



Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών  
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής  
και Υπολογιστών

## Thesis subject

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΟΝΟΜΑ ΦΟΙΤΗΤΗ

Επιβλέπων : Υπεύθυνος Διπλωματικής  
Τίτλος Υπευθύνου

Αθήνα, Σεπτέμβριος 9999





Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών  
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής  
και Υπολογιστών

## Thesis subject

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΟΝΟΜΑ ΦΟΙΤΗΤΗ

Επιβλέπων : Υπεύθυνος Διπλωματικής  
Τίτλος Υπευθύνου

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 9η Σεπτεμβρίου 9999.

.....  
Πρώτο μέλος επιτροπής  
Τίτλος μέλους

.....  
Δεύτερο μέλος επιτροπής  
Τίτλος μέλους

.....  
Τρίτο μέλος επιτροπής  
Τίτλος μέλους

Αθήνα, Σεπτέμβριος 9999

.....  
**Όνομα Φοιτητή**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Όνομα Φοιτητή, 9999.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

## **Περίληψη**

Περίληψη της διπλωματικής.

## **Λέξεις κλειδιά**

Λέξη-κλειδί 1, λέξη-κλειδί 2, λέξη-κλειδί 3



## **Abstract**

Abstract of diploma thesis.

## **Key words**

Key-word 1, Key-word 2, Key-word 3





## Ευχαριστίες

Ευχαριστίες.

Όνομα Φοιτητή,  
Αθήνα, 9η Σεπτεμβρίου 9999

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-\*-\* , Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Σεπτέμβριος 9999.

URL: <http://www.softlab.ntua.gr/techrep/>  
FTP: <ftp://ftp.softlab.ntua.gr/pub/techrep/>



# Contents

<b>Περίληψη</b>	5
<b>Abstract</b>	7
<b>Ευχαριστίες</b>	9
<b>Contents</b>	11
<b>List of Figures</b>	13
<b>1. Introduction</b>	15
1.1 Introduction/Motivation	15
1.2 Thesis structure	15
<b>2. Chapter 2</b>	17
2.1 Section 1	17
2.1.1 Subsection 1	17
2.2 Section 2	17
2.2.1 Subsection 1	17
2.2.2 Subsection 2	17
<b>3. Chapter 3</b>	19
3.1 Section 1	19
3.2 Section 2	19
3.2.1 Sub-section 3	19
<b>4. Chapter 4</b>	21
4.1 Section 1	21
<b>5. Design of cached</b>	23
5.1 General	23
5.2 The xcache xtype	24
5.2.1 Indexing	24
5.2.2 Eviction	24
5.2.3 Hooks	24
5.2.4 Refcounts and locks	25
5.2.5 Re-insertion	25
5.3 The xworkq xtype	25
5.4 The xwaitq xtype	26
5.5 Cached internals	26
5.5.1 Object states	26
5.5.2 Per-object peer requests	26
5.5.3 Write policy	27

<b>Bibliography</b> . . . . .	29
-------------------------------	----

**List of Figures**

3.1 This is an image . . . . . 19



## Chapter 1

### Introduction

#### 1.1 Introduction/Motivation

Bla-bla...

#### 1.2 Thesis structure

**Chapter 2:** We define what "cloud" means and mention some of the most notable examples. Then, we give a brief overview of the synnefo implementation, its key characteristics and why it can have a place in the current cloud world.

**Chapter 3:** We present the architecture of Archipelago and provide the necessary theoretical background (mmap, IPC) the reader needs to understand its basic concepts. Then, we thoroughly explain how Archipelago handles I/O requests. Finally, we mention what are the current storage mechanisms for Archipelago and evaluate their performance.

**Chapter 4:** We explain why tiering is important and what is the state of tiered storage at the moment (bcache, flashcache, memcached, ramcloud, couchbase). Then, we provide the related theoretical background for cached (hash-tables, LRUs). Finally, we defend why we chose to roll out our own implementation.

**Chapter 5:** We explain the design of cached, the building blocks that is consisted of (xcache, xworkq, xwaitq). Then, we provide extensive benchmark results and compare them to the ones of Chapter 3.

**Chapter ??:** TODO

**Chapter ??:** We draw some concluding remarks and propose some future work.





## Chapter 2

## Chapter 2

### 2.1 Section 1

This section has an important citation[[aeal99](#)]

#### 2.1.1 Subsection 1

This subsection has code in Haskell:

```
1 foo [] = []  
2 foo h:t = 9: foo t
```

**Listing 2.1:** Sample code

It also has a list:

**Item 1** First item

**Item 2** Second item and a footnote<sup>1</sup>.

**Item 3** Third item and text in *italics*.

And an enumerated list:

1. First item.
2. Second item and text in **bold**

### 2.2 Section 2

#### 2.2.1 Subsection 1

This subsection has a link to the block of code [2.1](#) in Section 1.

#### 2.2.2 Subsection 2

This subsection has a FIXME comment, visible only to the author.

---

<sup>1</sup>Footnote description.



## Chapter 3

## Chapter 3

### 3.1 Section 1

This how we add a url: <http://www.example.org>

### 3.2 Section 2

And this is how we point to Figure 3.1.



**Figure 3.1:** This is an image

#### 3.2.1 Sub-section 3

Another way to create a list:

- Item 1

- Item 2
- Item 3
- Item 4

## Chapter 4

## Chapter 4

### 4.1 Section 1

We can also use a special fonts to differentiate between text and *math()*.



## Chapter 5

### Design of cached

In the previous chapters, we have addressed the need for tiering in terms of scalability as well as performance.

We have also evaluated current caching solutions and described why they couldn't be used as a cache tier in Archipelago.

With the results of chapter 4 in mind, we can provide some more strict requirements that our solution must have:

1. Requirement 1
2. Requirement 2
3. Requirement 3
4. Requirement 4

The following two chapters are the main bulk of this thesis and they present our own implementation that aims to fill the above requirements.

More specifically, this chapter provides an in-depth description of the design of cached. Section 5.1 provides a general overview of cached. Sections 5.2 - 5.4 present the building blocks of cached and their design. Section 5.5 presents the interaction of cached and its building blocks. Finally, in Section 5.6 we illustrate the flow of requests for cached.

#### 5.1 General

In order to provide a caching tier for Archipelago that would fulfill our requirements, we had to create our own implementation. Its name is simply cached (**cache daemon**) and is another XSEG peer with similar structure to those seen in chapter 4.

For the creation of this peer, we have created some xtypes that act as the building blocks for this peer. These xtypes are the following:

- **xcache** (for cache support)
- **xwork** (for job support)
- **xworkq** (for atomicity in execution of jobs)
- **xwaitq** (for conditional execution of jobs)

and their design will be discussed in-depth in the following sections.

## 5.2 The xcache xtype

xcache is the main component of cached. It handles the indexing, reference counting, coherency and eviction jobs all by itself. A basic overview of xcache can be seen below

More specifically, xcache consists of two hash tables. One hash table is responsible for indexing objects (or more generally speaking "cache entries") that are active in cache. The other hash table is responsible for indexing evicted cache entries that have pending jobs. Again, more generally speaking, evicted cache entries whose refcount has not dropped to zero yet.

### 5.2.1 Indexing

In order to index the cached objects, xcache relies on another xtype, xhash, which is a hash table. What's more, it's actually the C implementation of the dictionary used in Python.

xhash is one of the main reasons our implementation is guaranteed to be  $O(1)$ .

Finally, the xhash xtype gives provides us with the basic hash table functions, namely:

- Insertion
- Look-up
- Deletion

### 5.2.2 Eviction

Cache entry eviction is done almost transparently from the user. xcache has two LRU algorithms, a linear LRU array or a binary heap (more at xbinheap) that can be chosen at runtime and are responsible for deciding which is the least recently used entry. Users do not explicitly interact with the LRU array. Eviction occurs automagically during the insertion of a new cache entry and the user is informed via a specific hook that is triggered upon eviction.

### 5.2.3 Hooks

The hooks that xcache provides to users are:

- on\_init: called on cache entry initialization.
- on\_put: called when the last reference to the cache entry is put
- on\_evict: called when a cache entry is evicted.
- on\_node\_init: called on initial node preparation.
- post\_evict: called after an eviction has occurred, with cache lock held.
- on\_free: called when a cache entry is freed.



- `on_finalize`: called to hint the user that the cache entry's ref has dropped to zero.
- `on_reinsert`: called when a cache entry has been in cache

#### 5.2.4 Refcounts and locks

The refcount model in xcache should be familiar to most people:

- When an entry is inserted in cache, the cache holds a reference for it (`ref = 1`).
- Whenever a new lookup for this cache entry succeeds, the reference is increased by 1 (`ref++`)
- When the request that has issued the lookup has finished with an entry, the reference is decreased by 1. (`ref--`)
- When a cache entry is evicted by cache, the its ref is decreased by 1. (`ref--`)

Some common refcount cases are:

- active entry with pending jobs (`ref > 1`)
- active entry with no pending jobs (`ref = 1`)
- evicted entry with pending jobs (`ref > 0`)
- evicted entry with no pending jobs (`ref = 0`)

and, as always, the entry is freed only when its `ref = 0`.

Finally, xcache uses one lock for each hash table but when a cache entry shifts from one hash table to the other, both locks are acquired.

#### 5.2.5 Re-insertion

In xcache, there is a concept called "re-insertion". In order for an entry to be re-inserted to the primary hash table (which will be called "entries" from now on) it must first reside in the hash table that indexes the evicted cache entries (which will be called "rm\_entries" from now on). As mentioned above, an entry that is in `rm_entries` has probably pending jobs that delay its removal.

So, what happens if a lookup arrives for that entry while on this stage? In this case, we re-insert it to `entries` and increase its refcount by 2, since there is one reference by the hash table and one reference by the one who requested the lookup.

### 5.3 The `xworkq` xtype

Every object has a `workq`. Whenever a new request is accepted/received for an object, it is enqueued in the `workq` and we are sure that only one thread at a time can have access to the objects data and metadata.

For more information, see the `xworkq`.

## 5.4 The xwaitq xtype

When a thread tries to insert an object in cache but fails, due to the fact that cache is full, the request is enqueued in the xcache waitq, which is signaled every time an object is freed.

For more information, see the xwaitq.

## 5.5 Cached internals

### 5.5.1 Object states

Every object has a state, which is set atomically by threads. The state list is the following:

- **READY:** the object is ready to be used
- **FLUSHING:** the object is flushing its dirty buckets
- **DELETING:** there is a delete request that has been sent to the blocker for this object
- **INVALIDATED:** the object has been deleted
- **FAILED:** something went very wrong with this object

Also, object buckets have their own states too:

- **INVALID:** the same as empty
- **LOADING:** there is a pending read to blocker for this bucket
- **VALID:** the bucket is clean and can be read
- **DIRTY:** the bucket can be read but its contents have not been written to the underlying storage
- **WRITING:** there is a pending write to blocker for this bucket

Finally, for every object there are bucket state counters, which are increased/decreased when a bucket state is changed. These counters give us an  $O(1)$  glimpse to the bucket states of an object.

### 5.5.2 Per-object peer requests

Reads and writes to objects are practically read/write request from other peers, for which a peer request has been allocated. There are cases though when an object has to allocate its own peer request e.g. due to a flushing of its dirty buckets. Since this must be fast, there are pre-allocated requests hard-coded in the struct of each object which can be used in such cases.

### **5.5.3 Write policy**

The user must define beforehand what is the write policy of cache. There are two options: writethrough and writeback. On a side note, as far as reads and cache misses are concerned, cached operates under a write-allocate policy.



## Bibliography

- [aeal99] Some author et al., “Name of citation”, in *Proceedings of the 99th ACM Symposium on Something (POPL’99)*, pp. 999–999, Nine, 9999.