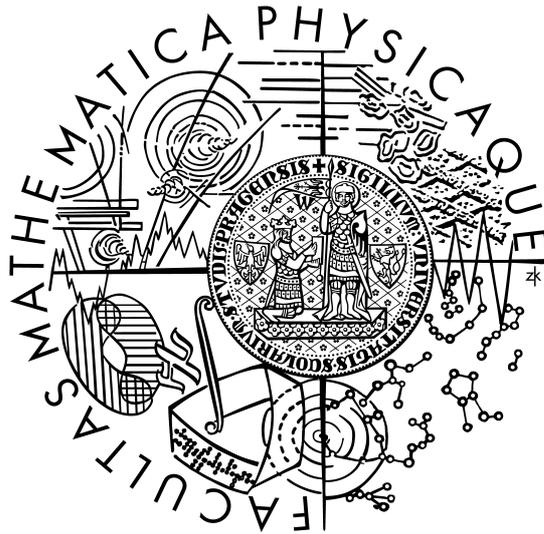


Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jakub Podhorný

Transakce ve fulltextovém vyhledávacím stroji

Katedra softwarového inženýrství

Vedoucí diplomové práce: *RNDr. Leo Galamboš, Ph.D.*

Studijní program: *Informatika, Softwarové systémy*

Poděkování

Děkuji tímto vedoucímu diplomové práce RNDr. Leo Galambošovi, Ph.D. za pozornost, kterou mé práci věnoval, za jeho odborné rady a metodické vedení při zpracování této diplomové práce.

Prohlašuji, že jsem svou diplomovou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne

Jakub Podhorný

Table of Contents

1 Assignment.....	8
1.1 Structure of master thesis.....	8
1.2 Assignment requirements.....	8
1.3 Form of a solution.....	9
2 Documentation of Egothor – index creation part.....	10
2.1 Introduction to creation and usage of index in Egothor.....	10
2.2 Process of adding a document.....	11
2.3 Creating of instance of the Tanker.....	11
2.3.1 Barrels.....	12
2.3.2 Tanker creation.....	13
2.3.3 Distributor – details.....	13
2.3.4 Dynamizer - details.....	13
2.3.5 GlobalPositions – details.....	14
2.3.6 DataRepository - details.....	15
2.3.6.1 DocumentsDB.....	15
2.3.6.2 Berkeley.....	16
2.3.7 Loading previous Tanker state.....	16
2.3.8 Creation of documents.....	16
2.3.9 Adding documents into the Tanker.....	17
2.3.10 The Tanker.append() process.....	18
2.3.11 Removing of a document.....	20
2.3.12 Merge.....	21
2.3.13 Listener.....	22
2.3.14 Query.....	22
2.3.15 Commit.....	23
2.4 Conclusion.....	24
3 Concurrency control.....	25
3.1 General introduction to transaction processing and terminology.....	25
3.1.1 Transaction.....	25
3.1.2 Rollback.....	26
3.1.3 Deadlocks.....	26
3.1.4 Starvation.....	27

3.2 Concurrency in Egothor.....	27
3.2.1 Project terminology.....	28
3.2.2 Model situations and requirements.....	28
3.2.3 Access granularity.....	29
3.2.3.1 Granularity problem.....	30
3.2.4 Deadlocks in Egothor.....	31
3.3 Concurrency control approaches.....	31
3.3.1 Locking.....	32
3.3.1.1 Basic terms.....	32
3.3.1.2 Scheduler.....	32
3.3.1.3 Deadlocks.....	33
3.3.1.4 Granularity.....	33
3.3.1.5 Evaluation.....	34
3.3.2 Timestamp ordering.....	35
3.3.2.1 Basic terms.....	35
3.3.2.2 Basic timestamp ordering scheduler.....	35
3.3.2.3 Implementation issues.....	36
3.3.2.4 Evaluation.....	37
3.3.3 Multiversion.....	38
3.3.3.1 Basic terms.....	38
3.3.3.2 Schedulers.....	38
3.3.3.3 Evaluation.....	40
3.3.4 Certification.....	40
3.3.4.1 Basic terms.....	41
3.3.4.2 Schedulers.....	41
3.3.4.3 Evaluation.....	42
3.4 Summary and sketch of solution.....	42
4 Implementation of the solution.....	44
4.1 Egothor issues.....	44
4.1.1 Index constancy.....	44
4.1.2 Documents' revisions.....	45
4.1.3 Data repository revisions.....	45
4.2 Main design of concurrency control.....	46
4.2.1 Lock server.....	46

4.2.2 Local solution.....	47
4.2.3 Lock granularity.....	48
4.2.4 Lock types.....	48
4.2.5 Lock validity.....	48
4.2.6 Deadlocks.....	49
4.2.7 Starvation.....	49
4.2.8 Local solution functionality issues.....	50
4.2.9 Recovery.....	50
4.2.10 Lock implementation.....	51
4.3 New user interface layer.....	52
4.3.1 Secure Tanker.....	52
4.3.2 Tanker initialization.....	52
4.3.3 Appending new barrels.....	52
4.3.4 Removal of documents.....	53
4.3.5 Tanker commit.....	53
4.3.6 Search queries.....	55
4.3.7 Index cleaning.....	55
4.3.8 Modifiers' active state.....	56
4.3.9 Global index recovery.....	56
4.3.10 Implementation issues and other modifications.....	57
5 Conclusion.....	58
6 Bibliography.....	60
A. Appendix.....	61
A.1 Functional tests.....	61
A.1.1 Lock management test.....	61
A.1.2 Overall test.....	63
A.2 UML diagrams.....	66
A.3 CD-ROM.....	71
A.4 Programmer's Quick Start.....	72

Table of Illustrations

Figure 1: Egothor's index file structure.....	17
Figure 2: Merging.....	21

Figure 3: Lock server method.....	47
Figure 4: Lock files method.....	48
Figure 5: Listeners.....	66
Figure 6: ThickBarrel and ThickBarrelIn.....	66
Figure 7: Slot Items and Barrels.....	67
Figure 8: Document class diagram - detail.....	67
Figure 9: Tanker class diagram - detail.....	68
Figure 10: Tanker class diagram.....	69
Figure 11: TankerImplSecure.....	70

Název práce: *Transakce ve fulltextovém vyhledávacím stroji*

Autor: *Jakub Podhorný*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Leo Galamboš, Ph.D.*

e-mail vedoucího: *Leo.Galambos@mff.cuni.cz*

Abstrakt: *Tato diplomová práce se zabývá implementací vhodného algoritmu transakčního zpracování z prostředí SŘBD do fulltextového vyhledávacího stroje Egothor.*

Práce zahrnuje analýzu funkcionality systému Egothor, procesů při vytváření a spravování indexu a analýzu původního zdrojového kódu vyhledávacího stroje. Následuje popis existujících algoritmů transakčního zpracování a jejich následné ohodnocení a vzájemné srovnání. Ohodnocení popsaných algoritmů je primárně zaměřeno na prostředí webového vyhledávacího systému Egothor, kdy je vybrán ten nejvhodnější k implementaci.

V rámci práce byl vybrán algoritmus plně naimplementován spolu s dalšími kladenými funkčními požadavky. Popis výsledné implementace uzavírá tuto diplomovou práci.

Klíčová slova: *Egothor, Transakce, Fulltextový Vyhledávací Stroj*

Title: *Transactions in a fulltext search engine*

Author: *Jakub Podhorný*

Department: *Department of Software Engineering*

Supervisor: *RNDr. Leo Galamboš, Ph.D.*

Supervisor's e-mail address: *Leo.Galambos@mff.cuni.cz*

Abstract: *This master thesis covers implementation of a suitable algorithm of transaction processing existing in DBMS environment into the fulltext search engine Egothor.*

The thesis consists of analysis of functionality of the Egothor system, of the processes being executed during creation and management of the index and analysis of the original source code of the search engine project. Description of the existing algorithms of transaction processing follows along with their evaluation and comparison. Evaluation of the described algorithms is primarily concentrated on the environment of the web search engine Egothor and the most suitable method is chosen for further implementation.

As a part of the thesis the chosen algorithm has been fully implemented along with some other project functional requirements. Description of the final implementation closes this master thesis.

Keywords: *Egothor, Transaction, Fulltext Search Engine*

1 Assignment

Egothor is a full text search engine written in Java language that stores its index on a hard drive of a computer machine and manages it using merge technique. So far there is no concurrency control mechanism that would allow engine's threads to execute in a safe parallel way. The goal of this work is to compare some suitable concurrency control mechanisms and implement the most suitable one for the Egothor system.

1.1 Structure of master thesis

The basic track of this master thesis will copy the way how it was elaborated. This initial part sums up the whole assignment, introduces to the problem and states all major requirements.

The second part then analyzes Egothor and its inner designs, structures and processes to fully illustrate the initial state of Egothor project and to help with comparison with the final result of the implementation at the end of the work.

The third part then describes some most common transaction processing algorithms that would fulfill project's requirement to implement concurrency control. It mainly focuses on the needs of Egothor full text searching system and compares the different approaches with each other and mainly with the requirements of the project.

Finally the fourth part describes implementation of the chosen and the most suitable approach and illustrates Egothor system design changes that were necessary to successfully complete the task.

1.2 Assignment requirements

The Egothor project has one distinguished feature. The whole index is stored at hard drives and the hard drives are shared by all working threads of the engine. So the whole disk volume is accessible by all threads locally. Egothor machine runs in a computer cluster.

From a project manager point of view there are several major requirements that must be fulfilled to successfully complete the task. These requirements, of course, cover concurrency and efficiency, for Egothor, as full text searching machine, has to execute swiftly with no great overhead caused by implementation of transactions. The following list sums up all set requirements.

1. Several threads executed in a single Java Virtual Machine cannot conflict with each

other with fatal consequences during appending of documents or removing them. Every such an operation must leave the index in a consistent state after commit and visible for everyone.

2. Several threads executed in a single Java Virtual Machine cannot conflict with each other with fatal consequences while one of them is reading the index.
3. Several threads executed in multiple Java Virtual Machines on a single computer machine should work as described for single Java Virtual Machine.
4. Several threads executed in multiple Java Virtual Machines on multiple computer machines should work as described for single Java Virtual Machine.
5. Once a thread opens an index and starts reading its content, the index visible to the thread should not change by any other thread's intervention without the reader noticing it or allowing it. In other words, once a thread opens an index, it must have a certainty, that the index will remain constant during some period of time, so when some read out data has a relationship with other so far unread data, the relationship will not change without the reader noticing it or even allowing it.
6. Avoid inefficient copying of files or unnecessary I/O operations.
7. Try to avoid any global central unit of the application, like a server. If it is necessary to have one, make another implementation, also, that can work without it, even it would worsen some aspects of security or functionality. That way working threads are totally independent and do not rely on any other unit. The user may choose his preferable way to use.
8. Do not inefficiently use disk space, delete unaccessible parts of index as soon as possible.
9. Do keep solution compact and transparent.
10. Implement a new layer of user interface for managing the index using concurrency control.

1.3 Form of a solution

It is clear even now that when the final transaction processing algorithm is chosen, it probably will be necessary to alter it to make it the most effective implementation for Egothor. This excludes right at the beginning available complex transaction processing solutions that would be just imported into Egothor, like CORBA and other ones available, because it is just not the most efficient way with respect to all the requirements.

2 Documentation of Egothor – index creation part

This documentation chapter has a goal to describe some Egothor's parts that are relevant to this part of the project. It will go over inner processes and inner structures of Egothor and explain their functionality and meaning. The whole process of creation of index that starts with instantiation of basic classes and ends with created index that is ready for user search queries will be described. The explanation is kept in a such detail that it helps to understand and picture the problem, but that does not bother with further technical details of the implementation.

In this chapter there has been used Egothor project documentation by RNDr. Leo Galamboš, Ph.D. [1] even, though, it mainly covers different parts of the project. The main written source of information for this chapter was the source code itself.

There are UML diagrams in the Appendix of this thesis. It is recommended to consult the description of the design with the illustrations.

2.1 Introduction to creation and usage of index in Egothor

Egothor offers several user-friendly classes that are supposed to be used for indexation and query processes. The basic class is a `Tanker` which represents a stored index in a specific location on the hard drive. It implements the necessary logic for all the processes of adding documents into index, removing and querying, but it is not very user-friendly and therefore is defined *abstract*. That is the reason why there exists basic user class `TankerImpl` that offers all the functionality user may need for indexing and searching, all done in a user friendly way through simple method calls `append(param)`, `removeDoc(param)`, `commit()`, `query(params)` etc. It is not multi-threaded safe, of course.

The class `TankerImpl` can be wrapped into several others, such as `BufferedTanker`. `BufferedTanker` improves speed of the whole process by introducing cache memories in which the index is gradually created. After a cache is full, it calls methods on the inner `TankerImpl`, which flushes the cache to the hard drive and proceeds along.

`TankerImpl` thus represents the whole index of all used documents as one standalone index. Documents that user wants to store in Egothor are processed at the beginning and inverted lists of their content are created. Inverted list is the basic structure used in Egothor

for storing document's indexed content.

During the process of building of index user's input documents represented by class `Document` are taken as an input and their inverted lists along with other information are stored in the Egothor index. This chapter will not discuss creating of instances of `Document` or their inverted lists in much detail, because that is not essential to my part of this project and it is also described in documentation of Egothor [1].

2.2 Process of adding a document

To add a document there must be a structure that may accept the document first. That, of course, represents a tanker. The whole process of indexation consists of four parts as described in the *Table 1: Document adding process*.

Step 1	Creating of instance of a <code>Tanker</code> – creates index folder or load already existing one, initialize all parameters for merge algorithm and other essential parts of the tanker.
Step 2	Creating of documents as Egothor classes
Step 3	Appending documents into tanker's index through tanker methods
Step 4	Commit of the process – saves all caches into the hard drive.

Table 1: Document adding process

The following chapters will cover the whole process in further detail, concentrating on every part that is important for using index. The chapter *2.3 Creating of instance of the Tanker* describes the whole tanker in detail. It is essential to understand how the tanker works, how the data is stored, how it is read, because implementing concurrency will lead to more or less changes in design of all parts.

2.3 Creating of instance of the Tanker

All the essential settings for the `Tanker` are passed to it as parameters of its constructor. For the basic `TankerImpl` that is

```
public void TankerImpl(String indexDir, DataRepository repo,  
    boolean MTE, int capacity, int mergeFactor).
```

First parameter `indexDir` is the root directory of the Tanker's index – the place where the index is gradually created and where it is finally kept. Second parameter `repo` is of type `DataRepository` which is an interface for classes that implement storing of metadata of all documents. This can be arranged in many ways, in Egothor there are two ways so far, first implementation is `DocumentsDB` class - a class that stores metadata in two files on the hard drive as a simple array of data, and second one is `Berkeley`, implementation that uses external Sleepycat Berkeley DB which is much more complex and it basically creates key-value database on the hard drive (further description in the chapter 2.3.6 `DataRepository` - details). Third `MTE` parameter is a flag that tells if the operations in the Tanker will run in parallel or not. Fourth parameter `capacity` specifies the count of the layers in merge algorithm of indexation and finally `mergeFactor` specifies the maximum number of index parts that have similar length.

In case of using `BufferedTanker`, the constructor looks like

```
public BufferedTanker(int cacheSize, Tanker orig),
```

where `int cacheSize` specifies the maximum number of documents stored in the cache memory before flushing process is called. The second argument `Tanker orig` is an instance of a `Tanker` which will be wrapped into the `BufferedTanker` and is supposed to implement the necessary logic of the processes, for example `TankerImpl` (this buffered wrapper does not implement any other logic except the caching).

2.3.1 Barrels

Index created by Egothor in its inner structure consists of multiple parts called `barrels`. `Barrel` is a stand alone structure of index of one or more documents, which can be opened for reading (via `BarrelReader` class), can manage the removal of documents from itself and can return metadata of inverted lists and documents. The main `tanker` itself is an instance of such a `barrel` also. There are several types of `barrels` depending on the functionality (`ThickBarrel` for barrel stored on the hard disk in one big file, `MemoryBarrel` for cached version of a barrel). In addition there are readers and writers to operate with the barrels (Figure 6: `ThickBarrel` and `ThickBarrelIn`, page 66).

2.3.2 Tanker creation

During the creation process of `TankerImpl` object several crucial objects are created that compose the final user friendly tanker object. These crucial object are described in the table *Table 2: Tanker's major inner structure.*

Inner object	Usage
<code>Distributor</code>	used for distributing tasks to the index objects (to the barrels)
<code>Dynamizer</code>	used for managing index objects (the barrels)
<code>GlobalPositions</code>	map of indexed documents in all the barrels
<code>DataRepository</code>	metadata of all indexed documents

Table 2: Tanker's major inner structure

The whole structure of objects is illustrated in the Figure 9 and *Figure 10* on the page 68 and 69. The following chapters will describe each part of the hierarchy in further detail.

2.3.3 Distributor – details

`Distributor` is class used for channeling tasks and messages to specified objects (recipients) in sequence of objects (barrels). It can also distribute a request until it is fulfilled. It has two basic versions – `STE` and `MTE` class. The main difference between them is that the `STE` is used to distribute tasks in non-parallel way, while the `MTE` is designed for parallel multi-threaded distribution. The right implementation for the tanker is chosen according to tanker's constructor second parameter `MTE` – if it is `true`, the `MTE` class is chosen, `STE` otherwise.

2.3.4 Dynamizer - details

`Dynamizer` is another crucial class used for managing barrels. For this purpose it uses map of all barrels and slots in which the barrels are stored. Each slot just represents a socket in which a barrel is placed. In the `Egothor` namespace this map class is called `slotter` and it is implemented by `SlotMap` class (*Figure 9*, page 68). For now it is important to mention that every slot has its own directory on the hard drive where it stores data of documents of its barrel. That in the end means that on the hard drive every barrel is represented by a directory in the root index location and the directory corresponds with a slot (*Figure 7: Slot Items and*

Barrels, page 67).

During creation of the tanker a dynamizer is created over specified directory `indexDir` which is taken from the first parameter of tanker's constructor. If some index (hierarchy of barrels) already exists in this directory, the dynamizer is created according to the state this already existing index was in after its last commit (chapter 2.3.7 *Loading previous Tanker state* is dedicated to this subject).

The dynamizer implements methods for adding barrels into the index, for removing documents from the barrels and committing the whole process. For all these purposes it uses its `slotter` which offers the main operations over its map of barrels. It also calls merge algorithms for merging of parts of the index when it is necessary to optimize the structure (merging is further described in the chapter 2.3.12 *Merge*). More information about the dynamizer and its primary functions that are put in use is in the chapter 2.3.8 *Creation of documents*.

There is an article by RNDr. Leo Galamboš Ph.D. about dynamization in IR systems that corresponds with the dynamizer interface used in the Egothor project. It explains that this approach is very effective in this type of application [9].

2.3.5 GlobalPositions – details

As was mentioned before, each slot in a `slotter` can contain one barrel and every barrel can contain several documents. Every document indexed in Egothor has two ID numbers. One is a global ID which is unique within the scope of the whole Tanker. The second one is a local ID which is unique only within the scope of the barrel where it is saved .

`GlobalPositions` class is used for translation from the global IDs of all used document into their local IDs and the slot number where the barrel is stored. It stores this translation in a file in root directory of the Tanker (`indexDir`).

The `mapper` file is divided into consecutive blocks of size of 12 bytes in which the lower 4 bytes mean slot number and upper 8 bytes mean local id in that particular slot. Every such a record is stored in the file on the position `global_id * 12 [bytes]`, so therefore this mapping is 1:1. The opposite way of translation is implemented, too. It works as an analogy of this first process and it is stored in “doc.idx” file in every slot directory (not in the root directory of index where “doc.idx” file exists, too, but it has a different purpose). More description of this

specific file and a directory is in the chapter *2.3.9 Adding documents into the Tanker* where it is described along with its demonstrated usage.

2.3.6 DataRepository - details

As was stated before, `DataRepository` interface is designed for storing other data of documents than inverted lists into some persistent repository. So far there are two basic implementations of this interface in Egothor that are supposed to be used in the index. There are classes `DocumentsDB` and `Berkeley`. What they have in common is that they are supposed to be used every time a new document is added into the Egothor index. `DataRepository` implementation takes metadata (whatever it is - that relies upon user, upon instantiation of `Document` class, what kind of data is passed to it) and stores it into the persistent repository. Every record in this repository can exist in many revisions; the revision number is returned as a result of a method

```
public int DataRepository.addItem(  
    long key, byte[] data, int length).
```

Every time a user calls a query, all barrels are searched for hits and the result is combined with data stored in this repository and it all is returned as a result (more in the chapter *2.3.14 Query*).

2.3.6.1 DocumentsDB

`DocumentsDB` class creates a database on a hard disk in a root directory of the Tanker (`indexDir`) consisting of two files – the first file is named “doc.idx” (mind the difference in the location of this file and the one in *2.3.5 GlobalPositions – details* chapter – there are two different files with the same name but different location and purpose) and it represents index to the structure of the second file named “doc.dta” which contains metadata of all indexed documents.

Global ID of every document multiplied by 8 gives position in “doc.idx” file, where an offset to the “doc.dta” file is stored at. At this offset document's metadata information starts and may be read in the second file.

This implementation is gradually being deprecated and the Berkeley implementation is being rather used instead.

2.3.6.2 Berkeley

Class `Berkeley` is designed for storing document's metadata in a database using open source Sleepycat Berkeley DB software. Class `Berkeley` then just uses this database for storing the data and its revisions.

This `Berkeley` custom implementation that uses the original Sleepycat software is much more complex and suitable for Egothor. By its nature it is a key-value database and it is implemented in various data structures, such as B-Tree or hash table and more. It can also store more revisions of one document data, which is very useful especially in multi-threaded environment.

2.3.7 Loading previous Tanker state

A `Tanker` may be created over a directory which already contains some index. In this case a configuring file is loaded (state file in root directory named "state") and `Tanker` is set up according to settings saved in this file. This file is saved or updated when commit of the `Tanker` is called. State file contains list of identifications of all used slots, date of creation and number of slot that was used as the last one in the hierarchy while saving barrels (chapter 2.3.9 *Adding documents into the Tanker* contains detailed description of the whole file structure of Egothor).

Also all parts of the `Tanker` are configured according to the previous state – `GlobalPositions` and `DataRepository` load data and files that are relevant to them instead of creating new ones.

After all these objects are created, everything is set up for real work, for adding documents into the index or removing them and queries. All the necessary file structures are created on the hard disk and Egothor is ready to operate over it..

2.3.8 Creation of documents

As was stated before, the actual process of creating Egothor's documents is described in project documentation [1]. In the *Figure 8* on the page 67 there is detailed view of `Document`'s structure.

The important fact is that the created `Document` has a `TreeMap` of inverted lists of all terms of the document. A `TreeMap` is a Java data structure which implements some sort of a sorted

map of items. The process of indexation of all words in the document takes place during Document's instantiation and a result of this indexation is this TreeMap and it is accessible via `Document.getILLists()` method which is abbreviation for “get inverted lists”.

The Document class implements Barrel interface and BarrelReader interface which is designed for reading from the barrel. As it can be seen in the *Figure 10* on the page 69 the Tanker implements method `void append(BarrelReader)`. This is a crucial method designed for adding documents into the Tanker's index. This method takes as a parameter a BarrelReader via which it can read all inverted lists of terms of all documents and other data. Simply what the Tanker does is that it appends all the data read via BarrelReader interface into its inner structure of built index. The chapter 2.3.9 *Adding documents into the Tanker* contains detailed description of this process.

2.3.9 Adding documents into the Tanker

After Tanker is fully created with all of its parts it is ready to accept user's documents into its index. In this chapter there will be described the process of creating Tanker's index and further management of it.

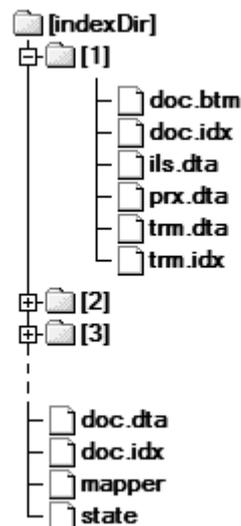


Figure 1: Egothor's index file structure

Each document's inverted list that is added into Egothor machine is saved in the file on the hard disk. The whole index then creates an ordinary structure of files and directories.

The final state of the index on the hard disk is as described in the *Figure 1*. Files saved in the

root directory `indexDir` have a special meaning as described in the *Table 3: Index root composition*.

Files saved in the subdirectories (slots) have their special meaning and it is described in the *Table 4: Slot (on disk barrel) composition*.

Filename	Meaning
subdirectories 1 - X	Stored barrels (slots)
doc.dta	Metadata of all indexed documents (<code>DocumentsDB</code> - 2.3.6.1)
doc.idx	Index over doc.dta (<code>DocumentsDB</code> - 2.3.6.1)
repo_patch, repo_text	Berkeley database files (<code>Berkeley</code> - 2.3.6.2)
mapper	Translation of global IDs of documents into local IDs in Slots (<code>GlobalPositions</code> - 2.3.5)
state	Configuring file

Table 3: Index root composition

Filename	Meaning
doc.btm	Bitmap of local Ids of deleted documents
doc.idx	Translation of local IDs of documents in the slot into global IDs
ils.dta	All inverted lists, it contains pointers into prx.dta
prx.dta	Proximities file – contains information about occurrence of words in concrete documents
trm.dta	List of all terms, it contains pointers into ils.dta
trm.idx	Index-sequential file for fast access to items in trm.dta

Table 4: Slot (on disk barrel) composition

2.3.10 The `Tanker.append()` process

The Tanker's `append` method takes its parameter `BarrelReader` and passes it to Tanker's `dynamizer` via `dynamizer.add(BarrelReader)` method. As already mentioned in the chapter 2.3.4 *Dynamizer - details* a `dynamizer` is used for managing barrels. An implementation of a `dynamizer` contains map of all slots in field `slotter` and some support fields to help organize barrels for merge algorithm and such (*Figure 9*, page 68).

In `dynamizer.add(BarrelReader)` method a free slot on a appropriate level of hierarchy of barrels is found first. The level depends on the number of documents the barrel

contains. If there is no free space for the new barrel on the level, then merge algorithm is called and it merges all necessary parts of index from the target level up and thus creates additional room for the new barrel. After this call the free slot for the new barrel is taken, the barrel is saved on the hard drive and barrel's handle is created and saved in the free taken slot. The saving process of the barrel itself is little more complicated.

After the free slot is found the saving process continues in the `slotter` (`dynamizer` contains `slotter` as a field). `Slotter` is responsible for creation of a physical barrel, for saving the barrel in its map of barrels (`HashMap<Slot, SlotItem> SlotMap.database`) and for returning the handle that is supposed to be saved in the free slot.

It creates a slot item implemented by `ThickSlotItem` (Figure 7, page 67), which represents a handle for the to be saved barrel and passes it the `BarrelReader` in the call `void ThickSlotItem.save(BarrelReader)`. Finally the process gets to the point of real saving the data from `BarrelReader` into the files on the hard drive. The `ThickSlotItem` creates `ThickBarrelOut`, an object that represents a barrel that writes its inverted lists to a single file, and calls its method `void ThickBarrelOut.append(BarrelReader)`. This call will append barrel from the parameter to the instance of `ThickBarrelOut`.

To point out one important fact, `ThickSlotItem` is an item representing a slot item, not the barrel itself. There are other classes, that represent the barrel in dependence on a usage – `ThickBarrelOut` for writing to the barrel, `ThickBarrelIn` for reading from the barrel, `ThickBarrel` for management actions with the barrel. For example, method `ThickSlotItem.getBarrel()` returns `ThickBarrel`, `ThickSlotItem.save(BarrelReader)` creates `ThickBarrelOut` to write to the barrel (just like was stated in the previous paragraph).

To get back to the saving process, an instance of `ThickBarrelOut` is created and its method `void append(BarrelReader)` is called. First it saves metadata of all documents held in the input barrel. As was stated before, this information is stored into `DocumentsDB` object that physically writes it into two files (“doc.dta” and “doc.idx”) in the root directory (`indexDir`) or using `Berkeley` class to store it in a appropriate database.

After that all inverted lists are stored into several files (described in *2.3.9 Adding documents into the Tanker* chapter) in the slot directory (`indexDir\slotNumber`) - the list of all terms is saved into “`trm.dta`” and “`trm.idx`” files using `TermsWriter` class (`ThickBarrelOut.terms`), the inverted lists with its proximities are saved into “`ils.dta`” and “`prx.dta`” files using `ProximitiesFileOut` class (`ThickBarrelOut.rf`) and `DataOutputStream` class (`ThickBarrelOut.writer`), and finally translation of local IDs of documents into their global counterparts is saved into “`doc.idx`” file (`ThickBarrelOut.docs`). The barrel is finally saved, so is documents' index with its metadata and thus handle of this `SlotItem` can be returned and saved in the free slot in the map of slots.

There is no black magic behind this file saving process – when the text says save data to a file, that means taking copy of the data from a barrel reader and just sequential save to a file, usually saving index of the data to another file (like `DocumentsDB` for example). How the data is organized in the sequential manner, that is another problem, but again it is not really important at this moment.

This whole process may seem little but confusing, but it was necessary to study it in every detail. This description is simplified, though.

2.3.11 Removing of a document

Removing of a document follows similar path like adding a document. It is initially called from `TankerImpl.removeDoc(int id)` method, where `id` parameter is a global Id of the document that is being removed. According to this `id` `TankerImpl` finds in its `GlobalPositions` a local `id` and a slot number of the slot in which the document resides. Afterwards these three numbers are passed to the tanker's `Dynamizer` via `removeDoc(int globalId, int slotNumber, int localId)` method – (*Figure 9* on the page 68 shows the described methods). `Dynamizer` then only calls the same method with the same parameters on its `slotter`, so again just like in adding a document, removing a document goes all the way through to the `slotter`.

`Slotter` then looks up in his database of all barrels a slot item corresponding to the slot number. This slot item can return a barrel (`ThickBarrel`) (*Figure 6*, page 66) by calling a

method `Barrel getBarrel()` (*Figure 7*, page 67). As was mentioned before, this `ThickBarrel` is for management actions with the barrel. It implements `Barrel` interface, where `boolean removeDoc(long id)` method is declared. So this method on a returned barrel is called and this is the point finally where a document is really removed.

`ThickBarrel` has got a field `Bitmap removedDocs`, which represents a bitmap of all removed documents in the barrel. And simply what happens now is that a bit is set to 1 in the `Bitmap` at an index corresponding to the document's `id`. From now this document's whole index no longer exists although it physically exists on the hard drive. Every time documents from a barrel are handled, those with flag “deleted” will no longer be taken into account.

For preservation reason of this information this `Bitmap` must be saved into some file. This action happens during commit phase, so this subject will be finished in chapter dedicated to it *2.3.15 Commit*.

2.3.12 Merge

As was stated in *2.3.9 Adding documents into the Tanker* merging is called during appending of a new barrel whenever there is no free slot on the level of hierarchy where the barrel belongs according to its length.

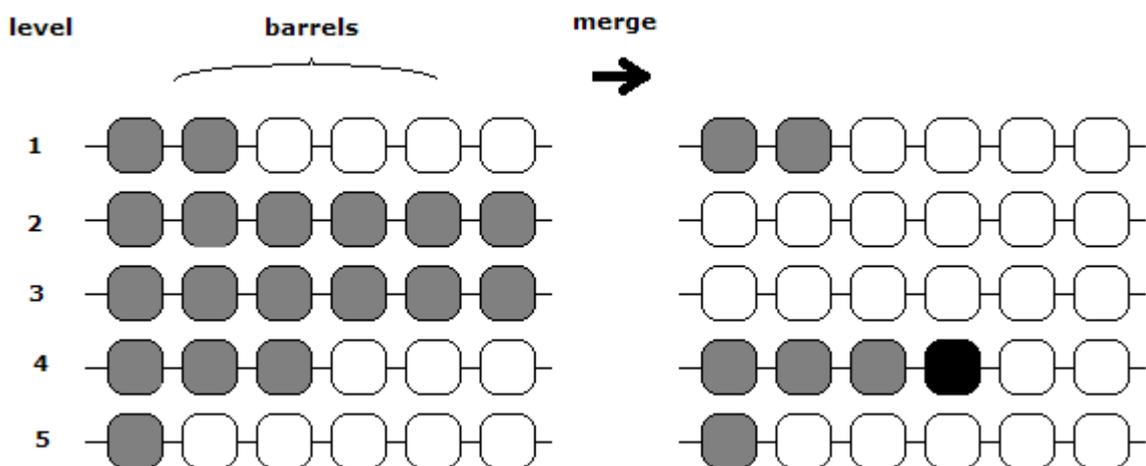


Figure 2: Merging

Merging process opens all barrels on that particular level of hierarchy or higher, if necessary,

loads them up, merges them into a new single barrel that contains all the documents and saves the new barrel on a higher level, in a new slot (new directory). The “merged out “ slots with all its barrel data in the directories are deleted.

If the higher level is full, too, the process continues recursively. An example of such merging is in the *Figure 2: Merging*, where a new barrel has to be put into the level 2, but the level is full, so is a higher level 3. The algorithm merges all barrels from the two levels, saves them as a new big barrel at the level 4, creating a free slot for the new barrel on the level 2.

That's how a new space for the new document is created, so it can be saved on that appropriate level of barrel hierarchy.

2.3.13 Listener

There is one more important fact in a class hierarchy that has not been mentioned yet, but it can be noticeable from the Figures. That is `Listeners`. Listeners' utilization idea comes out of their name – they listen and log actions made with barrels. Every object that moves, saves or destroys barrels in slots should implement a `Listener`. So far some of these objects have been encountered – `slotter` (`SlotMap`) or `ThickBarrelOut` (*Figure 7* and *Figure 9* on the page 67 and 68 show the listener fields). These object modify state of barrels in slots, so they must implement a `Listener`.

Every time an object modifies a state of a barrel in a slot, it should call listener's appropriate method for saving (`logSave(..)` + `logAssignment(..)`) or removing (`logRemove(..)`) a document. `Listener` writes this information to a log file which can be read from later. That moment, when listener's log file is read from, is during commit phase and it sums up all modifications to tanker and finishes the job, so it will be discussed it in the chapter dedicated to that subject – chapter *2.3.15 Commit*.

2.3.14 Query

Queries are processed over all tanker's loaded barrels. For this point of time it is important that query may read all saved barrels while searching for hits, looking for a match in their inverted lists. After some matches are found it reads from `DataRepository` (`docDB`) to fill in other document's information, where the hit was found and the completed result is then returned.

2.3.15 Commit

After Tanker gets some amount of work done it is time to commit all changes. Commit is important because no work done is visible to the outer world before commit. It starts at the top level – at `BufferedTanker` level or at `TankerImpl` level and it goes down through the whole hierarchy of the composite Tanker and commit is called there on every level, where in dependence of responsibility saving of particular data is taken care of.

At `BufferedTanker` level all cache memories are flushed first. That means all processed barrels that were held only in cache memory are appended to the lower Tanker implementation, in our case to `TankerImpl`. After that commit continues in the lower level Tanker implementation.

At `TankerImpl` level final state of the whole tanker as one entity is saved into the “state” file in the root directory of the tanker – in `indexDir`. This state file was already described in chapter 2.3.7 *Loading previous Tanker state*. Dynamizer fetches list of all present barrels and their slots plus additional information about last used slot number and all this is saved into the state file.

At ascendant's level – at `Tanker` level – dynamizer's commit is called first. This process covers going over every slot item in the database of the slotter and calling `close()` method on every each one of them. Afterwards the whole database of slot items is erased (just the slot items, not the barrels themselves).

Closing of slot item from this `slotter.database` means closing of barrel (`ThickBarrel.close()` in *Figure 6*, page 66), which in particular means flushing and closing of barrel files in the slot directory. As was mention before list of these files includes terms files, id translation file and a bitmap of removed documents .

Finally there is one more thing left to do – writing mapping of global ids of documents to slots and local ids into the mapper file. The process goes over all changes that were written into the log file by `listener` and for all appended or removed documents the mapper file is updated (their record is added/updated in the file).

Now the whole commit phase is finally done. Review of the Tanker's state after it follows:

- every barrel from database is flushed to the hard drive and closed (corresponding slot directories with the barrel files)
- flags of deleted documents are saved, too
- database in the `slotter` is cleaned
- metadata and translation of global ids of all active documents into slot numbers and local ids are saved in the root directory of the Tanker
- final state of the whole Tanker is summarized in the state file in the root directory

After commit the Tanker can close up (method `TankerImpl.close()`) which closes all opened files, stops distributor, saves `dynamizer`'s slot items again and deletes `listener`. Also, after commit another Tanker can start in the same root directory. It can load up all flushed configuration files and be in the same state as the first Tanker was in after commit. Although there is no concurrency control, so these two Tankers would interfere with each other and have fatal conflicts. And here it gets to the point of my part of work on the project. After analyzing of Egothor, of all the processes and classes, I have to analyze suitable concurrency control approaches and implement the most suitable one, so parallel execution of Egothor machines over the same index directory and data is safe and correct.

2.4 Conclusion

This chapter described functionality and implementation of Egothor in index processing. Now it should be clear how the whole process works, how the index is created from the very beginning when a document is passed to the system and ends with the last commit. The state of the project as it is now is that it is not possible to run more than one machine over the same index directory without fatal consequences. The desired state is that there is implemented some concurrency control mechanism, that takes care of parallel execution of Egothor machines. This is a good point to start at with concurrency designs and keep on mastering them to create effective and functional multi-threaded solution.

3 Concurrency control

This chapter will cover major concurrency control approaches and mechanisms in relation to Egothor aspects. First it will go over general overview of transaction processing terminology. Then it concentrates on the Egothor project and the specific problems that exist there and that need to be solved. Afterwards it describes some more or less suitable concurrency control protocols with overview of their advantages and disadvantages.

3.1 General introduction to transaction processing and terminology

Transaction processing is in computer science processing of information that can be divided into standalone and indivisible operations, that are called *transactions*. Every transaction must finish its work with success or failure as a complete indivisible unit. It cannot remain in an unfinished intermediate state leaving used resources in an inconsistent state [3, 5].

3.1.1 Transaction

Definition of an transaction is considered to be as follows: *Transaction is an operation with some data that has four properties (ACID) – atomicity, consistency, isolation, durability* (definitions taken from [2-6]).

- *Atomicity* means that the operation is performed completely as one unit or it is not performed at all. In case of a failure of any operation, effects of all operations that make up the transaction should be undone, and data should be rolled back to its previous state.
- *Consistency* means that on completion of a successful transaction, the data should be in a consistent state. In other words the operation will not damage the data, it will transform the data from one correct state to another.
- *Isolation* means that each transaction should appear to execute independently of other transactions that may be executing concurrently in the same environment. The effect of executing a set of transactions serially should be the same as that of running them concurrently. No transaction is allowed to see intermediate state of data of other transaction.
- And finally *durability* means that once the operation is finished, its result is persistent, it can survive some sort of failures.

Transaction's characteristics assure correctness of parallel operations with data. How to ensure

these four characteristics is another problem. There are several ways how to solve it. There are transaction processing systems that take transactions as an input and schedule them in such manner, so the ACID characteristics are assured and so is correctness of concurrent execution.

How schedulers work that is the main difference among them. There are several ways how to control transactions execution. Every way has its pro-and-cons, every way can be implemented in aggressive or conservative way (that means how much it postpones transactions that may be in conflict with other ones) [2, 3, 5]. And every way is more or less suitable for particular applications. Some of the approaches of concurrency control will be described and discussed along with their suitability for Egothor in chapter 3.3 *Concurrency control approaches*.

3.1.2 Rollback

During life of a transaction many unexpected events may happen. The transaction can find out that it cannot preserve its data, that it is in conflict with other data, a better (newer) one. Or it can simply crash and thus it leave unfinished job and data in inconsistent state. In all these occasions some action is needed to return the data to its previous consistent state. This action is called *rollback* and after its completion the database is unaware on the rolled back transaction as if it never happened [2-6]. Sometimes this is implemented by recording intermediate states of the database called before images and if transaction cannot be committed, these images are used to restore the database to its previous state [3].

3.1.3 Deadlocks

Sometimes two or more transactions may try to access the same part of a database at the same time and it happens in such a way that the situation prevents them from further proceeding. That happens when transactions lock the resource they access for their exclusive use. For example, transaction A may access data X of the database and locks it for its usage, and transaction B may access data Y and locks it for its usage, too. If, at that same time, transaction A then tries to access Y and transaction B tries to access X, a *deadlock* occurs, and neither transaction can move forward, because they both block each other on the resources they already have locked and cannot access the other data locked by the other transaction [2, 3, 5, 8].

For transaction processing systems it is necessary to detect these deadlocks when they occur or prevent their creation. In case a deadlock already occurs, a typically solution of this problem is that both transactions are canceled and rolled back, and then they are started again in a different order, automatically, so that the deadlock doesn't occur again. Or sometimes, one of the deadlocked transactions is chosen as a victim and it is canceled, rolled back, and automatically restarted after a short delay, so the other transaction may continue on [2, 5, 8]. Deadlocks can also occur between three or more transactions and that is a worse situation. The more transactions involved, the more difficult they are to detect. Transaction processing systems have a practical limit to the deadlocks they can detect. If it is possible, the best solution is if deadlocks do not occur at all. Unfortunately it is possible only in some special cases, if the application's design allows that kind of data access.

3.1.4 Starvation

In concurrently executed transactions another specific problem may occur, a *starvation*. In a model situation a transaction A tries to access portion X of the database, but at the same time this portion X is already accessed by transaction B. So thread A cannot continue and must wait for B to release portion X. After B releases portion X some other transaction C tries to access portion X just before A can make it in time and it succeeds. When A comes to the point of making another try to access X, it is refused again because transaction C has accessed it. So A must wait again until C releases portion X. This can go on and on in the same scenario and cause A to never access portion X of the database. Transaction A would be starved out [8].

Another case in which this type of situation may arise is when dealing with deadlocks. After a transaction processing system finds a deadlock, it can choose one transaction as victim and roll it back and restart again. As a consequence this transaction would start its executing all over and thus may happen to get into the same deadlocked situation. Transaction processing system then chooses a victim again and may by chance choose the same transaction again and roll it back and restart. Just like in the previous case there is a chance that this scenario would repeat itself infinite times and so the involved transaction would never get a chance to finish its work [8].

3.2 Concurrency in Egothor

After description of inner processes of Egothor and its index handling in chapter

2 Documentation of Egothor – index creation part and after short introduction into the transaction processing terminology it is possible to analyze behavior of concurrently executed processes in Egothor over single index location that randomly access any part of the index either for reading or for modifying. Typical reader-writer and writer-writer conflicts appear.

3.2.1 Project terminology

To establish some kind of Egothor terminology it is needed to give names to entities that will appear in model situations. For the threads the names are given according to their type of activity they are performing over the index. In case of writers there is a requirement to preserve ACID characteristics, namely isolation, and it forces to split writers into two different entities – committer and modifier, because no changes of the index should be visible to the outer world before commit.

- *Reader* – Reader is called every thread that reads any part of the index.
- *Modifier* – Modifier is called any thread that appends documents into the index or removes them from it. Because all these changes should be visible to the outer world only after its commit, modifier is defined to be in the state before commit.
- *Committer* – Committer is any thread, that is in state of committing its previously done modifications to the index.

It is clear that a thread slightly moves from the modifier state to the committer state as it first makes modifications to the index and then it performs commit of the whole work.

3.2.2 Model situations and requirements

All three types of threads have different roles and different behavior and demands.

- Reader
 - *Model situation:* As usual, readers perform only operations with the index that do not alter it. Thus number of concurrent readers is not limited for the single index.
 - *Special requirement:* On the other hand readers have a special requirement according to major project requirement number 5 (*1.2 Assignment requirements*). After they start to work with the index, there must be ensured that they work with the same data for some amount of time. That means that no modifier nor committer may change their image (snapshot). This requirement is demanded because of the real usage of web full text index, when the reader makes a search query for some key words, the system

returns to him a set of hits (count of hits limited by some number) that it has found. The reader then decides to search for the very next amount of hits, hits that consecutively come after the first set. If any other thread alters the index at the time between the first search query and the second one, it can alter possible relationship between those two sets of hits that would be returned. As a consequence the set of the next hits would not be exactly the next consecutive set and that is undesirable.

- Modifier
 - *Model situation*: Modifiers have a special role. They are supposed to do modifications to the index, but nothing should be visible to the outer world. That can lead to an idea, that it should be possible to allow unlimited number of modifiers into the index to do their modifications. If their isolation characteristic is ensured, their parallel execution is correct then.
 - *Special requirement*: No change done by any modifier is visible to other threads.
- Committer
 - *Model situation*: Committer is in state when it has finished all desired modifications to the index and it is doing committing of the changes, so every other thread from the point of time after commit can see it all. If it is possible to allow more than just one committer in the index at the same time, it depends on the committer's index usage and it will be discussed in the chapter 3.2.3 Access granularity.
 - *Special requirement*: Every modification is visible to the outer world.

3.2.3 Access granularity

As was stated before, Egothor manages its index by merge technique. In practice that means that whenever a barrel with documents is being appended to the index, it is put at a specific place in dependence of the number of documents it contains. More precisely it is placed at a specific level of the barrels' hierarchy into a free slot (discussed in the chapter 2.3.10 The Tanker.append() process). When the algorithm cannot find a free slot at the desired level, it merges all barrels at that level to a new bigger barrel, that contains all their documents, deletes the smaller barrels and then this new merged barrel is placed at a higher level and, thus, leaves a free slot for the original barrel to be put in.

This process can trigger a whole cascade of merging, because as the new big barrel created by merging of all the barrels at the specified level is being placed at the higher level, the whole situation can repeat. At the higher level there can be no free slot left for the new big merged

barrel, so algorithm merges all barrels at the higher level to a new bigger one along with the first merged barrel and tries to place this second big merged barrel at a even higher level of the hierarchy, where the situation can repeat again and then again. So after couple of recurrent merges it finally finds a free slot at some high level of hierarchy and there it puts a new merged barrel.

As a consequence a single append of a barrel may cause reorganization of the whole index, changing slots where documents are stored to newer ones. This leads to a granularity requirement of access control to the parts of the index for committers and modifiers during their execution, which comprises the whole index.

For readers and their search queries the index contains zero or more potential hits. Every barrel in the index is potential place where the hits can be found. This points to scale of index a reader needs to have access to during its execution – the whole index again. Additionally the index must remain constant, at least the image the reader starts working with, for some amount of time.

3.2.3.1 Granularity problem

For modifiers and committers the situation is much more complicated, because no change of a modifier is allowed to be visible. For committer it is the right opposite.

The problem may be put in a different way, resulting into a new problem. It is not enough just to disable commit for modifiers. If two or more concurrent modifiers alter the whole index by appending barrels and merging of them. At the end they all decide to commit. There would be two and more totally different images of the whole index, two and more sets of differently merged barrels with differently appended and merged new barrels. To resolve the common parts of the many images, the new parts and to shape it all to a final index consisted of neatly placed and merged barrels and to do it all in a time and space effective way according to the major requirements, that would be a hard task to accomplish.

Of course there exists a solution, an easier and more effective way and that is that modifiers will not be allowed to change the main index before commit and they will just gather their changes in a local place. And after they are done with gathering, they switch their state into commit phase and start to insert the local changes into the global index. That way only a single modifier/committer at one time will be changing the actual index, merging barrels and reorganizing the whole structure. After he is finished another one can commit his own

different changes, appending and merging barrels at state the previous one left them in into a new state. This technique is called **write ahead logging (WAL)** and it is commonly used and popular technique to provide atomicity and durability in database systems [2, 3].

The access granularity issue has been solved for all actors, the following *Table 5: Access granularity* sums up the results.

Type of thread	Access granularity
Reader	The whole index
Modifier	None (write ahead logging)
Committer	The whole index

Table 5: Access granularity

The previous table will be referred to when different types of transaction processing algorithms will be discussed along with their suitability for Egothor.

3.2.4 Deadlocks in Egothor

Fortunately deadlock will not be the issue in the Egothor engine. In an index processing the only way how a deadlock could occur would be a situation when one thread wants to read from two or more indexes at the same time or a modifier/committer wants to alter two or more indexes at the same time, too. From the way the processes in the Egothor work, this will never be the case. One thread works at one time with a single index location; and that is an invariant.

3.3 Concurrency control approaches

In practice there exists couple of transaction processing algorithms, that are used to correctly provide concurrent access to a database or to any other type of resource. They are usually implemented as **schedulers**, that take care of correctness of parallel execution [2, 3, 5, 6]. They differ in many aspects and therefore every each one of them is suitable for different type of application. This chapter goes over basic types of these algorithms and describes them and evaluates them in relation with the Egothor functional requirements. It will not go over further theory of the methods, it will concentrate on the mechanisms, implementation and suitability that should help to decide, which method is the right one for Egothor.

3.3.1 Locking

Locking is the most popular and widespread way how to ensure correct parallel database access. Though it has some issues that need to be solved separately, it is still the most used algorithm in many commercial products that offer transaction processing solution [5].

3.3.1.1 Basic terms

As title of this method foretells, this algorithm introduces locks that prevent access to potentially inconsistent states. Every access to a database must be covered by an appropriate lock. If so, the transaction is defined to be a **well formed** [2, 3, 5, 6].

Different implementations of locking introduce different types of locks. Though there are two basic types named after the two basic data action – **read lock** (RL) and **write lock** (WL). Such a transaction processing system offers along with standard common transaction operations, such as **commit**, **abort**, **read** and **write**, a new ones, precisely **read_lock**, **read_unlock**, **write_lock** and **write_unlock**. All types of offered locks form can be arranged into a conflict table, which tells what locks are *exclusive* (are allowed to be held only by one transaction) and which ones are *shared* in relationship with all other locks. An example of such a trivial table is in the Table 6: Basic lock conflict table [5, 6].

Transaction A has	Nothing	RL	WL
Transaction B wants			
RL	+	+	-
WL	+	-	-

Table 6: Basic lock conflict table

A plus sign means that if a transaction possesses a lock written in the head row and other transaction wants to acquire lock written in the head column, it is allowed to. In other words the possession of the lock is not exclusive. A negative sign means a right opposite. All transactions must use locking in order to the conflict table. Then their execution is defined to be **legal** [2, 5, 6].

3.3.1.2 Scheduler

The way a scheduler of transaction using locking works is whenever a request on some portion of data comes as an input into it, it checks that the request is not in collision with any

other existing lock on the portion of data. If so, it logs the requested for the same future checks with other requests and tells the caller that it is allowed to proceed. If the request conflicts with any other lock, it is postponed [2-6].

There are two ways how to actually implement basic behavior of such a scheduler. It can either be **conservative** or **aggressive**. Conservative scheduler tries to avoid rejecting requests because it is not able to continue on. It tends to delay them, tries to get as many information it can get to reorder them to assure their further execution rather than rollback. In locking implementation it demands all requests a transaction may need in advance. Then it tries to lock everything for it, so no deadlocks would appear and the transaction would have assured that it can continue with its execution until its very end. Aggressive implementation on the other hand deals only with real time information, it does not do any reordering and thus it is forced to reject a request completely sometimes [5].

3.3.1.3 Deadlocks

Deadlocks are a big issue in locking algorithm. It is a typical method where this problem occurs. There are many ways how to solve it. The system may either try to avoid it or let it happen and solve it afterwards. A deadlock can be detected in using **wait-for-graph**, where transactions are nodes and there is an edge between transactions A and B if transaction A waits for a resource transaction B has in its possession (has lock). Whenever there exists a circle in such a graph, the circle represents a deadlock [2, 5].

On a deadlock detection there needs to be picked a victim transaction that is canceled and restarted, releasing any held locks so the other transactions may have chance to acquire it and continue with execution.

As was stated in 3.2.4 Deadlocks in Egothor, deadlocks will not be the case in the Egothor, so there is no need to get further into deadlock's theory.

3.3.1.4 Granularity

Granularity of locks is another great issue in locking method. The problem is how coarse the locks should be. Each way has its pro-and-cons and it is possible to create a model situation that would be processed slowly and inefficiently for every way [5].

A coarse granularity brings longer and more frequent delays while waiting for a lock, on the other hand it has low overhead, because there is not so many locks to manage.

A fine granularity brings short and less frequent delays while waiting for a lock, but it has

great overhead as the number of locks increases and additionally it brings greater chance of deadlocks.

3.3.1.5 Evaluation

Locking with its mechanisms seems to be very suitable for the Egothor processes. The main locking questions can be answered with the project's specifics.

- Lock types - There are reader and writer processes, that would eventually try to achieve read or write locks.
- Deadlocks would not appear from the nature of Egothor processes as discussed.
- Lock granularity would copy the needs of threads' access granularity as discussed in the chapter *3.2.3 Access granularity*.

It is easily implemented even in the case that there can be no central unit of the whole transaction processing system, as is required in 1.2 Assignment requirements. It can simply be done by file locks.

There would, of course, emerge new problems as **starvation**. As read locks are shared types of locks, read lock request would not have a problem, if there were different consecutive read requests. At the same time if a write lock request appeared, it could not proceed along because of conflicting read locks and it would have to wait. Still if some other read lock requests were coming in, they could just proceed on, because they do not block each other, outrunning the write lock request and making it to wait even longer, even into infinity. This problem would have to be solved.

Another question comes up when checking the major project requirement in comparison with the functionality this method offers, precisely it is requirement number 5 (the index constancy). To fulfill this requirement only by using locks, it would be necessary to keep the data locked with read lock during the whole duration of thread's process with the index. Consequently it would lead to great delays of conflicting lock requests, in this case write lock requests. There is no basic limitation of the period during any thread would be allowed to have the index constant, so this problem would be a big issue. Write locks could be starved out or at least greatly delayed very easily and very frequently.

Still it is correct to say that locking is a suitable method for Egothor. It has got some issues that must be solved, but if it is possible to work them out in an efficient way, it would serve

the task well.

3.3.2 Timestamp ordering

Timestamp ordering is a technique that does not use locks. It is not currently used in many commercial products, but it provides an alternative to locking algorithms [5].

3.3.2.1 Basic terms

Timestamp ordering assigns a timestamp to every transaction. Timestamps are from a totally ordered domain, so to every two different transaction $T_i \neq T_j$ is assigned a timestamp $ts(T_i) < ts(T_j)$ or $ts(T_i) > ts(T_j)$. Each operation $p_i[x]$ is then stamped by a timestamp of its transaction. The algorithm is ruled by a *timestamp ordering rule (TO rule)*: Conflict operations are ordered by their timestamp [2, 5].

More accurately and formally *the timestamp ordering rule* [5]: if $p_i[x]$ and $q_j[x]$ are conflicting operations, then the scheduler processes $p_i[x]$ before $q_j[x]$ if $ts(T_i) < ts(T_j)$.

3.3.2.2 Basic timestamp ordering scheduler

An approach to implement the timestamp ordering rule may be a simple, optimistic, aggressive scheduler. It lets all incoming transactions proceed right away with the only exception of those that arrive “too late”. Operation $p_i[x]$ is *too late* if it arrives after the scheduler has already output a conflicting operation $q_j[x]$ such that $i \neq j$ and $ts(t_j) > ts(t_i)$. If p_i is too late, it can no longer be output without violating the timestamp ordering rule. Thus, $p_i(x)$ must be rejected, which implies that it has to be aborted. It can then be restarted later, at which point it will receive a timestamp with a larger value so that some, or ideally all, of the previous conflicts will now appear in the right order [3, 5].

The implementation described in previous paragraph is called **basic timestamp ordering scheduler** (BTO) [5] and in order to function correctly it must record the following timestamps for every data item x :

- *max-r-scheduled(x)*: the value of the largest timestamp of a read operation performed on x that was already allowed to proceed on;
- *max-w-scheduled(x)*: the value of the largest timestamp of a write operation performed on x that was already allowed to proceed on;

These records will be used in order to determine whether an operation has arrived too late and the decision algorithm is simple [3, 5]. When some operation $p_i(x)$ arrives, then $ts(t_i)$ is compared to $max-q-scheduled(x)$ for each operation q that is in conflict with p . If $ts(t_i) < max-q-scheduled(x)$ is true, $p_i(x)$ is rejected, because it has arrived too late. Otherwise it is allowed to proceed on and $max-p-scheduled(x)$ is updated to $ts(t_i)$ if $ts(t_i) > max-p-scheduled(x)$.

Another important point is that the scheduler has to make sure that the low level operations with data are executed in the exact order it initially orders them using the TO rule [3, 5, 6]. A BTO scheduler can send an operation $p_i(x)$ ready for being scheduled to the data manager level, where it performs its low level data operation, only if every conflicting and previously sent $q_j(x)$ has been executed and finished. Therefore the data manager has to acknowledge the execution of every operation and the scheduler has to wait for the acknowledgment for operations in conflict that are to be scheduled in the timestamp order.

3.3.2.3 Implementation issues

There must be solved an issue of *totally ordered domain of timestamps*. Timestamp assigning can be implemented by using a counter, which increments every time it provides a new timestamp number.

Another possible implementation is to use a local clock, but this may be little tricky, as it must be assured that the clock always ticks between two timestamps. The timestamp resolution is crucial for the correct behavior of the system. As the minimum time elapsed between two adjacent timestamps increases, the possibility that two or more timestamps are equal increases proportionally, too, and that enables some transactions to proceed and commit at incorrect order [5].

A difference between a conservative and an aggressive implementation is also relevant. If an aggressive scheduler receives operations that largely differ from the timestamp order, it will reject too many of them, causing to many complete aborts of respective transactions, which is not very effective.

This suggests to implement more conservative way, such as a variant in which it is possible to block operations artificially. When an operation $p_i(x)$ is received, a conflicting operation with a smaller timestamp could be received at some time later. Thus, if $p_i(x)$ is delayed, such conflicting operations hopefully arrive in time and are enabled to cut in front of $p_i(x)$

according to their timestamp and processed on instead of aborting. This clearly brings a trade-off between artificial delays that slow down the whole transaction processing and spared aborts. The choice of a time period for which an operation is artificially blocked can be a critical performance factor [3].

3.3.2.4 Evaluation

It is clear that timestamp ordering is not very suitable for the Egothor. Mainly because of the basic concept of timestamp ordering itself, where especially frequent aborts are the main inappropriate feature.

As was discussed in the chapter 3.2.2 Model situations and requirements and 3.2.3 Access granularity, once a committing thread starts to commit all its modifications, the whole index may be restructured and reorganized, which is done by a large number of basically I/O operations. Aborting and restarting of such a transaction would be very expensive, especially if it is close to its end. Any time some new transaction with new timestamp may come into the scheduler and may get a free way to proceed, causing the old one to abort.

To get a timestamp only for a single portion on an index is also impossible because of the whole merging technique and its impact, when a single addition of a barrel may lead to a need to reorganize the whole index, thus interfering with any data that was accessed by a transaction with newer timestamp. There would have to be granularity of operations with timestamps the same as declared for model situations in 3.2.3 Access granularity. That would probably solve the problem. On the other hand it would get too close to the idea of locking the whole index.

In addition when tried to be implemented as decentralized system with no central unit as demanded in 1.2 Assignment requirements, this idea of stamps gets really close to the locking implementation and its additions described in 3.3.1 Locking. The idea may look like there are timestamp files instead of lock files and threads behave accordingly to the algorithm and the stamps. Therefore it is much clearer to think about locking straight away than trying to bend timestamp ordering implementation to work it like locking.

Taking everything into account, timestamp ordering is not very suitable for Egothor.

3.3.3 Multiversion

So far a general assumption has been that data items exist in exactly one copy each. As a result, write operations overwrite existing data items. Multiversion concurrency processing algorithm introduces a possibility that more than one copy of each data is allowed [3].

In transaction processing systems multiple versions of data are stored for recovery purposes anyway, so extending this approach little bit further is not far away from usability. On the other hand theoretical considerations tend to assume that space is an unlimited resource, which in practice is just not true. That is why there are limits for this concurrency algorithm and there is a cost to be paid for storing multiple data [5].

3.3.3.1 Basic terms

The basic idea is that write operations create a new version of data instead of overwriting the old one, and read operations must specify which version they want to read from. That way write requests never collide with anything and operations that arrive too late are handled in a better manner [5].

The basic rules for readers that need to be followed are the following three [5]:

- A reader reads only those version that someone has already written.
- When a reader wants to read a data, he reads his version of the data in preference.
- If the reader reads uncommitted data, he cannot commit before the one who wrote that version of data, because if that writer aborted, the reader would have read nonexistent versions.

3.3.3.2 Schedulers

Multiversion scheduler can be implemented in several ways. A method using combination with **timestamp ordering** schedules every operation immediately. Additionally it processes every request as follows [3, 5]:

- Each read $r_i[x]$ is translated into $r_i[x_j]$ where j is the largest available timestamp for x that is smaller than or equal to i (A reader reads only those version that someone has already written, he reads his version of the data in preference).
- Each write $w_i[x]$ is translated into $w_i[x_i]$ where i is the transaction timestamp.
 - If a read $r_j[x_k]$ such that $ts(T_k) < ts(T_i) < ts(T_j)$ occurred, reject write.
 - If no such read exists, accept write.

- For recoverability, commit_i is delayed until commit_j for all T_j that wrote versions read by T_i

To put it in other words a reader reads only those version, that someone has already written and he reads his version of the data in preference; if the reader reads uncommitted data, he cannot commit before the one who wrote that version of data, because if that writer aborted, the reader would have read nonexistent versions.

Refused write request cause the respective transaction to be aborted and restarted.

Another way how to implement multiversion is to use **locking**. There is *two phase locking* used, which means that all lock operations precede all unlock operations. The scheduler works as summarized below [5]:

- Each read requires a read lock on the item being read
- Each write requires a write lock on the item being written
 - That way a write lock prevents reading of the locked item but not its earlier version and it prevents also creating of a new version of the item.

The scheduler then uses three types of locks – read, write and certify lock. Read lock only collides with certify lock, but does not with write lock, because there is always a previous version available for reading. Write lock collides with other write lock and certify lock, but does not collide with read lock, because a new version is being created. Finally certify lock collides with read, write and other certify lock [3, 5].

When a write operation $w_i[x]$ is received, the scheduler tries to set write lock $wl_i[x]$. After it is done, the write operation is converted into $w_i[x_i]$ and scheduled. When a read operation $r_i[x]$ is received, the scheduler tries to set read lock $rl_i[x]$. Once this lock is set, if transaction T_i already owns write lock $wl_i[x]$, the read is converted into $r_i[x_i]$ and scheduled. Otherwise the read is converted into $r_i[x_j]$, where x_j is the last committed version of x . The third type of locks the certify lock is used in commit. When a commit request is received, the scheduler tries to convert all write locks $wl_i[x]$ into certify locks $cl_i[x]$. This causes the scheduler to delay the commit until all read locks on the desired data are released [5].

The lock conversion, of course, can lead to a deadlock. On the other hand such an implementation does not waste too much space, because it does not keep many uncommitted or old versions of data.

3.3.3.3 Evaluation

Multiversion concurrency algorithm has one great benefit – it fulfills one of the major requirements, precisely number 5, the index constancy. Using multiversion every reader would have a version of the data for itself for as long as the space would be available without blocking the writers. That is exactly what the search queries of readers require.

Despite that, there is an obstacle in full implementation of this method for all processes of the Egothor. Namely there is the merge technique that is capable of changing the whole index. Upon every write or append to the index there would be a new version created. The version would be of the whole index, which is very unpractical and inefficient. It would waste too much space, because the index may be very large. Keeping of couple of these versions would lead to disk space shortage.

Another problem would occur at commit phase. With many versions of the whole index it would be difficult to resolve the final shape of all the barrels. One thread would merge the index in a different way than the other in dependence on their input. This points to a solutions, again, that modifiers would use **write-ahead-logging** and only commit would alter the whole index, one thread at a time only. There would be different versions of data only for readers.

The locking version of multiversion seems to be practical for Egothor. There are states of the process that would use locks to seclude away from conflicting processes. That would be enough to use just two types of locks – read and write lock, because modifiers would not do any changes to the global index. Once read locks are released, the commit may begin, after which there would be a new version of index. There is still a catch, because the subsequent reads have to read their version of the index, not the last committed. So there are some changes needed in the multiversion algorithm to fully fit the Egothor's needs, but it can be arranged and implemented with efficiency, when the reader can read version other than the last committed.

To put everything into account, multiversion with locking method and with some issues solved is very suitable for Egothor.

3.3.4 Certification

Certification concurrency control, also called *optimistic concurrency control*, is another method of parallel execution control that does not use locks. It is based on a assumption that conflicts of transaction among each other are very seldom. It is a very aggressive type of scheduling which is not used in many commercial products recently [3, 5].

3.3.4.1 Basic terms

Scheduler based on certification assumes that transactions do not conflict and so it schedules every request immediately. If there really was a conflict it checks at the commit of every transaction. If it finds any, it aborts the respective transaction [3, 5].

Such a scheduler can be based on many algorithms like *two-phase locking*, *timestamp ordering* or *serialization graph testing*. Each implementation functions at the same bases, but their verification phase on commit is different, relating to the specific algorithm. Every type of a scheduler upon receiving an operation records the operation and data item involved and then it schedules the operation for immediate execution [3, 5].

3.3.4.2 Schedulers

Two-phase locking (2PL) implementation of certification scheduler in its verification phase checks all operations executed by the transaction against all conflicting operations executed on the same data by other active transactions. If a conflict exists, the transaction is aborted. If there is no conflict, the transaction is committed.

To illustrate the locking base of this method, the algorithm of verification follows [5]:

- When the scheduler receives $r_i[x]$ or $w_i[x]$, it adds x to $r\text{-scheduled}[T_i]$ or $w\text{-scheduled}[T_i]$.
- When the scheduler receives commit_i , $r\text{-scheduled}[T_i]$ and $w\text{-scheduled}[T_i]$ contain set of all reads and writes of transaction T_i and they are checked if there is a conflict.
 - $r\text{-scheduled}[T_i] \cap w\text{-scheduled}[T_j]$,
 - $w\text{-scheduled}[T_i] \cap r\text{-scheduled}[T_j]$,
 - $w\text{-scheduled}[T_i] \cap w\text{-scheduled}[T_j]$ are test if they are empty for all active transactions T_j .

Because such a scheduler does not keep a track of the order of the operations, it sometimes aborts transactions that would commit successfully in ordinary two-phase locking scheduler. Additionally it is not that effective as it may seem. It has been tested that under low conflict rates it works about as well as basic 2PL scheduler, even it saves on locking management overhead. Beyond that under high conflict rates it wastes execution time and resources by completing all transactions until their commit, even those that will be aborted [5].

Another way how to implement certification scheduler is to use **timestamp ordering (TO)** [3, 5]. It differs only at the verification phase and the difference in the process is that when the

scheduler receives a request to commit a transaction, operations executed by the transaction are checked against all conflicting operations executed on the same data by other active transactions. If the conflicting operations violate timestamp order, the transaction is aborted. If the conflicting operations follow timestamp order, the transaction is committed. Data structures of such a scheduler are the same as of the basic TO scheduler and the processing conditions are the same also. Basic TO scheduler is preferable in practice, because it does not let transactions that are to be aborted to run until the end [5].

3.3.4.3 Evaluation

Taking into account requirements explained in the chapter 3.2.2 Model situations and requirements and 3.2.3 Access granularity it is clear that certification schedulers are very unsuitable for the Egothor.

It gets to the point when two or more concurrent committers alter the index at the same time, merging and appending the barrels at their own will, causing fatal conflict with each other. These conflicts would lead into final shape of index which would be hard to recover from in case of transaction abort. They all would start with the same index, but every each one of them would alter the index in a different way taking different path of merging depending on its input. This could be solved by merging the barrels in every writer's own private sandbox and after verification the changes could be just copied out to the global index.

On the other hand if any other transaction would be in conflict with the verified one, it would be aborted, causing all its work to be tossed away and executed again from the beginning. This scenario may repeat many times causing execution of all the I/O operations over and over again. Therefore this approach is very inefficient and it wastes too much execution time and I/O operations with the hard drive, which is very expensive [8].

All in all, concurrency control based on certification is very unsuitable for Egothor.

3.4 Summary and sketch of solution

After examination of four basic concurrency control algorithms it is possible to sum up all the results and to propose a winner, a method that will be implemented in the Egothor system.

As was already stated in all evaluation chapters, timestamp ordering and certification ordering are not very suitable mainly because of the principles of their functionality in relation to the way the index is managed and space and I/O operations efficiency. It is correct then to reject

them as possible solutions and concentrate on the other two methods.

In the Evaluation chapters of each method there was described that both methods have some advantages and some disadvantages, too. A combination of the methods, though, solves the local problems. At points where one method is not sufficient, the other one supplies a solution.

Only as a sketch of the offered solution can be prefaced the basic locking algorithm providing mutual exclusion of concerned transactions while accessing the index for modifications or reading, and multiversioning providing index constancy for readers, as was required. If these two methods are combined in a special way that spares I/O operations along with the write-ahead logging examined in 3.2.3 Access granularity, it should be possible to pronounce the solution to be correct with respect to all the major project requirements.

4 Implementation of the solution

For implementation of concurrency control in the Egothor a hybrid transaction processing algorithm has been chosen. The solution will use basic locking for read and write operations exclusion, and multiversioning to guarantee index constancy for readers. Because of the access granularity problem discussed in 3.2.3 *Access granularity* a write-ahead logging will be used for modifications. The following chapter describes the main characteristics of the chosen solution, some implementation issues that had to be solved and it serves as a final result documentation, that can be compared with the initial state described in the chapter 2 *Documentation of Egothor – index creation part* to evaluate the correctness of the chosen solution.

4.1 Egothor issues

In the Egothor there are some issues and requirements that need to be solved before it gets to the more detailed concurrency control design.

4.1.1 Index constancy

Firstly it is the many times mentioned index constancy for readers. Its description may be found in 3.2.2 *Model situations and requirements*. It is time to picture the exact process how it should be arranged. When a reader opens an index, he should demand the index to remain constant for him. Because of the limited hard drive space it would be convenient if some authority told the reader if it is possible to keep the index constant or not. So there would come handy a time period parameter during which the reader would like to have the index constant. After this period of time expires he should reload the index, releasing the old image of index for deletion and loading a new state of the whole index. This leads to a design that upon reader's opening operation of the index the reader specifies a time period during which the index should remain constant. Some authority either confirms this request or not. That way the system gets a possibility to manage the disk space usage and to avoid a denial of service attack, that would be aimed at disk space availability.

To make a snapshot of the whole index is really unnecessary, though. From the nature of system's index management – merging, every append of a new barrel may lead to cascading merges creating new greater barrels and deleting the old ones. So that should be enough just not to delete the merged out barrels and delete them later. When the reader opens the index, it

specifies the index constancy period, then it can load up the handles of all barrels (as described in *2.3.7 Loading previous Tanker state*) and it expects that nobody can change it for him. If it is assured that the merged out barrels will not be deleted, he would get the index constancy as required.

Another operation that alters the index is removal of documents. As described in the chapter *2.3.11 Removing of a document*, there is a bitmap of all deleted documents of the barrel. This bitmap is loaded when the barrel is access for the first time and then it is kept in memory. From this point on, if any other thread changes this bitmap in the barrel directory, it will not affect the reader's already loaded bitmap. So index constancy in relation to removal of documents is assured already as it is.

This index constancy solution may lead to a thought that using a “snapshot” of the index for that purpose every reader could then access the index at his will, not taking into account any conflicts, because there would be none. Unfortunately that is not true, because there are some other parts of the index that are common in the scope of the whole index and thus there must be an access management to them as well.

4.1.2 Documents' revisions

As a document may be stored for the first time, its newer revision may be stored later, when it occurs on the Web and is downloaded and stored in the Egothor - a new version of the same document. Every document has a global Id (explained in *2.3.5 GlobalPositions – detailsGlobalPositions – details*). From now on a unique global Id may appear in many revisions, thus, a document is unambiguously identified by its global Id and the revision.

4.1.3 Data repository revisions

For the search queries as described in *2.3.14 Query* everything should be correct while searching for hits inside the reader's “snapshot”. The process goes over all loaded barrels, which will not be deleted as assured, so the consistency of consecutive calls of query and its results is assured. When it gets to the point of reading additional information from data repository to complete the result (*2.3.6 DataRepository - details*), there it hits a problem.

When a committer modifies the index by appending new barrels with newer versions of the same documents with the same global Ids, it just puts newer version of the record into the metadata repository database, making the new record to be the actual one. As this data

repository resides in the root directory of the index, it is accessible by every thread. When query process starts to read the actual record to fill the result, it may find wrong versions of the entries, versions that do not match the older versions of documents found in its snapshot.

A solution would be to make the data repository a part of the reader's snapshot, but that would not be efficient, because this database may have very big size as it contains every document's every revision's data. With index constancy designed as above there is no additional I/O operation, there is only postponed deletion. If document data repository were added into the snapshot, it could raise the number of I/O operations greatly.

There is a better solution that was implemented – to alter the barrels themselves to be capable of storing exact numbers of revisions of the data repository entries of inner documents. Afterwards when hits are found within the reader's snapshot, the numbers of revisions are taken and the appropriate entry is read from the repository. While performing that, a read lock is required to exclude modifiers accessing the repository as well, which is not a problem.

To conclude, there are two kinds of revision number. The first one is a revision of a document which comes as an input. The second one is a revision of a data repository record, which is created inside the index location by appending the documents into the data repository. So a document with Id 1 may be in its revision 1000 and it can be stored in the index with data repository revision 5, as the index contains only 5 entries of revisions. The data repository revision is a key to other information that was stored along with inverted lists.

4.2 Main design of concurrency control

Finally it gets to the point of the final design. The following chapter will cover it, solving some local issues of implementation, but without many technical details to keep it clear.

4.2.1 Lock server

One of the requirements is that Egothor may run in multiple Java virtual machines on multiple computers. Every thread in those machines must have secure access to the index, avoiding another type of denial of service attack – claiming a lock for infinitely long time. So there must exist some central authority that should manage the lock distribution. As was already enounced there is a need for central unit for index constancy permissions, too. So these two decision can be made at one place, or even more at one time, because index constancy is demanded while reading an index and access for reading an index has a condition to have a

read lock. So it is evident that joining read lock requests and index constancy index together is a good idea. Illustration of the process is in the *Figure 3: Lock server method*.

An implementation issue is to choose a way threads and server will communicate. For the simplicity and speed the UDP protocol has been chosen. Only lock requests and results will be transferred forth and back, so single UDP packets will be just right [7].

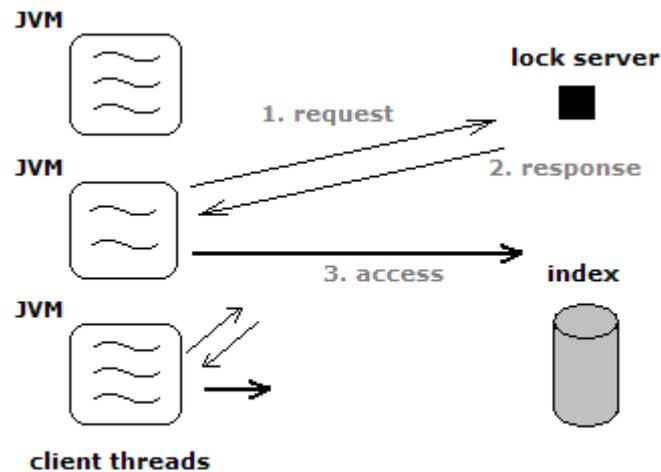


Figure 3: Lock server method

4.2.2 Local solution

One of the major requirements of the project is to implement a concurrency control even without any central unit, even at the cost of security. So previously mentioned global lock server is forbidden in the alternative implementation. As the distributed threads on several machines must be synchronized, they must have something in common to make the synchronization work. And the very last thing they have in common is the index itself. So the synchronization must be done via the disk volume, also.

This task leaves us with **lock files** and some form of a local server for every Java virtual machine, that takes care of lock requests and handles them in relation with the index state, with the presence of other locks from other virtual machines. Illustration of such a solution is in the *Figure 4: Lock files method*.

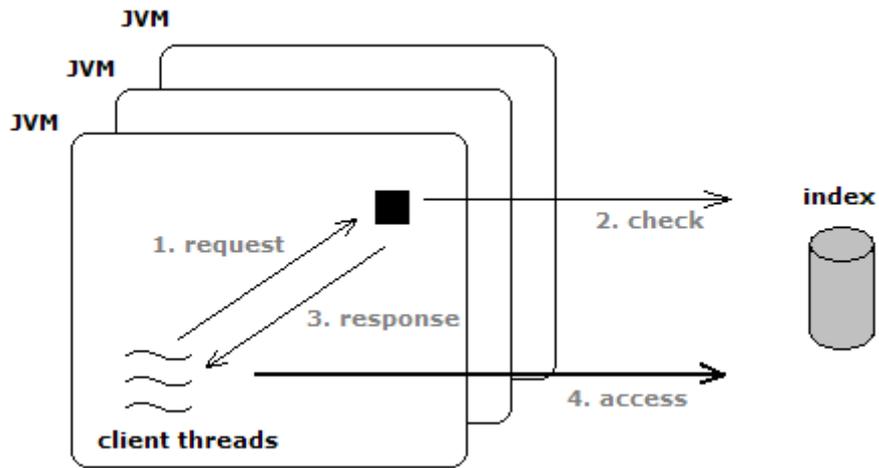


Figure 4: Lock files method

4.2.3 Lock granularity

Lock granularity has already been discussed (chapter evaluating scheduler based on locking 3.3.1.5 Evaluation). It has been set according to access granularity (3.2.3 Access granularity). So it is set that thread's would lock the whole index for their operations.

4.2.4 Lock types

According to Egothor thread types explained in the chapter 3.2.1 Project terminology, there are three basic types of action. The following table sums up the lock types and their usage.

Lock type	Usage
Read lock	Reading index (loading, search queries)
Write lock	Committing previously logged modifications
Recovery lock	For recovery purposes (discussed later in 4.2.9 Recovery)
None	Modifier (write-ahead logging)

Table 7: Lock types

4.2.5 Lock validity

As one thread acquires a lock, all kinds of failures may happen. The thread or the respective Java virtual machine may crash, leaving the lock unreleased. This dangerous situation may end up with locked, inaccessible index for everybody in case of exclusive lock crash, or just inaccessible for writer in case of shared lock crash. Either situation is unwilling and

forbidden.

The solution is to periodically **refresh** each lock. Once a thread is granted a lock, it is its duty to refresh it after small periods of time. If the refreshing stops, the lock expires and it is defined to be illegal. Illegally locked index locations must be recovered from that state, because the index may be in inconsistent state, as the crashed thread could be a committer. Recovery will be discussed later in the chapter *4.2.9 Recovery*.

In case of a lock server, the running server may record a time field for every granted lock and then upon refresh request just set the accurate values in the right fields.

In case of file locks it is efficient to implement the time validity field only as the lock file's modification time. That way only a “system touch command” of the appropriate file would be enough to refresh the lock, instead of writing the time value into that file, which would mean much more I/O operations to perform.

4.2.6 Deadlocks

As already stated in *3.2.4 Deadlocks in Egothor*, deadlocks will not occur, one thread will always try to lock only one location of index.

4.2.7 Starvation

Starvation is a classical problem of sharing resources. In this case starvation may occur whenever a lock request is sent and the index is locked with a conflicting lock. In this case the read lock is wanted and the write lock is present or write lock is wanted and any lock is present. Because the threads are distributed and they may send request to the server at any time, they may outrun any thread in the order of the requests. Thus, a negative response to a write lock request because of an existing conflicting lock may be produced by the server over and over again as other conflicting requests outrun this one. This situation is unwilling and forbidden.

A way how to solve it is to implement **reservations**. A reservation is created every time a negative response is made for a lock request because of some existing conflicting lock, and it is returned as a result. The client thread then may attach this reservation to every other consecutive lock request. A queue of these reservations is kept for every locked location and the order of conflicting reservations is assured to remain the same. After the conflicting lock is released, the first exclusive reservation on the list or the first set of shared reservations may

achieve the lock (a single write lock request or set of read lock requests).

As mentioned in *4.2.5 Lock validity* chapter, any thread may lose its interest in the lock (by crashing or by any other mean), so the reservation has a time validity, also, and it must be refreshed or it expires. Expired reservation is invalid and is deleted while proceeding.

As a consequence of reservations, there is no more outrunning in the order of requests, thus, no more starvation. Every request has assured that if its reservation is being periodically refreshed, it will never be starved out.

4.2.8 Local solution functionality issues

Two types of denial of service have already been discussed, first one against disk space availability, second one against lock availability. This all can be solved in the lock server, which will take care of recording all the request and making decision according to their actual state and actual index needs (space availability).

On the other hand the local solution with only local servers and lock files cannot apply. It has only local knowledge, local information about the requests, it cannot be aware of requests and their state in other Java virtual machines, so it cannot make any decision regarding and influencing them. That is why it is left to only trying to create lock files in the index and granting all index constancy requests. It has no right and information to do anything else.

Similar situations arises when dealing with starvation. Single local lock file server cannot assure in any way the order in which the request would be guaranteed to proceed. The reason is the same as with the DOS attacks. It could assure the order using reservations in the local scope, but that would mean nothing in relation to the other local lock file servers. Anytime those local lock file servers could outrun any other server's request, breaking all the order rules. Thus, local solution using lock files does not prevent starvation from emerging.

In conclusion local lock files implementation is less secure. It cannot prevent DOS attack against lock availability or disk space availability. It cannot avoid starvation, either.

4.2.9 Recovery

Lock validity is closely connected with recovery. Unrefreshed lock leads to expired lock and expired lock means there has been a trouble. In a normal standard execution process of a client thread there should always be matching release lock request for every acquire lock request. If this does not happen, then some error must have occurred.

Every expired lock is treated like the client thread has crashed. Thus, when an expired lock is

found, a recovery process is started. The crashed thread could have been either reader or writer, leaving the index in inconsistent state. Therefore, setting the index back to its original state is required. There is more detail about the index directory recovery itself in the chapter *4.3.9 Global index recovery*.

It is necessary to mention one more thing about recovery process at this level – when an expired lock is found, there may be several reservations awaiting their processing in the same index location. The index recovery process is implemented in a special thread, which can be launched at any time by user by his hand. When a lock server – the global or the local one – finds an expired lock, it cannot waste its precious time to do the recovery process by itself, which may take very long. Instead of that it launches this recovery utility.

The utility acts like an ordinary client and it demands a lock from the server. As was stated, there may exist reservations, which guarantee an order of processing. So this recovery utility needs to outrun them to get first to the broken index and fix it for the others. For this purpose there has been implemented a third type of lock request – recovery lock request. In practice, recovery lock is the same as write lock. The only difference is in an initial processing when deciding whether to grant a lock to a request or not. The recovery lock is capable of outrunning every other request in the reservation queue.

4.2.10 Lock implementation

For performance reasons there are two general types of locks implemented. The types are distinguished by the way they create requests and send them to the server. The first implementation is “spin lock”, when the client thread creates a request and sends it to the server. If server replies with a negative response, the client sends another request right away again trying its luck. This type of lock is very fast and assures the shortest time to acquire a lock, but it is rather CPU time and network resources intensive.

Second basic implementation is called “sleep lock” and it differs in the frequency it is sending new requests. When client receives negative response from the lock server, its thread goes asleep for a short period of time after which it wakes up, creates a new request and sends it to the server again. This implementation does not use the CPU time or network resources as much as the previous one, but it does not assure the fastest lock processing as it can sleep at the time it would have a chance to acquire a lock.

4.3 New user interface layer

After the base of concurrency control is implemented in Egothor it is time to implement a new layer of user interface that uses this concurrency control. A new tanker. The new tanker must convert all actual tanker's methods into a new multi threaded safe versions. Because the initial design of tankers does not count with concurrency, there will be necessary to make many changes in other places of Egothor than just in the newly implemented classes. Changes, that have full respect to functionality of surrounding parts of Egothor and to I/O operations efficiency.

4.3.1 Secure Tanker

As the user interface is implemented by tanker classes, a new *TankerImplSecure* class has been created. It is a descendant of *TankerImpl* and it implements all the necessary methods in a new way. The UML diagram of this class can be seen in the Figure 10: Tanker class diagram. Every new method has its special demands and a way how it is worked into the existing project. There was not just enough to cover the old methods with get lock and release lock requests. There had to be done other changes in the processes and data storing. The following chapters cover implementation of these methods and explain how the concurrency control has been supported by them.

4.3.2 Tanker initialization

New tanker's constructor is without any parameters. Everything is done via initialization method as described int the UML diagram. It takes all the parameters needed for its underlaying class *TankerImpl* plus some more for the new layer. User can choose between array or Berkeley implementation of the mapper or data repository classes (2.3.5 GlobalPositions – details and 2.3.6 DataRepository - details), as a new Berkeley implementation of the mapper has been added, too.

The very last parameter is a string value, that represents a path to a file that contains configuration of the lock server. In particular it is a host address and a port number where the lock server is listening. If this string parameter is a null value, then the user chooses the local non-central way which uses file locks.

4.3.3 Appending new barrels

As was many times stated, this implementation uses **write-ahead logging** for modifications.

Because of the access granularity all modifiers only gather their changes in a local repository without affecting the global index and then they transfer them into the global index when committing.

It is necessary to avoid needless I/O operations, so this log has a structure of a standalone index, as the global one. When the transaction appends a barrel, this barrel is appended into this local tanker – the log, performing merging algorithm only within the local scope. When the commit time comes, the local tanker is ready for being transferred into the global index just as it is, already in shape of barrels, the same structure the global index is based at. And that is exactly the kind of operation that spares many I/Os – **moving** of the barrels from local into the global index in the file system, just directory renaming.

The whole design is illustrated in the Figure 10: Tanker class diagram, the local tanker is an instance of the old *TankerImpl*.

For every document from the newly appended barrel there is a timestamp recorded, a timestamp of appending. This time mark is then used at commit when conflicting operations are discovered – remove and append at the same document with the same version. This timestamp is then the judge.

Every modifier has its own local tanker, of course. There are no conflicts among different modifier. Every each one of them has a local directory in the main index and that is his private sandbox.

4.3.4 Removal of documents

Removing of documents is logged, too. It is only written to log file along with a timestamp of removal, the same kind of timestamp as described in appending. Because every document may appear in many revisions, the removal call must contain the revision number along with the global Id. The log file is placed in the local tanker directory.

4.3.5 Tanker commit

Commit is the most complex process. Firstly it performs local commit of the local tanker, where it follows almost the same path as described in the commit chapter of the original tanker in 2.3.15 Commit, though, it contains many modifications that are not to be mentioned here for simplicity.

Then it requires a write lock on the whole index to proceed on. After it is acquired, it check if the state of the index has changed while it was appending its modification into the local

tanker. If some other commit has been performed on the global index between the client's index opening and the current commit, it reloads it.

After that it prepares the local tanker to be pulled over to the global index. Precisely it means that it goes over all local tanker's barrels and their documents and it checks, whether the same document with the same revision is already placed in the global index, that way it avoids duplicities. If the document in the local tanker has a newer revision or it is not in the global index yet at all, it is kept in the local tanker and it is removed from the global index. The preparation phase checks other facts to assure consistency of the final product and after that it pulls over every single barrel to the global index.

The pulling over is not that simple. It must be done with respect to the merging management of the whole index. So firstly it finds a free slot for the new moving barrel by possible merging of the actual barrels in the global index. After it is finished with it, it simply moves the local barrel into the global index and updates all handles held by the tanker.

To assure a snapshot constancy for readers, while merging it does not delete the dead merged out barrels, it only logs their numbers and a time of the commit into an extra file. The deletion will be executed later at cleaning phase, which is described in the chapter 4.3.7 Index cleaning.

Removal of documents is also processed taking the remove doc log and performing the deletion. Integrity checks are performed again to assure that for example a document that has been just added with older timestamps is not removed by a remove operation with newer timestamp.

All the time every operation with documents is being logged into a listener and at the end this log is transformed into the final mapper information. This follows the same path as described in 2.3.15 Commit, though, again, there are some modifications, for example mapper now contains a new field – the operation timestamp that serves as a judge while encountering conflicting append and remove operations at the same document with the same global Id.

Finally the commit is finished and the write lock may be released. As a result a list of documents' global Ids and their processing result is returned. That way the user can check the result list and see what documents he tried to append were really appended to the index (or removed) and which ones were not (because of the already existing newer versions).

After every commit an index cleaning thread is started as a standalone utility. There is written more about this utility in the chapter 4.3.7 Index cleaning.

4.3.6 Search queries

Operations that need a read lock on the index and they work with their snapshot must be altered in some other way, also. The caller of the methods must be informed if the snapshot validity (index constancy) has expired. In this case he must reload the index, releasing the old snapshot (the merged out barrels) for deletion. This is the reason why there are implemented new forms of the old methods, a new ones with the “Secure” postfix in their names and with a new exception in their throws declarations signaling the snapshot expiration.

For example query method gets a read lock and checks the index constancy state. If the time expired and the index has changed by some commit process of other transactions, it signals this fact. Otherwise if only the time expired and the index has not changed, it only prolongs the constancy time and continues with the operation.

There are many more methods that access the index for reading than just queries. For simplicity they are not stated here. Only the ones that are representative and suitable for demonstration are described in this work, others are documented in the source code in JavaDoc.

Search queries, thus, go over all barrels in the snapshot, accessing the data repository database with entry revisions read out of the respective barrels and combine the data together. At the end it releases the read lock, of course. Everything is functional and correct that way.

4.3.7 Index cleaning

Because of the index constancy the merged out barrels are not deleted during the merging process. They are kept there for the snapshots consistency. After the snapshot is released, they are really “dead” and should be deleted to release the disk space. This cleaning is done via index cleaning utility, which is launched at the end of every commit or it can be launch by user by his hand.

The utility lock the index for writing first. Then it goes over all snapshot definitions. Every transaction when creating a snapshot specifies the time of the snapshot creation and the expiration time. It writes this information into to the index. The utility then goes over these definitions and compares the timestamps with timestamps of commits written in the dead barrels file, which was created or updated on transactions' commit (described in the chapter *4.3.5 Tanker initialization*). All dead barrels' numbers that were recorded with the timestamp of commit older that the oldest snapshot definition are deleted. These barrels do not figure in any snapshot, so it is secure to delete them completely, finally.

After the deletion the utility updates the dead barrels file, also. Finally, the index does not contain any barrel that any transaction would not use.

4.3.8 Modifiers' active state

Modifier do not request locks. That way it is not possible to recognize crashed modifier by examining lock validity. Therefore, modifiers have something special – a modifier active state file. To clearly illustrate this it behaves like a shared lock, which does not block anyone, not even write lock requests. Like an ordinary lock it must be refreshed over the time to remain active (last modification time of the appropriate file again). This refreshing is a job that does not concerns either one of the lock servers. It is performed by the thread itself, for it is no other transactions' nor server's matter.

4.3.9 Global index recovery

As there are many transaction types operating over the index requesting locks, logging their modifications, reading or committing, they all might just crash and stop whatever they were doing at that exact moment. All these accidents may leave the index in a more or less inconsistent state. Therefore there was implemented an index recovery utility, which takes care of these problems.

Whenever there is an expired lock found in some index location, the recovery utility is launched there. The utility may be launched by user by his hand, also.

This utility locks the index with write lock and examines the dealt damage. It removes all expired locks and expired index constancy definitions files. The reader could not do any damage except for claiming a snapshot. The snapshot will expire and the dead barrels will be removed, so there is no more work necessary to be done here.

A crashed modifier may be distinguished from active modifier in the index by his active state file as already described – the crashed ones have their modification state files expired. If there are found such expired modifiers, their local tanker directory is examined and their work done so far is rolled back.

If the modifier was not already in a commit phase, his local tanker directory may be just deleted along with the expired modification state file. No committing was performed so far, so there is no more work to do.

A more complex case is if the modifier already was in a commit phase. The deletion of the local directory would not just be enough, because there are some modifications done to the

global index, also. Every committer writes its progress into a listener, which can be taken as his log file. So all modifications committed so far are deleted and the index is restored to the previous consistent state with merged barrels in a shape just like it was before.

4.3.10 Implementation issues and other modifications

There had to be solved some implementation issues, like the many times mention efficiency. In every Java virtual machine, there is always one timer thread, that gathers requests from all transaction threads to refresh their locks or state files or whatever they need. The transaction just registers its requirement in the timer and the timer takes care of everything else.

As was already mentioned, every timestamps binded to a file is saved as its last modification time. That way it saves a lot of I/O operations and a hard drive reading heads movement.

The class designed is made in such a way that there is a lock management layer which can be replaced by some different lock management in the future, if desired.

In addition, there were many other modifications implemented in the source code to make it all work. They were not mentioned here in this work, because they could clearly confuse the reader by overloading him with many details.

5 Conclusion

After the description of the whole process of implementing concurrency control into the Egorthor it is time to evaluate the outcome. The evaluation will be done via comparing the result with the major requirements stated at the beginning of this work.

1. Several threads executed in single Java Virtual Machine cannot conflict with each other with fatal consequences during appending of documents or removing them. Every such an operation must leave the index in consistent state after commit and visible for everyone.
2. Several threads executed in one Java Virtual Machine cannot conflict with each other with fatal consequences while one of them is reading the index.
3. Several threads executed in multiple Java Virtual Machines on single computer machine should work as described for single Java Virtual Machine.
4. Several threads executed in multiple Java Virtual Machines on multiple computer machines should work as described for single Java Virtual Machine.

Requirements 1 through 4 are all fulfilled. Locking in combination with multiversioning and write-ahead logging ensure these demands to be fulfilled.

5. Once a thread opens the index and starts reading its content, the index visible to the thread should not change by any other thread's intervention without the reader noticing it or allowing it. In other words once a thread opens an index, it must have a certainty, that the index will remain constant during some period of time, so when some read out data have a relationship with other so far unread data, the relationship will not change without the reader noticing it or even allowing it.

Done by using multiversioning, excluded by other threads.

6. Avoid inefficient copying of files or unnecessary I/O operations.

Using write-ahead logging, there are no necessary rollbacks in the index. Using multiversioning in a version as described, when there is only postponed deletion of the barrels, there are, also, no extra I/O operations.

7. Try to avoid any global central unit of the application, like a server. If it is necessary to have one, make another implementation, also, that can work without it, even it would worsen some aspects of security or functionality. That way working threads are totally independent and do not rely on any other unit. The user may choose his preferable way to use.

Central unit – the lock server – was inevitable. Therefore, there has been implemented a second method, plain local lock files solution.

8. Do not inefficiently use disk space, delete dead parts of index as soon as possible.

All merged out barrels, that are not a part of any snapshot, are deleted. All crashed transactions are rolled back, also, releasing the disk space. Snapshots are kept as small in size as possible, no duplication of any barrel.

9. Do keep solution compact and transparent.

The implemented solution is designed to fit the exact needs of the Egothor system, there were not used any third party software that would have to be modified to fit. Lock manager layer can be replaced, if desired, it's a standalone layer. There are no necessary threads being launched, everything tried to be gathered at one place (timer thread).

10. Implement a new layer of user interface for managing index using concurrency control.

New *TankerImplSecure* layer has been implemented. User has to only choose between lock server way and file locks way. Every method and every process has been reimplemented to work correctly in the multi threaded environment.

From my point of view, the requirements has been fulfilled and the implementation is functional. There were, also, functional tests implemented, to examine and demonstrate the correct functionality (described in the Appendix).

6 Bibliography

- [1] Leo Galamboš (2006): Egothor documentaion. Department of Software Engineering, Charles University, Prague, Czech republic. <http://www.egothor.org/docs/e2.pdf>
- [2] Philip A. Bernstein, Eric Newcomer (1997): Principles of Transaction Processing. Morgan Kaufmann Publishers, Inc. San Francisco, CA.
- [3] Gerharg Weikum, Gottfried Vossen (2002): Transactional Information Systems, Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers, Inc. San Francisco, CA.
- [4] Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete (1994): Atomic Transactions. Morgan Kaufmann Publishers, Inc. San Mateo, CA.
- [5] Petr Tůma (2006): Transakce. Matematicko-fyzikální Fakulta, Univerzita Karlova. Praha. <http://dsrg.mff.cuni.cz/~ceres/sch/txy/main.php>
- [6] Marek Procházka (2000): Transakce. Matematicko-fyzikální Fakulta, Univerzita Karlova. Praha. <http://dsrg.mff.cuni.cz/%7Eprochazk/lectures/Tp000529.pdf>
- [7] Eric A. Hall (2000): Internet Core Protocols: The Definitive Guide. O'Reilly & Associates, Inc. Sebastopol, CA
- [8] Andrew S. Tanenbaum (2001): Modern Operating Systems. Prentice Hall. Upper Saddle River, NJ
- [9] Leo Galamboš (2004): Dynamization in IR Systems. Department of Software Engineering, Charles University, Prague, Czech republic. <http://www.egothor.org/~galambos/twiki/pub/Egothor/CoreModel/TR-93-dynamization-fp.pdf>

A. Appendix

A.1 Functional tests

Along with the full implementation of the concurrency control into the Egothor system there have also been developed functional tests as a part of the product. These tests demonstrate and examine the implemented functionality. There are couple of minor tests written in Java using JUnit testing framework which will not be covered here. There are also two major tests, that need further description and demonstration. There is a special test only for lock management and there is an overall test testing the whole index management using the new user interface layer.

All the tests require a running lock server before launched, in case server locking is the target of the test. In case of file locking test there is no pre-requisite.

A.1.1 Lock management test

There are two types of this test – FileTest and ServerTest. Both testing lock management, but each using a different method.

ServerTest launches four different threads that try to acquire different locks concurrently communicating with the lock server. They use spinning locks as there is tested the exact time at which they acquired the lock in comparison with the expected time. Because four concurrent threads' execution is clearly a mess to understand only from the source code, an illustration of their execution is pictured in the *Table 9: Lock server test scenario*.

The key of illustration of the test is explained in the *Table 8: Test illustration key*.

Figure	Meaning
+	Request to acquire a lock
-	Request to release a lock
RL	Read lock
WL	Write lock
*	Lock acquired
1, 2, 3	Index locations (directory “names”)

Table 8: Test illustration key

Time/ seconds	Thread 1	Thread 2	Thread 3	Thread 4
0		+RL 1 *	+RL 1 *	+RL 1 *
1				
2	+WL 1	-RL 1		
3			-RL 1	
4	*			-RL 1
5		+RL 1		
6			+RL 1	
7				+RL 1
8	-WL 1	*	*	*
9	+WL 1			
10	*	-RL 1	-RL 1	-RL 1
11		+RL 1		
11.4			+WL 1	
11.8				+RL 1
12	-WL 1	*		
13		-RL 1	*	
14			-WL 1	*
15	+RL 2 *	+RL 2 *	+RL 1 *	-RL 1
16			+WL 1 *	+WL 3 *
17		+WL 2		
18	-RL 2	*		
19			-WL 1	
20	+RL 3		+WL 2	
21	*	-WL 2	*	-WL 3
22	-RL 3		-WL 2	

Table 9: Lock server test scenario

The other test is the FileTest. It tests concurrent lock requests, also, but using lock files. As the local lock files method does not guarantee any procession order because of its locality, the test is much simpler, using only two threads that test only the basic lock management behavior. It is illustrated in the *Table 10: Lock files test scenario*.

Time/seconds	Thread 5	Thread 6
0	+RL 1 *	+RL 1 * +WL 1
1		
2		
3	-RL 1	*
4	+RL 1	
5	*	-WL 1
6	+WL 1 *	
7		+ WL 1
8	- WL 1	*

Table 10: Lock files test scenario

A.1.2 Overall test

Overall test is a major test examining the main functionality of the newly implemented source code, simulating a real time usage of the Egothor system. There are five threads launched, two of which are readers and the other three are modifiers. User can pass as a parameter to them a path to a lock server configuration file, if he chooses the lock server version, or null, in case of lock files version, as was eventually mentioned in the chapter 4.3.2 *Tanker initialization*, where the usage of the new secure tanker was explained. So this test can be either run with file locks or server locks, as a real time running system could. The illustration of the test is pictured in the *Table 12: Overall test scenario*.

Modifiers append particular documents with particular revisions and particular content along with some “padding” (every time 200 of some pre-generated documents with Ids and versions and content of no interest to the test), while readers are trying their search queries, comparing the hits with anticipated results (number of hits, the documents global Ids and revisions).

The readers reload the index sometimes to release their snapshot for deletion and to load up newly appended documents and try to process them as well. The tankers are set to have `mergeFactor` of size 2, so merging happens very often, thus, snapshots consistencies are tested correctly. Thread modifier number 2 simulates crashing after his second task, so index recovery is tested, also. There is, also, tested, that documents with revisions that are already in the index cannot be added into it again.

Reader number 6 only periodically performs query on his old snapshot. The key of the

illustration of test is in the *Table 11: Overall test key*.

Figure	Meaning
A	Append
R	Remove
C	Commit
Q	Search query
*	Should processed OK
-	Should NOT process OK
Initialization	Loading the index for the first time
Reload	Reloading of the index (release snapshot, load a new state of index)
00, 10, 21	Document [document's global ID][revision]

Table 11: Overall test key

Task	Modifier 1	Modifier 2	Modifier 3	Reader 5	Reader 6
1	A 00, 10 C 0, 1 *				
2				Initialization Q 00 10 *	
3					Initialization
4		A 00, 10 C 0, 1 -			Q 00, 10 *
5	A 01, 11 C 0, 1 *				Q 00, 10 *
6				Q 00, 10 * Reload Q 01, 11 *	Q 00, 10 *
7		A 20 crash			Q 00, 10 *
8				Recovery launch	Q 00, 10 *
9				Q 01, 11 * 20 -	Q 00, 10 *
10			A 02, 20 R 11 C 0, 1, 2 *		Q 00, 10 *
11				Reload Q 02, 20 * 11 -	Q 00, 10 *
12			A 21 C 2 *		Q 00, 10 *
13				Reload Q 02, 21 * 11 - [5-times more]	Q 00, 10 * [x-times more]

Table 12: Overall test scenario

A.2 UML diagrams

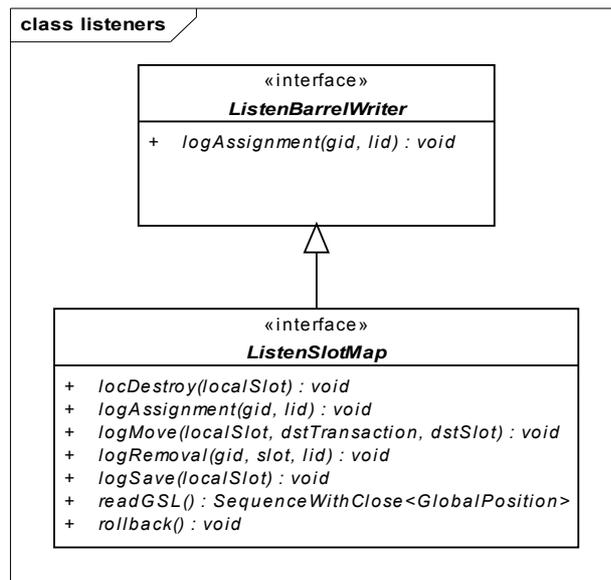


Figure 5: Listeners

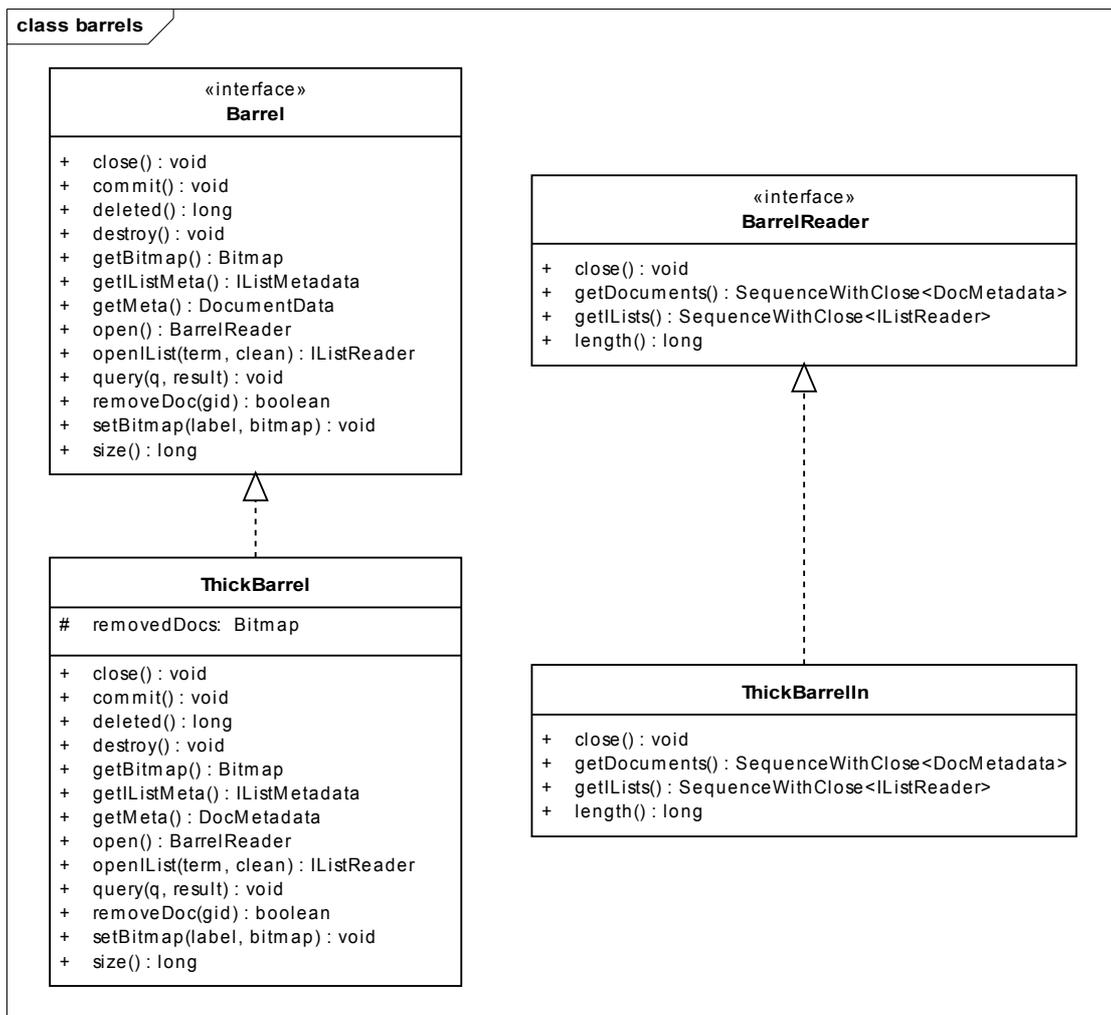


Figure 6: ThickBarrel and ThickBarrelIn

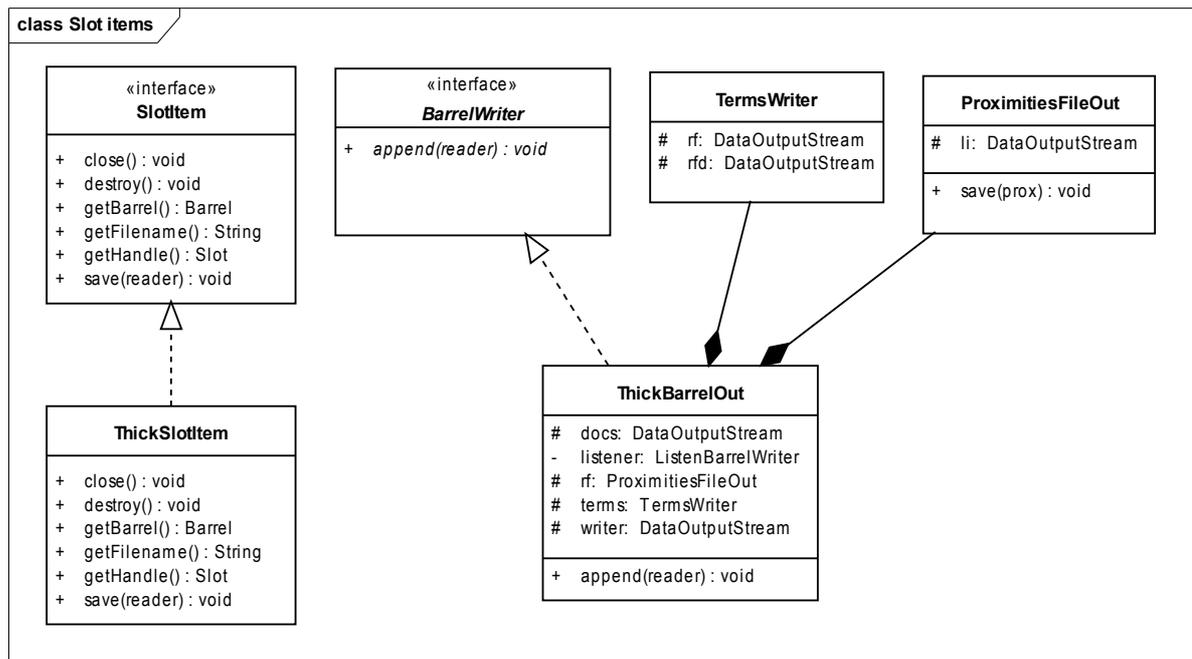


Figure 7: Slot Items and Barrels

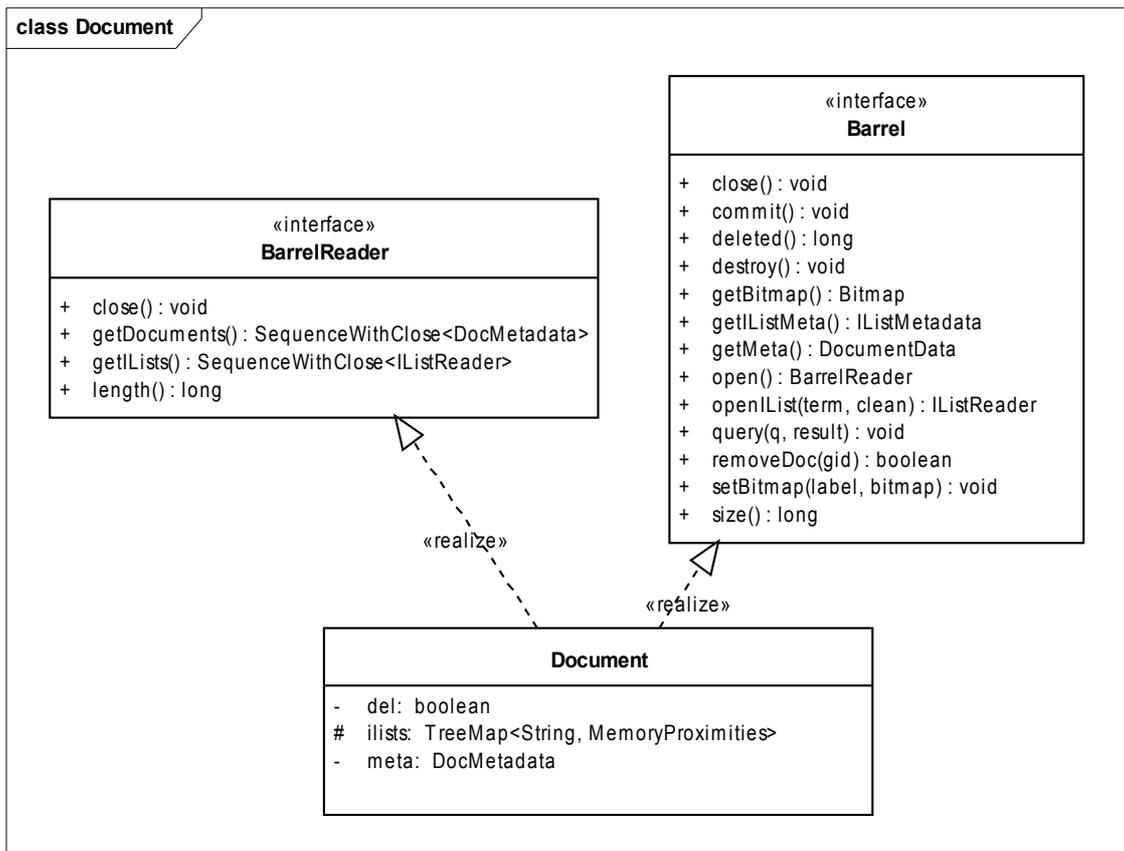


Figure 8: Document class diagram - detail

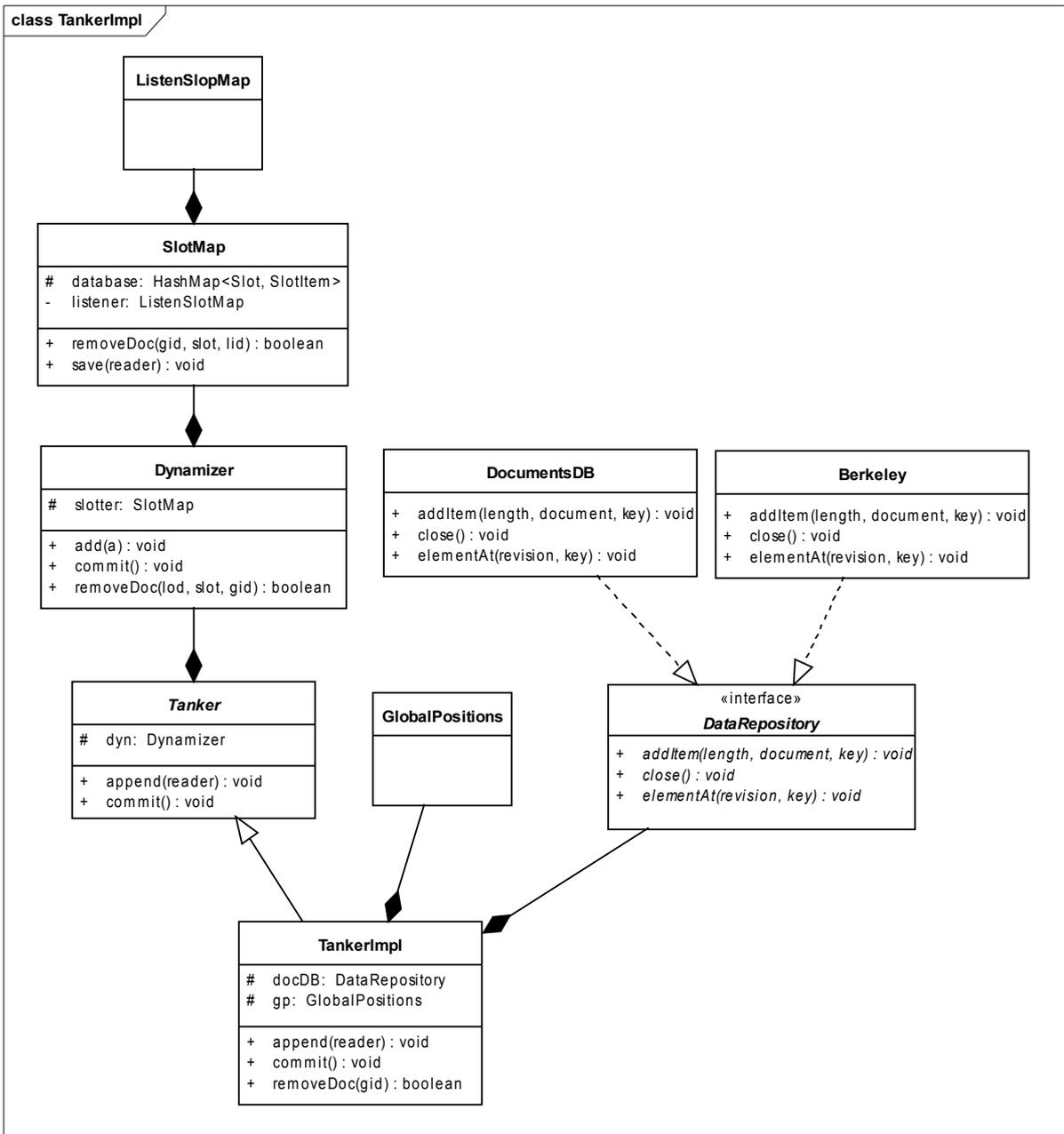


Figure 9: Tanker class diagram - detail

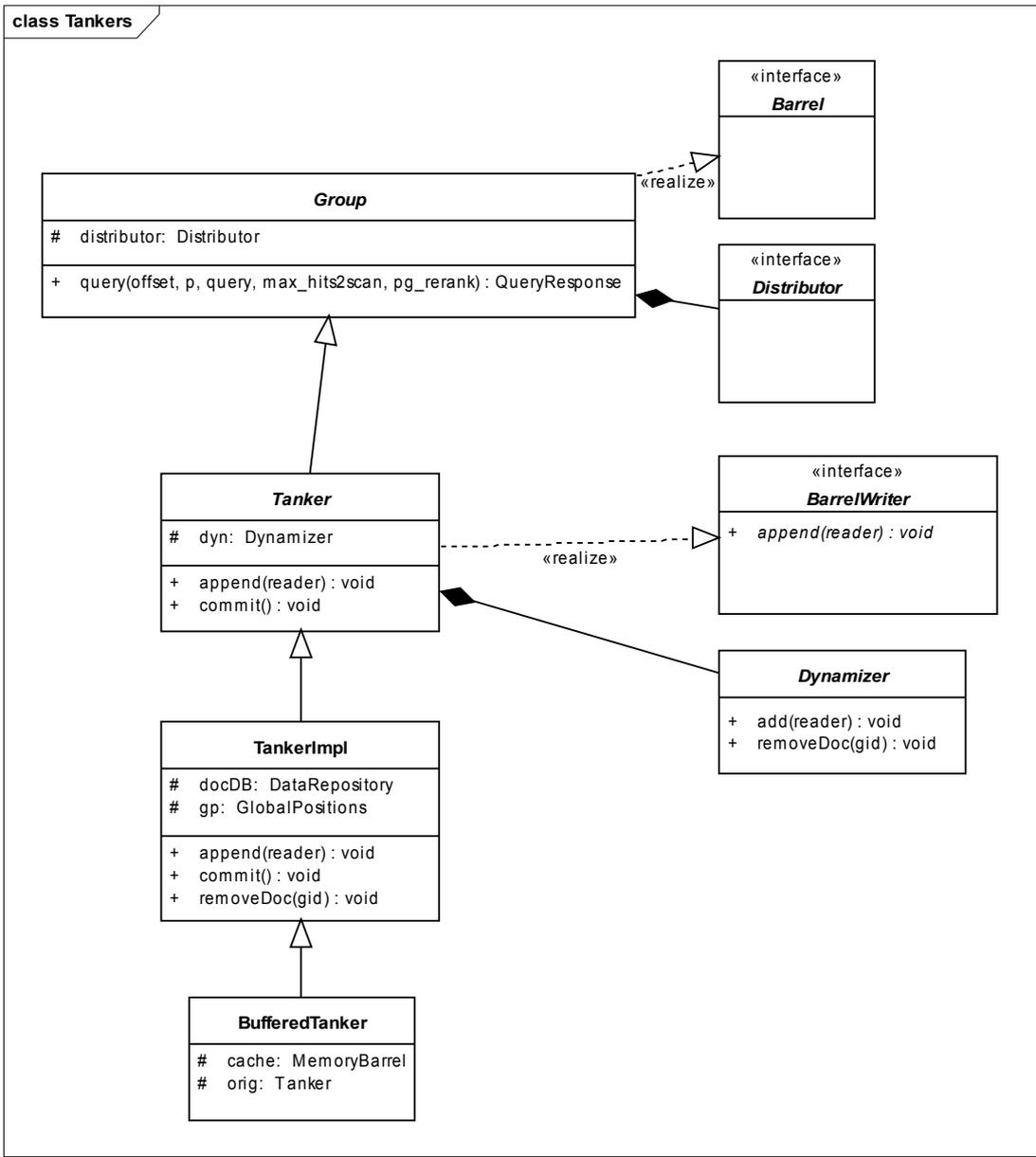


Figure 10: Tanker class diagram

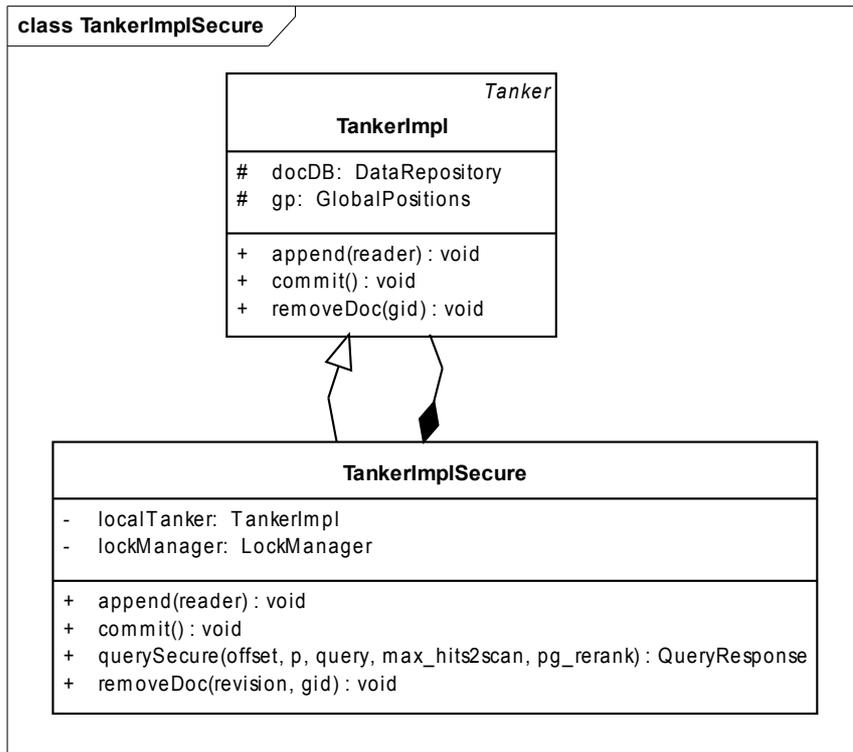


Figure 11: TankerImplSecure

A.3 CD-ROM

A.3.1 Content of the CD-ROM

CD-ROM contains this master thesis in PDF format, complete source code of the Egothor project with implemented functionality described in this thesis and source code of the functional tests.

Along with the project source code there is an example of a configuration file for lock server (“lockServerConfig”).

A short programmer's documentation has been added, too, to allow quick start for new programmers – Programmer's Quick Start (ProgrammersQuickStart.pdf).

A.3.2 How to run the application

To run the application in other way than just already written tests check the Egothor documentation [1], where the whole process is explained in detail.

To run the tests run the lock server first. Afterwards run the tests either as JUnit test or as ordinary Java application (depends on each test). Tests described in the *Appendix A.1* are supposed to be run as ordinary applications. The rest of the tests are supposed to be executed in the JUnit framework. There is some more information about the tests in the Programmer's Quick Start document.

A.4 Programmer's Quick Start

- 1) Content of the source code
 - 2) Short description of the implementation
 - 3) Short description of the tests
-

- 1) Content of the source code

Source code contains:

- Egothor with newly implemented transactions + new version of a Tanker, which uses transactions + a lot of other changes. Placed in the *src* directory. It contains the whole project as it was downloaded from the repository + new modifications.
- Functional tests of the new implementation. Placed in the *test* directory. It contains only new tests.
- File *lockServerConfig* which is a configuration of a network communication among threads and a lock server. Placed in the *src* directory and *test*, also, as it is needed there, too, to run the tests. It has a structure of a `java.util.Properties` export file.

- 2) Short description of the implementation

Transactions are implemented in two versions – with central network unit (LockServer) and without any central unit (using lock files). The user chooses which version he wants by specifying a parameter for the tanker's initialization method – *TankerImplSecure.initialize(..., String lockServerConfigFilename)*. If this parameter is *null*, then the lock files version is chosen. Otherwise this string is interpreted as a path to a file (*lockServerConfig*) which contains the IP address (if the address is “localhost”, the local loop is used) and the port number of the server.

- i) Locking

- 3 types of locks – READ, WRITE, RECOVERY
- RECOVERY lock is basically WRITE lock, only during decision making about granting a lock the RECOVERY request is given precedence. After the process it

becomes a normal WRITE LOCK. RECOVERY lock is used only in *IndexRecovery* class which is being launched as a new thread every time there is an expired lock found (some thread has crashed, recovery of the index is needed). Or it can be launched by hand. A location of the index to be recovered is given to it as a parameter, so it is important to use absolute paths, because different JVM = different relative paths.

- Threads make requests to get a lock. In case of READ they can additionally specify a time period during which the index will remain constant for them (the index constancy).

- After the lock is granted a *TimerTask* is created and it is placed into the *Timer* (each JVM has one *Timer*). Task is then responsible for periodical refreshing of the lock, until the thread that requested it dies or until it is released.

- clients may use Spin locks, Sleep locks and ordinary single lock requests. Spin lock is a consecutive sending of requests (*getSpin[Write|Read]Lock*), Sleep lock always goes asleep for a short period of time and then it tries the request again (*getSleep[Write|Read]Lock*). Single ordinary request is only one request (*get[Write|Read]Lock*). Everything is in the *LockManager* class. Its descendants *LockServerManger* and *LockFileManager* then implement functionality in relation to their type.

ii) LockServer

- communication via UDP

- to requests which cannot be granted with a lock is assigned a reservation. It is implemented to avoid WRITE lock starvation (READ requests could outrun it all the time, they do not block each other).

Such a reservation must be periodically refreshed, or it expires (thread crashed, lost his interest). During every refreshing of the reservation (more precisely upon every lock request with specified reservation number) there is the lock state checked first, if it can be granted or not. If only READ requests are among reservations, their order does not matter, they do not block each other, lock is granted. If there is a WRITE lock reservation, no READ request can outrun it, except for those that have their reservation made before the WRITE request. Every reservation may be outrun by RECOVERY request.

- server records for every location index the index constancy requests, also. It is not

used now, though, later it may be used for decision making about whether to grant the constancy request or not (in dependence on disk space availability).

- server may be launched with parameters of the network communication. If not specified, it tries to look for it in a configuration file with default filename as specified in the *Constants* class.

iii) LockFile

- in every JVM there is one lock file manager object (local server) which acts as a server. Because it has no global information about all the clients using the index, it cannot make decisions about index constancy requests. At the same time it cannot assure any order of the requests' execution, so WRITE lock starvation may occur.

- it tries to create a lock file. If it succeeds, true returned. If not, it returns false as a result. If some other local server has done the same thing – created a lock file and they block each other now – the lock is deleted and false is returned, also.

iv) Tanker usage

- new *TankerImplSecure* has been created, constructor is without any parameter, initialization done via *initialize(...)* method. It implements *appendSecure*, *commit*, *commitWithResults* (returns the result of the whole commit – of all appends and removes), *removeDoc*, *openSecure*, *querySecure* and many others. Everything is implemented using locks, so it is multi threaded safe. Those methods with *Secure* postfix throw a new exception which means that the index has changed, the time constancy for the given index has expired, it is necessary to reload the index (it is possible use *TankerImplSecure.reloadIndex(int)* method). The exception is thrown only if the constancy expired AND the index has changed. If the index has not changed, only the constancy has expired, the execution continues with newly declared constancy time period – the user is not bothered by useless by-hand reloading.

- Inherited methods without the *Secure* postfix are declared to be deprecated. They should not be called from the *TankerImplSecure* class.

- when choosing a lock server version it is necessary to have a configuration file and its path put as a parameter in the initialization of the tanker.

1. Append – performed in a local tanker in the index. In commit phase its new barrels are only moved (renamed) to the global index.

2. RemoveDoc – it is being gradually logged in the local tanker.
 - For append and remove there is no lock needed, everything is only local. So just active modifier state is being refreshed of such a thread (last modification time of a appropriate file) to make it possible to distinguish active living modifier and a crashed one (recovery is needed to clean the index, roll it back).
3. Query – needs RL
4. Commit – needs WL. By using comparison of uids and revisions of documents from local tanker and the global ones it can decide whether to import the local tanker's document or not. Barrels are only moved, as a whole unit. Another decisions about appending the new document or not can be made by comparing its revision and the operation's (append, remove) timestamp (in case of conflict on the same document with the same revisions).

3) Short description of the tests

I could find any multi threaded JUnit plugin, so some tests must be launched as an ordinary Java application.

Run the lock server first as a standalone application. In case of testing lock files there is no need for that.

- The basic usage is tested in *org.egothor.test.Indexer*.
- Another test is *org.egothor.dir.TankerAppendTankerTest*, where some more things are tested (ie. *tanker.append(tanker.open())*).
- Test of only lock management alone is in *org.egothor.lock.ServerTest* a *FileTest*. Classes *ThreadX* belong to these tests.
 - ServerTest* – the test scenario is described and illustrated in the master thesis text in the Appendix.
 - FileTest* – the test scenario is described and illustrated in the master thesis text in the Appendix.
- Test of it all is in *org.egothor.dir.OverallTest*. the test scenario is described and illustrated in the master thesis text in the Appendix. At the end it should write to the output *[thread id, thread name: processedOK = true]* as a sign that everything went OK.
- There are some more modifications in *samples.CompleteTest*, (test optimize()).