and our assumption would not hold. However, one would imagine that for fault tolerance in general, systems should be deployed in a way that ensures that failures are independent of partitions.

It is possible to use conditional probabilities and combinatorics to calculate the probability of one or more partitions having more than a percentage of failures. It would then be possible to say that a generation should not be committed if there is for instance, more than 0.01 percent chance that one or more of the partitions have more than 10 failures.

As mentioned in Section 3.7.3, this equation grows exponentially. It has the complexity of number of partitions in the power of the number of failed nodes accepted in each partition. Doing this in real time is impossible. There is typically a large number of partitions and/or participants in each partition — which is why we are doing this in the first place — so computing this accurately in real time, is simply not feasible. With 10 partitions and allowing 10 failed in each partition, the generation controller needs to calculate $10^{10}$ probability calculations, each involving factorial computations with relatively large numbers. To our luck, there are simpler approaches.

The computation can be solved by estimation. If the maximum number of failures accepted in a partition is known, the fail frequency for partitions can be estimated using sampling. By taking multiple samples where failures are "placed" randomly in imaginary partitions, the fraction of samples containing
a satisfactory degree of failures in each partition can be extracted. This will not be totally accurate, but by doing larger number of samples increased accuracy is achieved. This estimation technique can be applied once and then the estimated threshold may be used in realtime. Examples of threshold estimation are given in Section 5.3.

### 4.1.4 Other alternatives to 2PC and B2PC

B2PC is a protocol aimed at increasing the scalability of 2PC, trading off the possibility to guarantee that all participants can commit the transaction. If the goal is merely to make 2PC more scalable with respect to number of participants, there are some alternatives. One naive way of solving the scalability problem, is to organize the participants in a logical ring. The coordinator may then multicast the prepare message as normal, and have the participants send one vote along the ring. A node may only vote for commit if it is able to commit and has received a commit-vote from its predecessor. The vote is then sent to its successor in the ring, until it reaches the coordinator. If some node is unable to commit it can instantly send an abort-vote to the coordinator, and the coordinator will instantly abort the transaction globally. This approach would result in a single vote to be received at the coordinator. The protocol would however scale badly with respect to the time to complete, as the votes are sequentially processed at each participant. Multiple logical rings operating in parallel would decrease the time to finish, while requiring the coordinator to handle multiple votes. This approach allows a trade off between the number of votes to process per transaction and the time to complete the commit protocol. The strong guarantees of 2PC with respect to atomicity are also maintained.

Another approach would be to use hierarchical two-phase commit. The protocol allows the voting to be done in levels. A coordinator sends prepare messages to a set of sub-coordinators. These send prepare messages to its share of participants. The participants acts as normal 2PC participants, but the sub-coordinators send their vote to the main coordinator based on the participants votes. The main coordinator makes a global decision, and the decision is propagated to the participants. This protocol scales well with respect to the number of participants each coordinator must handle, but one round of 2PC is carried out for each level of coordinators. The approach
trades off indexing latency for reduced number of messages needed to be received by the generation controller.

4.2 Distributing the Generation Controller

In order to create a distributed generation controller, the three main issues with the component are addressed. The first issue is the scalability problem of two-phase commit as the number of participants grows. For this part, using Binomial Two-Phase Commit as presented above, is proposed as it trades off the possibility to make deterministic decisions regarding commits, and not indexing latency as the other two approaches presented. Scalability of content load and achieving a higher level of fault tolerance, the other two issues with the current generation controller design are now addressed.

![State diagram for generation controller using B2PC.](image)

Figure 4.3: State diagram for generation controller using B2PC.

In figure 4.3 a state diagram for a B2PC-based generation controller instance is presented. In the initial state, the generation controller selects batches to include in the next generation. The component should queue new batch
identifiers as they arrive at the generation controller until they are selected for inclusion.

When the GC decides to create a new generation, it runs the B2PC protocol as described. When reaching the state ”Commit”, it sends notifications to the QR servers that a new generation is available and the identifier associated with it. It then sends an acknowledgement informing the CDS that the batches were successfully indexed. If the GC reaches the state ”Abort”, it does not notify the QR servers, but informs the CDS that something went wrong with the indexing, and that the batches must be resubmitted if they should be indexed at a later stage.

Figure 4.3 shows how a single generation controller instance will operate. Approaches for creating a distributed generation coordination scheme is now presented. A distributed component will hopefully increase the systems content load capacity and fault tolerance.

The CDS instances should be able to speak to different generation controller nodes. If this is achieved, the number of CDS instances should scale by increasing the number of concurrently running generation controller nodes. A single node that receives content batch identifiers from all CDS instances could then be avoided. In addition, sharing the responsibilities of the generation controller might provide better fault tolerance, because if one of them fails, generation coordination could possibly be continued by the other generation controller instances.

Distributing the generation controller offers a range of design choices. Firstly, should the system stick with a single generation concept? One could imagine multiple generation controllers operating in parallel each on one generation concept. CDS instances would deliver content batches to instances of the CTS as normal, and send information about the content batches to one generation controller instance for inclusion in its generation concept. Queries would have to be annotated with all generation identifiers, one for each concept. This would have some consequences for the content target service. The CTS is designed for one generation concept, a decision made because consistency of the index is easily maintained.

If a single concept of generations should be maintained, another range of issues have to be addressed. Where is the decision regarding what content
batches a generation should include made? A master amongst a set of generation controller instances could decide. This node would in one way or another need to receive all the information regarding content batches from all generation controller nodes, so the gain in scalability of content load might not improve much. The fail-over time could possibly be reduced by having a set of nodes ready to take over operation in case of failures. A new master election would have to be run and the winner would possibly need to gather its predecessors state, depending on the chosen replication strategy.

For master elections, Paxos is proposed. Paxos is a consensus algorithm, where there are three types of agents. Proposers propose new values to agree on, in this case the identity of the new master together with a proposal number. Acceptors accept proposals and promises to never accept new proposals with lower proposal numbers. If a proposer gets a majority of acceptances for its proposal, it won the election and informs the learners of the elected master. In our case the generation controller instances would play the roles of all three agents. The algorithm ensures that a single value is agreed upon as a proposer needs a majority of acceptors to accept the value in order to win the election. For details see [14] and [15].

### 4.2.1 One generation concept

In this section designs keeping the notion of a single global generation concept are presented. The concept of a generation can be thought of as a shared resource. If a set of concurrently running generation controllers instances is introduced, the generation must be accessed in a way that enforces the ACID properties. If this is not enforced, the system may experience that a generation does not include the same content system wide.

In Mars, a single generation concept is assumed, and our solution will also keep the notion of a single generation concept.

**Master-based generation coordination**

The intuitive and our very first thought was to run a master generation controller. The master decides what content to include in each generation.
A set of subordinate generation controllers receive content batch identifiers from the CDS instances, and ask the master node to include the content in the next generation. This would allow CDS instances to relate to different generation controller instances. However, the information regarding content batches would need to be forwarded to the master generation controller at some point and would probably not offer much with respect to scalability of indexing capacity.

If the master fails, a round of master election is needed as well as recovery of the generation in progress as with the current generation controller design. The solution is still a single-node solution with respect to fault tolerance.

Since the master would need to know about all content batches, thus not offering us scalability of content load, Token-based generation coordination is believed to be a better solution.

**Token-based generation coordination**

The reason Token-based generation coordination was proposed is because it distributes the responsibility of initiating new generations between a set of processes. However, the generation controller instances cannot initiate new generations in parallel, so the gain with respect to fault tolerance and indexing capacity might be questionable. The solution trades off indexing latency. This is not optimal, as indexing latency is perhaps the one property the system needs to maintain. The solution is explored, as no other clear cut alternatives stand out.

The token-based approach, keeps the notion of a single global generation concept. It uses a set of equal generation controllers organized in a logical ring as seen in Figure 4.4. A unique generation controller identifier is assumed for all generation controller instances. A token is passed along the ring, and only the generation controller instance holding the token may initiate new generations. This approach would allow CDS instances to send batch identifiers to any generation controller instance, but the generation controller would have to wait for the token before including the content in a new generation.

One of the members is elected master and is responsible for maintaining the ring as members join and fail. If the token is lost, the master is also
responsible for initiating a new token.

![Figure 4.4: Generation Controllers organized in a ring. One is elected master and is responsible for keeping the circle consistent in case of failures. The token is passed round the ring and the token counter is increased for each step.](image)

**Failure actions with Token-based coordination**

If one of the generation controller instances fails, it could either be waiting for, or have the token. If it was waiting for the token, it would simply be excluded from the ring, and the CDS would be notified and have to resend the lost batch identifiers to another generation controller instance. If it is in possession of the token, the token is lost. An easy way to discover a missing token is to have a incremented counter for each time the token is passed. The master will periodically send alive requests to the members of the ring. If one of the members fails to reply, it is suspected failed. The master would then ask the successor and predecessor of the failed member for the last token counter they saw. If the predecessor has seen a larger counter than the successor, the token is considered lost.

The generation controllers would have to know what the ring looks like. If
4.2. DISTRIBUTING THE GENERATION CONTROLLER

new instances want to join, they contact the master. The master places the
generation controller instance in the ring and sends it information regarding
the shape of the ring. If the master fails, a new must be elected and the
new master must make sure that the ring is in consistent shape. When a
geneneration controller instance fails, the master would notify the predecessor
and the successor of the failed instance that they are now neighbors. They
should respond to this notification with the latest token counter they seen.
The master can use the token counters to discover a lost token. If the pre-
decoessor has seen a larger token than the successor, the token is considered
lost. If the successor just joined the ring and has never seen a token, it will
need to ask its own successor for the latest token identifier it has seen.

In addition, there is no guarantee that the token is actually is lost. The failed
geneneration controller could for instance just be slow. For this reason the mas-
ter generation controller needs to make sure that the old token doesn’t come
back in rotation. The master should for this reason notify the generation
controllers that the old token is no longer valid. With each token there
should be a unique token identifier intended for this use. When the master
initiates a new token, it can combine its generation controller identifier and
a timestamp to generate the token identifier.

If the master fails, the other generation controllers would elect a new master
using Paxos. The new master asks all generation controller instances to
return information regarding their neighbors in the ring and the last token
counter they saw. From this, the new master can discover inconsistencies in
the ring structure, and find out if the token was lost together with the old
master.

When a generation controller fails, it can be in any of the states shown
in Figure 4.3. If the generation controller did not hold the token, it must
be in "Receive batchids", since it cannot start a new generation without
the token. The batch identifiers lost with the failed generation controller
instance, should be resubmitted to a different generation controller instance.
If the failed generation controller did hold the token, it could have failed in
any state. Thus, it could for instance have committed a generation without
acknowledging the CDS yet. The generation controller instance initiating
a new token must figure out what state the failed instance was in. If a
generation has been committed, the batches should not be resubmitted as
they are already in the index. We also need to figure out what the last
committed generation is in order to initiate new generations correctly.

Figure 4.5: Master generation controller initiates a new token, as the failed generation controller held the token as it failed. 1) The GC requests state information from the CTS instances. 2) The GC decides if there is need to commit or abort a pending generation and sends the decision to the CTS instances. 3) The QR servers are notified of what the newest generation is. 4) The GC acknowledges batches to the CDS instances, and/or asks them to resubmit unacknowledged batches to another generation controller instance. 5) A new token is initiated using the knowledge of the last committed generation.

This is done, as seen in figure 4.5, by gathering state from the CTS instances. The master needs to know what the latest committed or uncommitted generation is, and what batches the generation contains. If there are CTS instance in uncertain state the master needs to commit or abort the generation that the generation controller was working on. If any of the other CTS instances have committed the generation so should all the uncertain and likewise if anyone has aborted — but not failed — the generation should be aborted. If no decision was made, the master can make a decision based on the information from the CTS instances and complete the generation according to protocol. When the master is sure that the CTS instances are consistent again, it can inform the QR servers of the latest generation, and acknowledges the content batches to the CDS. The master now knows what the last completed generation is, and can generate a new token and resume operation.
Notice that this recovery protocol relies on talking to all CTS and CDS instances. If a single node does this recovery procedure, it will need to communicate with all of them. This communication can be distributed. Each generation controller instance could talk to a set of CTS instances and aggregate the data from them. From the CTS instances the master generation controller instance only needs to know if anyone has committed, aborted, or are uncertain. If each generation controller aggregates this information and forwards it to the master generation controller, it can make the final decision regarding the incomplete generation and finish the recovery protocol. If all CTS instances are uncertain, the master needs to make a decision. The aggregation would have to be done in a way that allows the master generation to know how many of the CTS instances are in uncertain state, and how many have failed, so it can make a sensible estimation and decide what the outcome is.

Notice that as a result of operating in an asynchronous environment, the master cannot guarantee that the generation controller actually did fail. For this reason, the CTS instances should ignore any messages from the suspected failed generation controller after they have been asked to return their state to the master generation controller.

### 4.2.2 Multiple generation concepts

Another approach is to allow multiple concurrently evolving generation concepts in the index. Each generation controller instance creates new generations within its own *generation concept*. A generation controller instance would receive batch identifiers and include them in generations as with a single generation concept. The QR would receive generation notifications from each generation concept and would need to keep track of them independently. Queries would have to be annotated with generation identifiers from all generation concepts. The CTS instances would have to support an index that doesn’t evolve in a sequentially manner like with one generation concept, but more like a tree where each branch of a generation concept evolves independently. To illustrate, imagine two generation controllers working on separate generation concepts. As the generation controller instances would work independently we have no control over what sequence the generations are committed in at different CTS instances, nor notified to the QR servers.
This implies that the search index would no longer evolve as a single global sequence of new generations.

In figure 4.6 the CTS first sees the commit from generation controller instance 1 and then the commit from instance 2. The query result server sees the second instance’s generation first. Because of this, the QR may annotate a query with two generation identifiers out of sequence from the CTS’s perspective. The CTS would have to support that the index does not evolve strictly sequentially. In today’s design, the search index is not capable of handling this, as it expects a single sequentially growing generation concept.

Another problem with this approach would be that as a document changes and is re-indexed, it could be come part of a different generation concept. This would possibly lead to multiple versions of the document in the search results, and would have to be addressed in the index at CTS level. One way to avoid this would be to route content into a specific generation concept in the same manner as annotations are routed to CTS instances. However, this would result in different documents in the same content batch being part of different generation concepts, and the atomicity property which generations
are meant to enforce in the first place, would be broken. It seems that a solution with multiple generation concepts would require a lot of changes to the way indexing is done as at the CTS.

Another issue is what should be done in case of failures. If the CDS should resend the lost batches to another instance, we risk indexing the same content more than once. The best approach would perhaps be to do recovery of the lost generation similar to the approach used in today’s design. The CDS instances that have sent batches to the failed generation controller instance would then have to wait until recovery finishes before resubmitting unacknowledged content batches.

Because the idea of multiple generation concepts is not feasible with the existing index structure, token-based generation coordination is chosen in this design. The idea of multiple generation concepts was described as the idea might be worth while for FAST to consider.

4.3 Complete generation controller design

Using B2PC and Token-based generation coordination, a complete view of the system and the new generation controller design is presented. B2PC was chosen as it scales well as number of participants grow, and does not trade off indexing latency. Token-based generation coordination was chosen in order to scale the systems total indexing capacity.

In Figure 4.7 an overview of the system is given. The Content Submission Service and the Content Processing Subsystem are excluded. The CDS forwards its annotations to the CTS and sends content batch identifiers to one of the generation controller instances. The generation controller instances pass a token between themselves, and whoever holds the token may chose to create a new generation. It runs 2BPC with the CTS and notifies the QR servers that a new generation is available. The generation will at this point be included in new search results.
Figure 4.7: System overview with new generation controller design.

4.3.1 Purpose

The purpose of the new generation controller design is to maintain the functionality of the old generation controller design, while improving scalability. Content batches are still added to generations in a consistent way, allowing clients to trust that search results represent the indexed content correctly. In addition, the purpose of the new generation controller design is to ensure atomic indexing while reducing the amount of communication with the CTS instances. In terms of fault tolerance, the token-based generation coordination is meant to decrease the amount of annotations needed to be resubmitted by the CDS in case of failures.

4.3.2 GC Composition

By using token-based generation coordination, the overhead of passing the token between initiations of generations is introduced. The generation controllers do no longer have full control over when it can initiate a new generation, as it will have to wait for the token. The gain is first and foremost that in failure situations, only the content batches sent to the failed generation controller will have to be resubmitted. With a single generation controller instance, all content batches will have to be resubmitted. The solution will
increase the indexing latency as a result of the token passing. For fairness, it would be reasonable to only allow one generation to be initiated each time a generation controller sees the token.

The difference between this system as opposed to the original design, is for one that the number of votes the generation controller needs to handle is restricted. This is accomplished by trading off the possibility to deterministically decide whether a generation should be committed or not. In effect, the generation controller can no longer guarantee that content is indexed atomically. In the next chapter the probabilities for breaking the atomicity property is shown.

All messages between the generation controllers themselves, the CDS, CTS and the QR servers are control traffic of low volume. As the generation controller instances receive content batch identifiers from the CDS instances, they will queue them in a separate queue for each CDS instance as with the original design. The scheduling of content batches into new generations will be done identically to the scheduling in the old design. However, it will be harder for the generation controller to delay batches, as it does not know how long it will wait for the token to come back.

The biggest messages will be the multicasted prepare message, as it will contain possibly hundreds of voter identifications and a number of content batch identifiers. Assuming that a 4 byte node identifier is used, the identifications would be at least 400 bytes with one hundred voters, 800 bytes with two hundred voters and so on. In addition, this message will contain the content batch identifiers for the specific generation.

Assuming 4 bytes for each of the serviceinstanceid, sessionid and batchid (see Section 2.2.3 for identification of content batches), and 500 voters — giving plenty of estimation accuracy — within a maximum gigabit ethernet frame size of 8992 bytes [11], there should be room for more than 500 content batches in the prepare message.

Commit and abort messages are small control messages, containing only generation identifiers. The token contains a token identifier, a token counter and the last committed generation identifier. The token counter should be an 64-bit integer. In practice, a 64-bit token counter would last forever. The votes from the CTS instances are also small control messages and will fit in
the minimum gigabit frame size of 512 bytes.

### 4.3.3 Operation

![Sequence diagram for generation controller interaction.](image)

Figure 4.8: Sequence diagram for generation controller interaction.

Figure 4.8 shows how the generation controller instances interact with each other, the CDS, CTS and QR servers. The two CDS instances deliver their content batch identifiers to one of the generation controllers. While holding the token, the second generation controller instance initiates a new generation using B2PC. When completed, it notifies the QR servers, and acknowledges the content batches in the generation.

Figure 4.9 shows an example of how B2PC would be carried out between a generation controller instance and the CTS. The generation controller multicasts prepare and asks for votes from four of the CTS instances. Two votes for commit, one for abort and one of the instances fails before sending its vote. The failed CTS instance’s vote times out at the generation controller. The generation controller makes an estimation of the number of CTS instances that can commit. Apparently, the threshold was not exceeded and
the generation controller multicasts commit to all CTS instances. The first CTS instance does not commit the transaction, as it voted for abort, it will enter recovery.

The solution will be more efficient in terms of computing, as the generation controllers can prepare itself for initiating the next generation while not holding the token. The work of selecting the appropriate content batches for inclusion and receiving batches will be done in parallel at all generation controller instances, minimizing the preparation time when the token actually arrives.

4.4 Summary

In this chapter B2PC was described, token-based generation coordination and a description of how the system would operate using these two concepts was also presented. Some alternative solutions to regular 2PC and B2PC were explored, and also alternatives for a distributed generation coordination
scheme.

In the next chapter the design is analyzed and the specific properties offered by the solution with respect to scalability of content load, CDS- and CTS instances is examined. In addition, the level of fault tolerance offered by the solution compared with the original design is explored.
Chapter 5

Analysis

In this chapter the properties of each alternative presented in chapter 4 are analyzed. The chapter begins with analyzing 2PC and the alternatives described in Section 4. The token-based approach to distribution is compared to a single node solution in terms of fault tolerance and content load.

5.1 Performance analysis of commit protocols

In this section the commit protocols with respect to the total number messages generated in the system, the time to complete and the number of messages the coordinator needs to handle per round are analyzed.

5.1.1 2PC

In 2PC, the total number of messages is dependent on the number of participants. From the generation controllers point of view, there are one message for multicasting a prepare message, one message for multicasting commit or abort and one message for each participant’s vote. Thus, the total number of messages seen by the generation controller is number of participants plus
two. The multicasted messages will fan out to one message per participant. Hence, the total number of messages generated in the system is 3 times the number of participants. The total number of messages in the system scales with $O(n)$ with respect to participants.

The time to complete the protocol is constant at three rounds, one for multicasting \textit{commit}, one for the votes and one for multicasting the decision.

The number of messages received at the coordinator equals the number of votes, or the number of participants. Thus, the number of votes the coordinator needs to receive scales with $O(n)$ with respect to increase in participants.

### 5.1.2 B2PC

The number of messages sent by one round of B2PC depends on the number of voters selected. With estimation in normal distributions, the accuracy is dependent on the number of samples chosen and not the population. Recall the equation from Section 3.7 used for estimating a confidence interval.

\[
\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}
\]  

(5.1)

There is no notion of population size, only the number of samples. For this reason, the accuracy of the estimation is dependent on the number of voters, and not the number of participants. In other words, the accuracy of the estimations does not deteriorate as the number of CTS instances increases (also see Section 5.2).

From the generation controllers perspective, there is one message for multicasting \textit{prepare}, $n$ messages in the votes and a single message for multicasting \textit{commit} or \textit{abort}. The total number of messages from the generation controllers point of view, is number of voters plus two. System wide, the multicasted messages fan out to one message for each participant. Thus, the total number of messages system wide equals 2 times the number of participants plus the number of voters.
The number of rounds to complete the algorithm is the same as with regular 2PC, one round sending *prepare*, one for the votes and one for *commit* or *abort*. Hence, the number of messages from the generation controllers perspective in B2PC scales with \( O(1) \) with respect to increase in participants, \( O(n) \) with respect to increase in number of voters and the completion time is three rounds. The number of messages in the entire system scales with \( O(n) \) with respect to participants, as a result of multicasted messages fanning out to one message for each participant.

The number of messages received at the coordinator equals the number of voters. The number of votes the receiver needs to be able to handle scales with \( O(1) \) with respect to increase in participants as the accuracy of estimations is not dependent on population size.

![Comparison between 2PC and B2PC in respect to total number of messages](image)

Figure 5.1: Total number of messages from the generation controllers point of view, as a function of number of CTS instances.

Figure 5.1 shows how the total number of messages generated by 2PC versus B2PC from the generation controllers perspective. As seen, the total number of messages generated by B2PC is constant. In the figure B2PC assumes 500 voters. For the number of messages which the generation controller needs to receive, the trend will look the same.

The tradeoffs between B2PC and regular 2PC is that the possibility to deterministically decide what state all our participants are in is given up. In 2PC, the coordinator knows the state of all participants and can make *safe*
decisions whether to commit or not. In B2PC, the coordinator can only make probable safe decisions regarding committing. In return the coordinator gets a bounded limit of messages that it needs to handle independent of the system size. The time to complete both algorithms is still three rounds.

5.1.3 Ring-based 2PC

With Ring-based 2PC, there is one message for multicasting \textit{prepare}, and then each participant sends one vote each along the ring, until the last participant sends the final vote to the coordinator. Finally there is one message for the decision. From the generation controllers point of view, the total number of messages equals three. The two multicasted messages and the final vote received. The total number of messages system wide equals number of participants times three, and the number of messages system wide therefore scales with $O(n)$ with respect to increase in participants.

The time to complete is one round for multicasting the \textit{prepare} message, then one round for each participant as the vote is propagated through the ring and finally one round for multicasting the decision. The time to complete is number of participants plus two, which scales with $O(n)$. This implies that the indexing latency will be directly be affected by the size of our system, as the votes need to be propagated through the ring before committing the generation.

The number of messages that the coordinator needs to handle is one, as the last participant in the ring is the only one sending a vote directly to the coordinator.

With Ring-based 2PC, time to complete the protocol is traded off for a single message that the coordinator needs to handle. The total number of messages is the same as in regular 2PC.

Also, the possibility to deterministically decide the outcome of the protocol as long as all participants can commit is maintained. The coordinator can not make a decision to commit if there are some failures, as with regular 2PC and B2PC, as the coordinator only knows if all can commit of if someone cannot commit. Note, that with regular 2PC in our specific FAST domain, the coordinator knows exactly which nodes can commit and which voted for
abort. Because of this, it would be possible, with some modifications to the participant part of the protocol, to decide to commit in case of failures, as the generation controller knows the state of all CTS instances in the system.

5.1.4 Hierarchical 2PC

In hierarchical 2PC, there is one multicast message for prepare, and another for commit. These messages are received by all participants and sub-coordinators. In addition there is one vote for each participant, and one vote for each sub-coordinator. The generation controller only receives votes from the sub-coordinators on the level below the generation controller. Thus, from the generation controller point of view, the total number of messages equals the number of sub-coordinators in the level below the generation controller plus two messages.

The total number of messages system wide equals three times the number of participants and sub-coordinators. The number of messages system wide scales with $O(n)$ with respect to participants.

The time to finish the protocol is one round for sending prepare, one for sending the decision, and one round for propagating votes per level of coordinators, as this is done sequentially. The number of rounds needed to complete the protocol is two plus the number of coordinator levels. The number of rounds to complete the protocol scales with $O(\log n)$, with respect to participants, as the number of levels is a logarithmic function of the number of participants.

The number of messages the top level coordinator needs to handle equals the number of sub-coordinates. Thus, the number of messages the coordinator needs to receive scales with $O(\log n)$, with respect to participants.

The tradeoffs between regular 2PC and hierarchical two-phase commit is that we reduce the number of messages the coordinator needs to handle at the expense of increased time to complete the protocol. The coordinator can still make deterministic decisions regarding committing, but as with Ring-based 2PC, it loses the possibility to commit in case of some failures, as the top coordinator only knows if there are failures or not.
5.1.5 Summary of commit protocols

The table below summarizes the complexity of 2PC, B2PC, Ring-based-, and Hierarchical 2PC. \( n \) is the number of CTS instances, \( v \) is in B2PC, the number of voters and \( c \) the number of sub-coordinators in hierarchical 2PC. If \( x \) participants and sub-coordinators are allowed to speak to a single higher level sub-coordinator, the number of sub-coordinator levels is \( \log_x(n) \), where \( n \) is the number of participants in the protocol. Notice that the number of messages received by the generation controller is in B2PC a function only depending on the number of voters, and not the amount of CTS instances. There is one protocol for each row in the table. The column ”System” shows the number of messages generated by the protocol in the entire system. The column ”GC” shows the number of messages from the generation controllers perspective, and ”Received by GC” is the number of messages that the generation controller needs to receive in one round of the commit protocol. Finally, the last column shows the number of rounds required to complete the protocol.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>System-wide</th>
<th>GC</th>
<th>Received by GC</th>
<th>Time-to-complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>3n</td>
<td>( n + 2 )</td>
<td>( n )</td>
<td>3</td>
</tr>
<tr>
<td>B2PC</td>
<td>2n + v</td>
<td>( 2 + v )</td>
<td>( v )</td>
<td>3</td>
</tr>
<tr>
<td>Ring-based</td>
<td>3n</td>
<td>( n + 2 )</td>
<td>1</td>
<td>( n + 2 )</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>3(n + c)</td>
<td>( 2 + c )</td>
<td>( c )</td>
<td>( 2 + \log_x(n) )</td>
</tr>
</tbody>
</table>

5.2 Sample size in estimation of confidence intervals

B2PC uses the normal distribution to estimate a percentage of committable participants. The normal distribution is only a model, and it has strengths and weaknesses as any model. It is said as a rule of thumb, that it works best when both \( n \cdot p > 5 \) and \( n \cdot (1 - p) > 5 \) [22]. For instance, if the overall percentage of committable participants is 50%, then the sample size should at least be 10. In our case, it is reasonable to assume that the overall percentage of committable participants will in the normal case be closer to 1, but how close is hard to say. We do not have any existing data as to what the normal
failure rate would be in such an installation. If the committable percentage is 99%, there should be a sample size of 500 in order for the model itself to be good. However, the generation controller is not as interested in finding an accurate percentage, as in finding if the percentage satisfies some threshold. Thus, we claim that the number of voters should be selected according to the threshold and not the expected commitability percentage.

If the **threshold** is set at 50 percent, there should be 10 voters according to the rule of thumb, and the model should still be good. For instance, with 10 votes the estimation might say that 80% of the nodes could commit, according to the rule of thumb, this **estimation** did not have a big enough sample size for the normal distribution to be a good model, in other words, the estimation might not be accurate. We could however conclude that our estimation satisfies the threshold as the sample size is large enough to say something about the relationship between the estimated committable percentage and the threshold. Note that the confidence interval would be very large with such a small amount of voters, but we are now only addressing the amount of voters needed for the model itself to be good.

![Graph showing sample size required according to success-rate](image)

**Figure 5.2:** The number of participants needed to make sound estimations, with respect to percentage of committability.

Figure 5.2 shows how the sample size should be selected according to the number of participants that can commit. In order to accommodate the rule of thumb, a number of voters **above** both lines in the graph should be chosen. Each line represents one of the requirements of the rule of thumb. The blue
line represents $n \times (1 - p) > 5$ and the green represents $n \times p > 5$. Note that this number does not say anything about the accuracy of the estimation, but what the sample size should be for the model itself to be good. The number of participants able to commit is not known in advance, but by selecting 500 participants, our model will be sound as long as 1% cannot commit. However, we argue that the number of voters can be selected using the threshold, and not the number of participants able to commit. What the threshold should be depends on our system. If there is a large number of partitions and few replicas, the threshold will typically be high, as the chance of breaking the atomicity property increases. The approach to selecting a threshold is now analyzed.

5.3 Thresholds

The approach to selecting a threshold is not trivial. First and foremost it can be difficult for system designers to select a hard threshold. It is not straightforward to say the e.g. 50% of the system should be up and running in order to commit the transaction. This would depend on the consistency requirements needed by the application and will be a consideration made by the designers. An experiment where different thresholds dependent on the different system constellations, is now presented.

There are no deployed systems within FAST of these sizes, so the experiment will imagine different systems consisting of 10 000 CTS instances. 10 000 instances is chosen in order to show how B2PC behaves in a large scale system. The experiment could just as easily have been done with 100 000 CTS instances, and the calculations would have looked similar. In all experiments, a system consisting of 10 000 CTS instances, all involved in each generation, is assumed. The different experiments assume different system constellations, meaning that the number of partitions and replicas changes for each experiment.

We say that in order to commit a generation, there should not be more than 0.01 percent chance for one or more partitions have more than 50 percent failures. Be ware that this is not the same as saying that 50 percent of the total CTS instances are functional. The threshold requirement is that there
should not be more than 0.01 percent chance for any partition to have more than 50 percent failures.

Remember that the CTS instances are organized in rows and columns. In order to offer the complete index, a query needs to be routed to one member of each column. This member can have any position in the column, meaning that a row can consist of any members from each column. Thus, as long as there are live members in each column — or partition — the system is able to offer a complete search index.

Assuming that failures are distributed independent of partitions, thresholds will be calculated for different system setups, meaning that the number of partitions and CTS instances in each partition changes from system to system. Notice that the fewer CTS instance in each partition, the lower amount of failures the system will tolerate, as the probability for a partition to fail increases. A partition with only 2 CTS instances, will only tolerate one failure without breaking the 50 percent failure requirement, whereas a partition with 5000 CTS instances will tolerate 2500 failed CTS instances before failing.

The reasoning behind wanting to guarantee that partitions have no more than 50 percent failures within a partition is that one would imagine that recovery procedures become increasingly harder the more CTS instances that need to recover a generation per functional CTS instance. This number should be considered an example, and only practical experience with recovery procedures can help setting this limit appropriately in a real system. With probability techniques as B2PC, there will always be a chance for more than 50 percent failures in each partition without the models knowledge, so there is no guarantee for holding this limit, the model only say that in most cases it is.

The approach to calculating the threshold is by doing extensive sampling. For each system constellation, a number of experiments is performed. For each experiment, there is a different amount of failed CTS instances. For each experiment the failed instances are distributed randomly over the partitions in the system. By counting the number of distributions where one or more partitions fail, we get the fraction of distributions satisfying the 50 percent failure limit for any partition. If this fraction is close to 0.01 percent, the total number of failures this system can tolerate while with fair certainty, saying that no partitions have failed, is found. In each experiment, the
amount of failures are distributed randomly 10000 times. If exactly one of the distributions has one or more failed partitions, the partition failure fraction will be 0.01 percent.

randomDist.c is used for the sampling. The code is given in the appendix.

In the first experiment, a system consisting of 2 partitions with 5000 CTS instances each is assumed. This would be an installation with very high query load, but not much content. In the real world, imagine for instance Amazon.com, having a large number of clients, but not extreme amounts of content\(^1\). As a result of the pigeonhole principle [8], such an installation would require at least 2500 failures if there should even be a chance for one of the two partitions to have more than 50 percent failures. Also, if there are more than 5000 failures, one partition must have more than 2500 failures. The fraction of distributions breaking the partition requirement from 2500 to 5100 failures at intervals of 20 failed nodes is calculated. For each amount of failed nodes there are 10000 distributions, and the number distributions failing one or more partitions is counted.

Parameters are:

ctsintstances 10000
partitions 2
failedstart 2500
failedstop 5100
interval 20
trials 10000
partition fail limit 2500

In Figure 5.3, the frequency of partition failure increases exponentially from roughly 4700 failed nodes until 5000 nodes. Above 5000, the partitions fail

\(^1\)We are not saying such an installation would be appropriate for Amazon.com, but only exemplifying the concept of many queries and low amount of content.
Figure 5.3: Partition fail frequency for 2 partitions of 5000 CTS instances each.

every time as expected. The closeup in Figure 5.4 shows that highest number of failed nodes with a partition fail frequency below 0.01 percent is around 4720. Thus, 4720 could be used as a threshold, when estimating the total number of failed CTS instances in this system.

A system with the same number of CTS instances, but divided over 5000 partitions, is now examined. There are 2 CTS instances in each partition. If there are more than 5000 failures, one or more partition will fail automatically. There must at least be 2 failures if there should be a chance for any partition to fail. This kind of system would typically have an enormous amount of content indexed, but not very heavy query traffic. In order to have some kind of fault tolerance, each partition should have a minimum of two CTS instances.

Parameters are:

csstnstances 10000
partitions 5000
failedstart 2
failedstop 5102
Figure 5.4: Close up of partition fail frequency for 2 partitions of 5000 CTS instances each.

interval 20
trials 10000
partition fail limit 1

Figure 5.5: Partition fail frequency for 5000 partitions of 2 CTS instances each.

Figure 5.5 shows that the frequency increases heavily with relatively few failures. It seems that already at 400 instances there is a very low probability
for all partitions to be correct. To get more fine grained data, another experiment with failures ranging from 0 to 400 and with an interval of 1 failure, is conducted.

Parameters are:

ctsintstances 10000
partitions 5000
failedstart 0
failedstop 400
interval 1
trials 10000
partition fail limit 1

Figure 5.6: Partition fail frequency for 5000 partitions of 2 CTS instances each with interval of 1 failure.

Figure 5.6 shows that already at 50 failures, there is more than 10 percent chance for one of the partitions to have failed. It seems that this installation tolerate a very few amount of failures before partitions are in danger of failing completely. Consider the following; What is the probability for two failures to
be in the same partition? The chance for the second failure to be in the same as the first is $1/9999$, because after placing the first failure there are 9999 CTS instances where exactly one is in the same partition as the first failure. Thus, the chance for both failures to be in the same partition is $1/9999$ which is larger than 0.01 percent. Hence, no generations can be committed with B2PC unless the estimation shows that there is not more than one failure. However, estimations will not be this fine-grained as discussed in section 5.4. As a result, a system with 5000 partitions and 2 CTS instances in each partition, cannot use B2PC, unless a bigger uncertainty than 0.01 percent in partition failure probability is allowed.

Finally, a more balanced system is presented, where the 10000 CTS nodes are distributed on 100 partitions with 100 nodes each. In order to have any failed partitions, there needs to be more than 50 failed nodes. If there are more than 5000 failures, one partition must have failed. parameters are:

\begin{verbatim}
ctinstances 10000
partitions 100
failedstart 50
failedstop 5050
interval 20
trials 10000
partition fail limit 50
\end{verbatim}

Figure 5.7 shows exponential increase in partition failure frequency at around 3500 nodes. In the close up, figure 5.8, at 2700 nodes the first occurrences where the partitions fail start to appear. Below 2700 there are no occurrences of failed partitions. Since there was 10000 trials, and no occurrences, the probability of failure of partitions with less than 2700 failures is less than $1/10000$, or 0.01 percent. Thus, the threshold could be set at 2700 failed nodes.

The partition failure frequency graphs increase exponentially, before converting to 1. To explain why the graphs look this way, consider the following.
The different distributions for a certain amount of failures can be put in three different categories. The first category are the distributions not in risk of breaking the partition fail requirement even though one more failure is assumed. The second category are the distributions which did not fail, but would have, if there was one more failure in a specific partition. The last category are the distributions where there are failed partitions.

As the number of failures increases, the different categories will include different fractions of the total distributions. With few failures, most distributions will be in the first category of distributions. There will be a lot of distributions which would not fail even though one more failure is assumed. Few distributions will be in the second category, in risk of failing if another failure is assumed. As a result, there will be only a small increase in the partition fail frequency when one more failure is assumed.

As more failures are assumed, more and more distributions will move from the first category into the second. In this mid-range of the graph there will be a lot of distributions which "almost" failed, and will fail if another failure is assumed. Thus, there will be a large increase in the partition fail frequency as we assume more failures. Eventually, most distributions will have been moved in to the last category. There will be fewer and fewer distributions in risk of breaking the partition failure requirement as another failure is assumed. Thus, difference in distributions in the third category for each
CHAPTER 5. ANALYSIS

Figure 5.8: Close up of fail frequency for 100 partitions of 100 CTS instances each.

5.4 Accuracy of the estimation technique

When calculating a lower boundary of the confidence intervals, we say that with some degree of confidence the true number of committable search nodes lies above the boundary. This is only an estimate, and the true number will in some cases be below the estimated number. In the case where the estimated lower number of committable nodes is too low, the effect will be that we might abort generations that should have been committed. In the case where the estimated number of committable nodes is too high, the result could be that a generation is committed when it should have been aborted. If a 99.95% confidence interval is used, in average, the estimated interval is wrong once every two thousand generation. For this to be critical, this estimation must occur at a time when the generation cannot be committed without breaking the atomicity property, and the estimation must also be off to the extent where the threshold is satisfied, meaning that the generation controller actually commits the generation. These two necessities decreases the chance that this actually occurs, i.e. atomicity will in the long run not be broken for every two thousand generation.
5.4. ACCURACY OF THE ESTIMATION TECHNIQUE

Also, as the lower boundary of the confidence interval will always suggest that there is a possibility for some failed CTS instances, the protocol will never work if the threshold is above the maximum lower boundary possible to estimate.

To illustrate an example, the generation controller asks for 101 votes from the CTS instances, and all are in favor of committing the generation. 101 voters gives us 100 degrees of freedom and the Student’s t-table state that the z-value is 3.390. The estimated percentage of committable nodes is denoted \( \hat{p} \). The lower boundary of the confidence interval is calculated by subtracting the calculated interval from \( \hat{p} \).

\[
\hat{p} = \frac{101}{101} = 1 \quad (5.2)
\]

\[
\hat{p} - 3.390 \times \sqrt{\frac{101/101 \times (1 - 101/101)}{101}} = \hat{p} - 0 = 1 \quad (5.3)
\]

According to the model the confidence interval will only include 100% committability when there are only positive votes. This is naturally not correct. Only 101 CTS instances were asked for their opinion, and there is a chance that others may not be able to commit. In these cases, one negative vote will always be assumed. With one negative vote the following confidence interval is calculated.

\[
\hat{p} = \frac{100}{101} \approx 0.990 \quad (5.4)
\]

\[
\hat{p} - 3.390 \times \sqrt{\frac{100/101 \times (1 - 100/101)}{101}} \approx 0.990 - 0.003 = 0.987 \quad (5.5)
\]

With 101 voters, the highest estimated percentage of committable nodes possible is 99.0 percent and the maximum lower boundary of the confidence interval is 98.7 percent. With 10000 CTS instances, this implies that the generation controller can never say that more than 9870 CTS instances are
correct. If the threshold is above 98.7%, the generation controller gets a problem. If the threshold is above 9870 committable nodes, the generation controller will never commit any generations. In the example with 10000 CTS instances and 5000 partitions, with 130 failed nodes — the lowest number of failures possible to estimate with 101 voters — there is more than 78 percent chance for one of our partitions to have failed, so the threshold in this system would be much higher than what the generation controller is able to estimate.

Let's increase the number of voters to 500 and see if a different picture emerges. One failure is still assumed. Thus, the highest lower boundary possible to estimate of the confidence interval is:

\[
\hat{p} = \frac{499}{500} = 0.998
\]

Continuing:

\[
\hat{p} - 3.371 \times \sqrt{\frac{499/500 \times (1 - 499/500)}{500}} \approx 0.998 - 0.0003 \approx 0.9977
\]

Notice that the z-value (3.371) has changed as a result of having more degrees of freedom.

The lower boundary of the confidence interval is 0.9977. Thus, with 500 voters the generation controller can at best estimate 9977 committable CTS instances. Even with 23 failed nodes there is more than a 4 percent chance for one of our partitions to have failed in the example with 5000 partitions.

This shows some of the limitations of B2PC. The estimations will not be fine-grained enough to handle all system installations without increasing the number of voters substantially.

The old FAST web search engine "AllTheWeb.com" consisted of 750 search nodes in 250 partitions. It would be interesting to see if B2PC would have been useful in this installation. As FAST have experienced that network becomes an issue when around 200 nodes communicate with a single other node, the generation controller requests 200 votes for each generation. With 200 votes, the maximum percentage of committable nodes B2PC is able to estimate is:
\[ \hat{p} = \frac{199}{200} = 0.995 \quad (5.8) \]

\[ \hat{p} - 3.373 \times \sqrt{\frac{199/200 \times (1 - 199/200)}{200}} \approx 0.995 - 0.0012 \approx 0.9938 \quad (5.9) \]

99.38 percent of 750 is 745.35. Thus, it is not possible to estimate that more than 745 nodes are able to commit.

If the system tolerates two failures in each partition, thus keeping the atom-icity property, and finding it acceptable that there is in worst case a single functioning node in a partition, the partition fail frequency for a system with 250 partitions and 3 replicas are given below: Parameters are:

- ctsintstances 750
- partitions 250
- failedstart 0
- failedstop 500
- interval 1
- trials 10000
- partition fail limit 2

Figure 5.9 shows that the probability for partitions to fail increases with relatively few failures. The least amount of failures possible to estimate is 5. In the table below, an extract of the data set is shown. The estimated probability of a partition to fail with 5 failures is 0.01 percent. This is however an estimation, and if we look at the probability for partitions to fail with 4 failures, it is estimated to 0.02 percent. If this level of uncertainty is acceptable, B2PC could be used in installations that looks like AllTheWeb did. It would however not take to many failures before the uncertainty of committing generations becomes significant.
Figure 5.9: Partition fail frequency for AllTheWeb example between 0 and 50 failed nodes.

<table>
<thead>
<tr>
<th>Failures</th>
<th>Partition Fail Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000000</td>
</tr>
<tr>
<td>2</td>
<td>0.000000</td>
</tr>
<tr>
<td>3</td>
<td>0.000000</td>
</tr>
<tr>
<td>4</td>
<td>0.000200</td>
</tr>
<tr>
<td>5</td>
<td>0.000100</td>
</tr>
<tr>
<td>6</td>
<td>0.000100</td>
</tr>
<tr>
<td>7</td>
<td>0.000100</td>
</tr>
<tr>
<td>8</td>
<td>0.000500</td>
</tr>
<tr>
<td>9</td>
<td>0.000200</td>
</tr>
<tr>
<td>10</td>
<td>0.000400</td>
</tr>
</tbody>
</table>

**Node failure characteristics**  When a node fails, it will probably stay failed for some time. As a result, if there is a large fraction of failed nodes in the system as the generation controller tries to create a new generation, it will probably be a large fraction of failed nodes when it tries to create the next generation. This can be used to make it even less likely that generations that should have been aborted are committed. It seems logical that the number of live nodes at one point in time is dependent of the number of live nodes just before. Figure 5.10 shows an example of how the total number of nodes could evolve in a real system. If at time 20, the generation con-
controller suddenly makes an estimation suggesting that the system is healthy and generations should be committed, there is reason to suspect that this estimation is wrong, since the generation controller most likely has seen a series of low estimations. In these cases, the generation controller may run a more expensive protocol to gain more knowledge regarding the state of the system. It could for instance do a new round of voting with a new set of voters, and make a new estimation. If this estimation also shows that the system is healthy, the generation controller can have more confidence in that the system is actually healthy again. Using the history of estimations the generation controller has seen, will make it even less likely that generations which should have been aborted are committed.

![Example of live nodes over time](image)

Figure 5.10: A thought example of how the number of live nodes evolve over time.

5.5 Performance of Generation Coordination

In this section the different approaches to distributed generation coordination are analyzed. In section 4, token-based generation coordination was proposed as a possible approach to distributing the generation controller. An analysis of its operation is now given, together with an analysis of master-based generation coordination. In addition the approach with multiple generation concepts is shortly addressed.
5.5.1 Token-based

With token-based generation a round of token passing is introduced between each generation. The batch identifiers received from the CDS are distributed amongst the generation controller instances, so the total capacity of amount of content waiting for indexing at the generation controllers scale linearly with the number of generation controller instances. Each generations will include more content, as the generation controller instances are allowed to receive more content batch identifiers in between each generation. Each generation controller instance, can not initiate generations at the rate as the generation controller in the original design. The generation controllers will compensate for this by including more content per generation. Thus, the rate of new generations in the system will be nearly the same as with the original design, but with token-based generation, each generation will include more content, and thus increase the indexing capacity. The generation controllers can also prepare themselves for creating a new generation while the token is at another instance, so this would increase parallelism in some sense.

With respect to fault tolerance, the generation holding the token would still be a single point of failure. It would have the same probability to fail as with the original single node solution. In terms of fail-over, the recovery procedures initiated when the token is missing is as complex as with the original design. In both cases, the generation controller needs to recover state from the CTS instances, in order to recover half-completed generations. The difference in terms of recovery is that with the token-based approach, the CDS only needs to resubmit batch identifiers from the failed instance. With the original single node solution, all content batch identifiers would have to be resubmitted.

In addition, if the node responsible for ring management fails, a round of master election would have to be completed before an eventual token is initiated.

5.5.2 Master-based

In master based generation coordination, the generation coordination is done by a single master. This master needs to receive all information regarding
new content batches. New generation controllers can be added in order to handle a larger number of CDS instances, but these instances will at some point inform the generation controller of the content batches to include in the next generation. Thus, the bottleneck will still be at the centralized master.

As for fault tolerance, a master-based approach would still require a single node to run, and the fail over procedures would be as complex as with token-based coordination. In addition, if the master fails, a round of master election would have to be run between the remaining generation controller instances.

### 5.5.3 Multiple generation concepts

If the approach with multiple generation concepts initiated in parallel at each generation controller instance could be used, the indexing capacity would increase linearly with the number of generation controller instances. The tradeoff would be that queries become more complex as they need to be annotated with generation identifiers from all generation concepts. In addition, the search index structure would be more complex, as it needs to handle a index which doesn’t grow linearly as with a single generation concept.

The fault tolerance would be much better, as a failed generation controller instance does not halt indexing, as with the original design. However, if batch identifiers from a failed generation controller instance are resubmitted to a different generation controller instance, they could be re-indexed with a different generation concept. This might lead to repeating documents in the search results. In addition, since two different generation concepts potentially can contain two different versions of a document and both be searchable in the same query, there are issues with the user seeing a logical evolution of the index. The approach with multiple generation concepts is not investigated any further, but if the problems with running multiple generation concepts in parallel is solvable, it would give much better scalability in terms of fault tolerance and indexing capacity.
5.6 Summary

In this chapter the properties of the proposed commit protocols where analyzed with respect to number of messages generated system-wide, from the generation controllers perspective, the number of messages the generation controller needs to receive and the time to complete the protocols. The properties of Token-based and Master-based generation coordination where also analyzed with respect to fault tolerance, fail over, indexing capacity and indexing latency.

In addition, threshold estimation was exemplified and the limitations of B2PC with respect to estimation granularity was shown.

In general, B2PC will work for any systems bigger than 200 nodes, where the number of CTS instances in each partition is of such an size that estimations will be able to distinguish a possible failed generation from a safe one. In practice, this means that the amount of votes the generation controller is able to handle per generation is big enough to make estimations of such granularity that it can make estimations which lies above the threshold. There are to many uncertain criterias for us to be conclusive regarding exactly what system constellations this is.

For one, the criteria for deciding that a partition failed is not a given. In the examples above a partition was considered failed if more than 50 percent of the CTS instances in it had failed. 50 percent was only meant as an example, and in a real domain one would have to consider recovery procedures, query load and required fault tolerance in order to set this limit properly. We also said that there should be less than 0.01 percent chance for any partition to fail for us to risk committing the generation. This percentage should also be regarded as an example, and changing it will have an effect on the thresholds.

The number of CTS instances in total does really affect the suitability of B2PC. It scales with $O(1)$ with respect to increase in CTS instances. What does matter is the relationship between partitions and replicas, the desired fault tolerance and the acceptable uncertainty in committing generations.

As a result of these uncertain parameters it is impossible to be conclusive regarding the system constellations B2PC is appropriate for. The technique used to estimate thresholds is however generic, in the sense that future system
designers can use their desired parameters and get sensible results. If the estimation provides enough granularity to separate a failed partition from a functional, then B2PC can be used as advertised in this thesis.
Chapter 6

Discussion

In the previous chapters some approaches to distributing the generation controller and solve the scalability issues with regular 2PC were designed and analyzed. The approaches have strengths and weaknesses and it is not claimed that they are ideal for all scenarios. For instance, the system architecture in terms of number of partitions and number of replicas in each partition plays a central role in the effectiveness of B2PC. This chapter aims to describe the efficiency of our solution in the specific FAST domain. The chapter also points at considerations system designers should make in order to make systems suitable for B2PC.

6.1 Commit protocols

In the cases where 2PC becomes too expensive in terms of vote-processing at the generation controller, B2PC offers a cheaper way to handle the votes at the expense of not being able to guarantee that the generation is actually committable.

As FAST has experienced that more than 200 nodes talking to a single node becomes troublesome, a maximum of 200 voters could be selected when B2PC is used. According to the rule of thumb, with 200 voters, the largest estimation possible while keeping a good model is
\[ p = \frac{5}{200} = 0.975 \] (6.1)

From Section 6, the maximum lower boundary of a 99.95 percent confidence interval possible to estimate with 200 voters was found to be 99.38 percent. Since 99.38 percent is larger than 97.5 percent, the normal distribution model is still a good enough model for all thresholds possible for the generation controller to make estimations about with 200 voters. Why is this important? This shows that for all installations with a threshold below 99.38 percent, the generation controller does not need to receive more than 200 votes.

In the cases where there is a low amount of replicas per partition, the certainty of the B2PC estimation might become too coarse grained — in other words, if the threshold is larger than 99.38 percent. In such systems, there will typically be a very low amount of failures needed before partitions fail entirely and new generations should be aborted.

As shown with the AllTheWeb example, B2PC might be a feasible choice for installations with relatively few replicas in each partition. System designers who wish to use B2PC, should consider the amount of replicas in each partition needed for B2PC to be useful.

In the case where there are very few partitions and a lot of replicas in each partition, a very high number of failures may be tolerated before starting to abort generations. In these cases, it is probably more likely that generations will be aborted as a result network partitioning, meaning communication problems between the generation controller and a large fraction of the CTS instances. In practice, this event will probably be very rare, and one could question the need to ensure atomicity for each generation.

In stead, the system could chose to disregard ensuring the atomicity property and run a totally optimistic protocol, where generations are just committed on the fly using one-phase commit. In addition, a monitoring system could check if the index is consistent periodically, and not for each generation. In a system with a large number of replicas per partition the fault tolerance will be so good, that the gain in ensuring atomicity is perhaps too small to bother.

The cases where B2PC provides useful is in the cases where it takes a considerable amount of failures to break the atomicity property, but not so many
that it is completely unrealistic that they will occur at the same time. It is
difficult to give examples of what constellations this will be appropriate for as
long as there are no real data from a data center showing how often and for
how long nodes fail.

B2PC is not directly more efficient in terms of indexing latency. It is still
a blocking protocol and uses three rounds to complete. It is a matter of
consideration if it is worth to spend an additional two rounds to ensure
atomicity for each generation, but if so, B2PC is a good approach where in a
large-scale environment, the generation controller can keep fairly tight control
over the state of the CTS instances while not generating huge amounts of
control traffic.

Both Ring-based 2PC and Hierarchical 2PC reduces the number of messages
the generation controller needs to receive by trading off the time to com-
plete the protocol. In effect, the two protocols trades off indexing latency
in exchange for scalability. Trading off indexing latency is not desired, as it
should be considered a key property in the system. The key purpose of the
updateable index is to ensure fast indexing, so indexing latency should be
kept at an absolute minimum.

In our opinion, it is better to trade off the possibility to deterministically
decide if a generation should be committed or not. Using B2PC ensures
atomicity with a very high probability, and in a real system there will always
be a chance for the system to fail in some way anyway. In real systems
we are dealing with probabilities in general. There will always be a certain
probability for the system itself to fail. If we introduce a protocol which
in itself has a certain probability to fail, but decreases the chance for the
system itself to fail, this is a good thing. For instance, if regular 2PC creates
so much traffic for the generation controller that it fails to accomplish its
tasks, 2PC which in itself is a secure protocol, will make the system as a
whole less secure. Introducing a protocol that has some small probability to
fail in itself may actually increase the chance for the system to be functional
as a whole. Running a totally secure protocol with respect to commitability
as 2PC, only makes sense if the surrounding environment stays secure. In
our case, 2PC may break the surrounding environment as a result of being a
bottleneck, thus decreasing the safety of committing generations as opposed
to increasing it.
With sub-second indexing, a large scale system will be dependent on reducing the number of messages the generation controller needs to receive per generation. B2PC accomplishes this. With 200 voters, the accuracy of the estimation is still fine-grained enough for use in a large set of system constellations. System designers should consider using more replicas for each partition, not only in order to provide higher query capacity and fault tolerance, but also because it allows the system itself to run cheaper synchronization protocols. It seems logical that if the availability of a system is better, there is less need for guaranteeing availability.

6.2 Token-based generation coordination

Using Token-based generation coordination allows scaling the amount of batch identifiers the CDS offers the generation controller for inclusion. In the cases where the CDS sends more batch identifiers than the generation controller is able to handle per time unit, the token-based approach might be useful. It does however solve the scalability by increasing the indexing latency, as the generation controller instances need to wait for the token before initiation new generations. Indexing latency will typically be one of the key properties of system, and should not be traded off, so the solution is not optimal in that respect.

However, it seems that the use of a single generation concept requires the use of a single node making decisions. Thus, if a single generation concept is used, there is no apparent way to do limitless scaling of indexing load without effecting indexing latency. This comes from the fact that a single node may only receive a certain amount of content per time unit. Also, there is no obvious way to increase fault tolerance, as there will in some way be a single "master" making the decisions. In the master-based approach, there is a single master, and in the token-based approach the generation controller instance holding the token will be the single point of failure. For this reason, if the master or the token holder fails, incomplete generations must be recovered and a new master elected or a new token initiated.
6.3 Summary

Strengths and weaknesses with B2PC and Token-based generation coordination has been argued. B2PC will under some circumstances offer a reasonably cheap and very scalable way to control that the atomicity property is not broken when indexing new content. It is not able to guarantee that atomicity is broken, but it is very likely to make the right decisions. 2PC is a completely safe protocol, in the sense that the generation controller knows about the state of all participants and can therefore make the correct decision in all cases.

Master-based generation coordination was dismissed as it does not offer our system anything with respect to scalability, fault tolerance or indexing latency. Token-based generation coordination was proposed as a better solution, though it is neither optimal. It trades off indexing latency for indexing capacity.
Chapter 7

Conclusion

7.1 Conclusion

In this paper FAST’s search platform with respect to indexing and querying was described. The generation controller ensures that content is made available atomically and consistently. It was argued that it will become a bottleneck as the system scales as a result of using 2PC. Also, as a result of being a single node solution, it will have to receive batch identifiers from all CDS instances. This will be a scaling restriction as well.

An alternative to 2PC, referred to as B2PC was designed to decrease the number of messages the generation controller needs to receive in the voting phase of 2PC. It scales with O(1) as it only receives votes from a set of the total amount of CTS instances, and this set does not need to grow as the number of CTS instance does. In addition we discussed Ring-based 2PC and Hierarchical 2PC, as possible alternatives.

Distributing the generation controller was also the goal of this thesis, in order to increase fault tolerance and scalability with respect to the indexing load. A master-based approach and a token-based approach was discussed. In addition, an idea of multiple generation concepts was presented.

Distributing the generation controller for fault tolerance and indexing capacity was found hard. We claim that this is a result of having a single concept of
generations. A single generation concept will require some sort of centralized approach to including content in new generations. Token-based generation coordination trades off indexing latency for scaling of content load. It is not the task of our work to conclude whether this approach should be chosen when the time comes to build such a large scale installation. We do however claim that the scalability problems of content load is a direct result of having a single generation concept, and that future designers should consider the effects of this concept. The single generation concept solves the problems of the updateable index with respect to having a logically evolving search index, but it is a single synchronization point, which it seems will have scalability issues in any case.

Binomial Two-Phase Commit was presented as a way of ensuring atomicity in a large-scale environment. We have argued that it is superior to regular 2PC, with respect to number of messages the generation controller needs to handle, and also because it allows a number of failures while it with a large degree of certainty ensures atomicity of new generations. It is our opinion that it should be preferred to regular 2PC in systems at least larger than 200 CTS instances and with enough CTS instances in each partition for the estimations to become fine-grained enough to distinguish a failed partition from a functional one. When system architects design large-scale solutions, they should not only consider the content load requirements, availability and query capacity of the system, but also the effects the system constellation has on possibilities to run efficient protocols. If a system fulfilling the customer requirements to content load and query capacity, is unable to use B2PC as a result of having too few replicas in each partition, but too large to effectively use regular 2PC, designers could consider increasing the number of replicas making the system more eligible for use with B2PC.

Our problem statement was defined as follows:

*How can a distributed generation controller, handling failures gracefully, achieving index consistency effectively without compromising search performance nor indexing latency and scales with respect to content load, number of nodes and query load be created?*

In terms of handling failures gracefully, token-based generation coordination still needs to recover state from all CTS instances if the token holding generation controller fails. In addition, the token has to be recovered, so the
procedure is slightly more complicated than with the old design. Using B2PC
achieves search index consistency very efficiently in the normal case, but may
break consistency occasionally. In practice, we claim that this will be very
rare. As for indexing latency, B2PC does not introduce more latency than
the current 2PC design. Token-based generation coordination introduces in-
dexing latency by passing the token, and the solution scales up indexing
capacity by trading off indexing latency. Search performance is not affected
by our solution. The solution scales much better with respect to number
of CTS instances than the old design, B2PC scales with $O(1)$ with respect
to CTS instances, 2PC scales with $O(n)$. Content load scales by increasing
indexing latency, but a single node does no longer need to receive all content
batch identifiers. The solution will eventually, as the number of generation
controller instances increas, have problems guaranteeing sub-second indexing
as generation controller instances do not know how long they will have to
wait for the token to return.

7.2 Future work

For future work, we propose considering the effects of restricting index to
a single generation concept. As argued, multiple generation concepts would
allow indexing of content in parallel, possibly offering considerable gain in the
amount of content the system can index per time unit. However, there are
unsolved issues with such an approach, not relevant with a single generation
concept. For one, how one would restrict different versions of the same
document being made visible through different generation concepts is one.
How to make sure content is only indexed once in case of failures is another.
Solving this at CTS level might to be an approach, but it seems hard to do
this with today’s search index structure.

B2PC is an example of how it is possible in a large-scale environment, to
utilize probabilities within a large set of homogenically functioning nodes.
FAST researchers should consider if there are more areas within the FAST
architecture where such approaches might be useful.

The normal distribution model is not optimal for rare events. If the amount
of concurrent failures in a large-scale installation is relatively small, there are
perhaps better estimation techniques which could be used. For instance, the Poisson distribution could be considered as a replacement possibly offering more accurate estimations.

Also, we encourage FAST to when possible, gain experience with our approach in a large scale data center environment. It is hard to make conclusive arguments when there is a large number of uncertain parameters. As a result, even simulation of these kind of systems becomes hard, because one have to make a lot of assumptions of the environment which are not funded on real data. The failure rates in a real environment is one of the properties having a large impact in the effectiveness of B2PC.

7.3 Acknowledgements

A big thanks should be directed to Torgeir Hovden and Rolf Michelsen for their contribution to this thesis. They welcomed me to the FAST environment and have been a source of knowledge and inspiration throughout the process. Without their focus, knowledge and practical experience this thesis would not have been. The discussion we have had during the last one and a half years have been of great importance for the shaping of this thesis, but have also thought me valuable lessons regarding the pragmatic nature of professional software development, knowledge hardly acquired in the academic university setting.

I would also like to direct a big thanks to Pål Halvorsen at Simula for his contributions to this thesis. The supervision from Pål Halvorsen has been of great importance with respect to the academic aspects of the thesis. His contributions have been crucial, especially in terms of shaping the work into a master thesis.

I would also like to thank Øystein Torbjørnsen at FAST for presenting the problem domain. The assistance of Torbjørn Helvik with respect to the statistical aspects of the thesis has been of great help, and I would like to thank him for the contribution.

I would like to thank fellow students Egil Aspevik Martinsen, Eivind Berg Johannessen and Øystein Gundersen for excellent technical and social input.
On a personal basis, I would like to thank all the fantastic employees at FAST for all support and for welcoming me to their work environment. The chess-gang at 4th were especially appreciated.
Bibliography


## Appendix

### Student’s t-table [21] [20]

<table>
<thead>
<tr>
<th>V</th>
<th>$t_{000}$</th>
<th>$t_{050}$</th>
<th>$t_{025}$</th>
<th>$t_{010}$</th>
<th>$t_{005}$</th>
<th>$t_{001}$</th>
<th>$t_{005}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.078</td>
<td>6.314</td>
<td>12.706</td>
<td>31.821</td>
<td>63.656</td>
<td>318.289</td>
<td>636.613</td>
</tr>
<tr>
<td>3</td>
<td>1.638</td>
<td>2.353</td>
<td>3.182</td>
<td>4.541</td>
<td>5.841</td>
<td>10.214</td>
<td>12.924</td>
</tr>
<tr>
<td>4</td>
<td>1.533</td>
<td>2.132</td>
<td>2.776</td>
<td>3.747</td>
<td>4.604</td>
<td>7.173</td>
<td>8.610</td>
</tr>
<tr>
<td>5</td>
<td>1.476</td>
<td>2.015</td>
<td>2.571</td>
<td>3.365</td>
<td>4.032</td>
<td>5.594</td>
<td>6.869</td>
</tr>
<tr>
<td>6</td>
<td>1.440</td>
<td>1.943</td>
<td>2.447</td>
<td>3.143</td>
<td>3.707</td>
<td>5.208</td>
<td>5.959</td>
</tr>
<tr>
<td>7</td>
<td>1.415</td>
<td>1.895</td>
<td>2.365</td>
<td>2.998</td>
<td>3.499</td>
<td>4.785</td>
<td>5.408</td>
</tr>
<tr>
<td>8</td>
<td>1.397</td>
<td>1.860</td>
<td>2.306</td>
<td>2.896</td>
<td>3.355</td>
<td>4.501</td>
<td>5.041</td>
</tr>
<tr>
<td>9</td>
<td>1.383</td>
<td>1.833</td>
<td>2.262</td>
<td>2.821</td>
<td>3.250</td>
<td>4.297</td>
<td>4.781</td>
</tr>
<tr>
<td>10</td>
<td>1.372</td>
<td>1.812</td>
<td>2.228</td>
<td>2.764</td>
<td>3.169</td>
<td>4.144</td>
<td>4.587</td>
</tr>
<tr>
<td>11</td>
<td>1.363</td>
<td>1.796</td>
<td>2.201</td>
<td>2.718</td>
<td>3.106</td>
<td>4.025</td>
<td>4.437</td>
</tr>
<tr>
<td>12</td>
<td>1.356</td>
<td>1.782</td>
<td>2.179</td>
<td>2.681</td>
<td>3.055</td>
<td>3.930</td>
<td>4.318</td>
</tr>
<tr>
<td>13</td>
<td>1.350</td>
<td>1.771</td>
<td>2.160</td>
<td>2.650</td>
<td>3.012</td>
<td>3.852</td>
<td>4.221</td>
</tr>
<tr>
<td>14</td>
<td>1.345</td>
<td>1.761</td>
<td>2.145</td>
<td>2.624</td>
<td>2.977</td>
<td>3.787</td>
<td>4.140</td>
</tr>
<tr>
<td>15</td>
<td>1.341</td>
<td>1.753</td>
<td>2.131</td>
<td>2.602</td>
<td>2.947</td>
<td>3.733</td>
<td>4.073</td>
</tr>
<tr>
<td>16</td>
<td>1.337</td>
<td>1.746</td>
<td>2.120</td>
<td>2.583</td>
<td>2.921</td>
<td>3.686</td>
<td>4.015</td>
</tr>
<tr>
<td>17</td>
<td>1.333</td>
<td>1.740</td>
<td>2.110</td>
<td>2.567</td>
<td>2.898</td>
<td>3.646</td>
<td>3.965</td>
</tr>
<tr>
<td>18</td>
<td>1.330</td>
<td>1.734</td>
<td>2.101</td>
<td>2.552</td>
<td>2.878</td>
<td>3.610</td>
<td>3.922</td>
</tr>
<tr>
<td>19</td>
<td>1.328</td>
<td>1.729</td>
<td>2.093</td>
<td>2.539</td>
<td>2.861</td>
<td>3.579</td>
<td>3.883</td>
</tr>
<tr>
<td>20</td>
<td>1.325</td>
<td>1.725</td>
<td>2.086</td>
<td>2.528</td>
<td>2.845</td>
<td>3.552</td>
<td>3.850</td>
</tr>
<tr>
<td>21</td>
<td>1.323</td>
<td>1.721</td>
<td>2.080</td>
<td>2.518</td>
<td>2.831</td>
<td>3.527</td>
<td>3.819</td>
</tr>
<tr>
<td>22</td>
<td>1.321</td>
<td>1.717</td>
<td>2.074</td>
<td>2.508</td>
<td>2.819</td>
<td>3.505</td>
<td>3.792</td>
</tr>
<tr>
<td>23</td>
<td>1.319</td>
<td>1.714</td>
<td>2.069</td>
<td>2.500</td>
<td>2.807</td>
<td>3.485</td>
<td>3.768</td>
</tr>
<tr>
<td>24</td>
<td>1.318</td>
<td>1.711</td>
<td>2.064</td>
<td>2.492</td>
<td>2.797</td>
<td>3.467</td>
<td>3.745</td>
</tr>
<tr>
<td>25</td>
<td>1.316</td>
<td>1.708</td>
<td>2.060</td>
<td>2.485</td>
<td>2.787</td>
<td>3.450</td>
<td>3.725</td>
</tr>
<tr>
<td>26</td>
<td>1.315</td>
<td>1.706</td>
<td>2.056</td>
<td>2.479</td>
<td>2.779</td>
<td>3.435</td>
<td>3.707</td>
</tr>
<tr>
<td>27</td>
<td>1.314</td>
<td>1.703</td>
<td>2.052</td>
<td>2.473</td>
<td>2.771</td>
<td>3.421</td>
<td>3.689</td>
</tr>
<tr>
<td>28</td>
<td>1.313</td>
<td>1.701</td>
<td>2.048</td>
<td>2.467</td>
<td>2.763</td>
<td>3.408</td>
<td>3.674</td>
</tr>
<tr>
<td>29</td>
<td>1.311</td>
<td>1.699</td>
<td>2.045</td>
<td>2.462</td>
<td>2.756</td>
<td>3.396</td>
<td>3.660</td>
</tr>
<tr>
<td>30</td>
<td>1.310</td>
<td>1.697</td>
<td>2.042</td>
<td>2.457</td>
<td>2.750</td>
<td>3.385</td>
<td>3.646</td>
</tr>
<tr>
<td>40</td>
<td>1.303</td>
<td>1.684</td>
<td>2.021</td>
<td>2.423</td>
<td>2.704</td>
<td>3.307</td>
<td>3.551</td>
</tr>
<tr>
<td>60</td>
<td>1.296</td>
<td>1.671</td>
<td>2.000</td>
<td>2.390</td>
<td>2.660</td>
<td>3.232</td>
<td>3.460</td>
</tr>
<tr>
<td>120</td>
<td>1.289</td>
<td>1.658</td>
<td>1.980</td>
<td>2.358</td>
<td>2.617</td>
<td>3.160</td>
<td>3.373</td>
</tr>
</tbody>
</table>

*Table 3: Critical values for Student's $t$. 

Note: $t$-values are approximations for large values of $V$. 

For $V = \infty$, the $t$-values approach the standard normal distribution.*
<table>
<thead>
<tr>
<th>t Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cum. prob</strong></td>
</tr>
<tr>
<td><strong>one-tail</strong></td>
</tr>
<tr>
<td>1.00</td>
</tr>
<tr>
<td>0.50</td>
</tr>
<tr>
<td>0.25</td>
</tr>
<tr>
<td>0.20</td>
</tr>
<tr>
<td>0.10</td>
</tr>
<tr>
<td>0.05</td>
</tr>
<tr>
<td>0.02</td>
</tr>
<tr>
<td>0.01</td>
</tr>
<tr>
<td><strong>two-tails</strong></td>
</tr>
<tr>
<td>1.00</td>
</tr>
<tr>
<td>0.50</td>
</tr>
<tr>
<td>0.25</td>
</tr>
<tr>
<td>0.20</td>
</tr>
<tr>
<td>0.10</td>
</tr>
<tr>
<td>0.05</td>
</tr>
<tr>
<td>0.02</td>
</tr>
<tr>
<td>0.01</td>
</tr>
</tbody>
</table>

**Confidence Level**

<table>
<thead>
<tr>
<th>%</th>
<th>0%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>95%</th>
<th>99%</th>
<th>99.8%</th>
<th>99.9%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0.000</td>
<td>0.674</td>
<td>0.842</td>
<td>1.036</td>
<td>1.282</td>
<td>1.645</td>
<td>1.960</td>
<td>2.336</td>
<td>2.576</td>
<td>3.080</td>
</tr>
</tbody>
</table>
#include <time.h>
#include <stdio.h>

// main()
//
// params:
// ctsinstances - the total number of nodes to distribute
// partitions - the number of partitions (or baskets) the nodes are distributed on
// failedstart - the initial number of failed nodes assumed
// failedstop - the last number of failed nodes assumed
// interval - the interval at which failed nodes are increased
// trials - the number of times a specific amount of failures are distributed
// (the higher number the higher accuracy in estimation)
// partition fail limit - the maximum number of failures allowed in a partition before it fails
// outputfile - filename for output (existing files are overwritten)
//
// This program does a number of experiments where there are a total of <tsinstances> nodes to be distributed on
// <partitions> number of partitions (or baskets). For each experiment there is a number of failed nodes amongst
// <tsinstances>. The number of failed nodes start at <failedstart> and is increased by <interval> for each
// experiment, until <failedstop> is reached. For each experiment the nodes are distributed onto the partitions
// <trials> times and the number of distributions where there are one or more partitions with more than <partition fail limit>
// failures are counted. The program returns a list containing pairs on the form:
//
// <failednodes> <failedDistributions / trials>
//
// which is written to <outputfile>.
//
// There is no error checking of any sort, and the implementation is based on 'brute force'.
//
int main(int argc, char **argv)
{
    if (argc != 9)
    {
        printf("randomDist <tsinstances> <partitions> <failedstart> <failedstop> <interval> <trials> <partition fail limit> <outputfile>\n");
        exit(1);
    }
    int nodes = atoi(argv[1]);
    int baskets = atoi(argv[2]);
    int failedNodesStart = atoi(argv[3]);
    int failedNodesStop = atoi(argv[4]);
    int interval = atoi(argv[5]);
    int basketSize = nodes / baskets;
    int n = atoi(argv[6]);
    int basketFailLimit = atoi(argv[7]);
    int basketStruct[baskets];

    FILE *output = fopen(argv[8], "w");

    int h;
    int i;
    int j;
    srand(time(NULL));
    clock_t start = time(NULL);
    double results[(failedNodesStop - failedNodesStart) / interval];
// one experiment for each number of failed nodes.
for (h = 0; h < (failedNodesStop - failedNodesStart) / interval; h++)
{
// for each trial
    float failCounter = 0.0;
    for (i = 0; i < n; i++)
    {
        bzero(basketStruct, baskets*sizeof(int));

        // distribute the failed nodes randomly in each basket
        // if the basket breaks the failure limit, register and abort
        for (j = 0; j < failedNodesStart + (interval * h); j++)
        {
            int id = rand() % (baskets);

            // as a basket gets filled with failures the chance of
            // putting another failure in this basket decreases
            // consider 10000 nodes divided on 5000 baskets with room for two nodes each.
            // if a basket gets the first failure, there are now 9999 possible places to
            // put the next failure and only one of these places are in our basket.
            // thus, we have to take into account the number of failures already in the basket.
            if ((rand() % basketSize) >= basketStruct[id])
            {
                if (++basketStruct[id] > basketFailLimit)
                {
                    failCounter += 1;
                    break;
                }
            }
            else
            {
                j += 1;
            }
        }
    }
    results[h] = failCounter/n;
}
clock_t end = time(NULL);

printf("Time to complete: %d\n", end - start);
for (h = 0; h < (failedNodesStop - failedNodesStart) / interval; h++)
{
    fprintf(output, "%d\n\n", failedNodesStart + interval*h, results[h]);
}
fclose(output);