Project Report
Group Badgers

Java and C# in depth, Spring 2013

Thomas Frick (03-150-927)
Matthias Ganz (04-862-850)
Philipp Rohr (04-397-030)

May 11, 2013

1 Introduction

This document describes the design and implementation of the *Personal Virtual File System* of group *Badgers*. The project is part of the course *Java and C# in depth* at ETH Zurich. The following sections describe each project phase, listing the requirements that were implemented and the design decisions taken. The last section describes a use case of using the *Personal Virtual File System*.

2 VFS Core

VFS Core is a library that provides an implementation of a virtual file system. The API that a client of this library can use consists of three interfaces that are described in 2.2.1. The VFS Core provides functionality to create/open/dispose new virtual disks and allows the management of files and directories within such a disk. Furthermore it provides a simple way to import/export files from/to the host file system.

The library internally works with a virtual disk that is divided into a header, index and data section, having the index represented as B-tree. Such a design allows quick access to the data on the disk.

2.1 Requirements

Below one will find a list of the requirements implemented in this project.

2.1.1 disk management

- *The virtual disk must be stored in a single file in the working directory in the host file system.*
- *VFS must support the creation of a new disk with the specified maximum size at the specified location in the host file system.*
- *VFS must support several virtual disks in the host file system.*
- *VFS must support disposing of the virtual disk.*
- *VFS must support querying of free/occupied space in the virtual disk.*

These requirements are met with the classes

`ch.eth.jcd.badgers.vfs.core.VFSDiskManagerImpl`
`ch.eth.jcd.badgers.vfs.core.config.DiskConfiguration`

allow creation/deletion and opening of the disk on the host file system. Clients of the library have to pass a *DiskConfiguration* to the *VFSDiskManagerImpl* when calling *create* or *open*

2.1.2 file management

- *VFS must support creating/deleting/renaming directories and files.*
- *VFS must support navigation: listing of files and folders, and going to a location expressed by a concrete path.*
- *VFS must support moving/copying directories and files, including hierarchy.*
- *VFS must support importing files and directories from the host file system.*
- *VFS must support exporting files and directories to the host file system.*

These requirements are met with the classes

`ch.eth.jcd.badgers.vfs.core.VFSEntryImpl`
`ch.eth.jcd.badgers.vfs.core.VFSPathImpl`
`ch.eth.jcd.badgers.vfs.core.VFSFileInputStream`
`ch.eth.jcd.badgers.vfs.core.VFSFileOutputStream`

The *VFSEntryImpl* allows copy, move (and rename), delete, listing of children and going to the parent (navigation). Together with the streams it also supports importing/exporting to any location clients of the VFS core library wish to. The classes

`ch.eth.jcd.badgers.vfs.ui.VFSConsole`
`ch.eth.jcd.badgers.vfs.ui.VFSUIController`

demonstrate this by importing and exporting to the host file system.

2.1.3 bonus features

- *Elastic disk: Virtual disk can dynamically grow or shrink, depending on its occupied space.*

The implementation expands the underlying disk file up to a specified maximum size if required. Shrinking however is not supported.

- *Compression, if implemented with 3d party library.*
- *Compression, if implemented by hand (you can take a look at the arithmetic compression)*

  The classes

`ch.eth.jcd.badgers.vfs.compression.BadgersLZ77CompressionInputStream`
`ch.eth.jcd.badgers.vfs.compression.BadgersLZ77CompressionOutputStream`
`ch.eth.jcd.badgers.vfs.compression.BadgersRLECompressionInputStream`
`ch.eth.jcd.badgers.vfs.compression.BadgersRLECompressionOutputStream`

implement streams that can be wrapped around *VFSFileInputStream* and *VFSFileOutputStream*. The *DiskConfiguration* allows to switch compression on and to declare which algorithm shall be chosen. This allows easy configuration of any 3rd party compression streams (which was not chosen to implement, because of the implementation of our own compression algorithm).

- *Encryption, if implemented with 3rd party library.*
- *Encryption, if implemented by hand.*

The classes

`ch.eth.jcd.badgers.vfs.encryption.CaesarInputStream`
`ch.eth.jcd.badgers.vfs.encryption.CaesarOutputStream`

show how encryption can be implemented in the library. It was chosen to implement encryption similar to compression with streams, which allows easy configuration via *DiskConfiguration*. These streams are mainly for demonstration how encryption should work and shall not be used in high security environments :-)

- *Large data: This means, that VFS core can store & operate amount of data, that can't fit to PC RAM (typically, more than 4Gb).*

Having all operations implemented with streams achieves this requirement on the fly. Manual tests with 6GB files showed pretty efficient import and export to and from virtual disks.

2.2 Design

This section describes the main aspects of the VFS core library. It shows the implementation of the core interfaces and classes, explains the mock classes and tests and eventually describes the file format and its management classes.

2.2.1 Core Classes

Figure 1 gives an overview of the main interfaces and classes that were implemented. The interfaces *VFSDiskManager*, *VFSEntry* and *VFSPath* can be used by clients using the library implemented here.

- *VFSDiskManager* The implementations of this interface provide mainly a way to open, create and dispose new virtual disks. Additionally one can get the root entry of the file system and get additional information about an opened disk.

- *VFSEntry* Represents a directory or file on the file system. A *VFSEntry* provides all the required methods to manipulate files and directories and importing/exporting files into the virtual file system. The general meaning of *VFSEntry* is, that such objects usually exist on the filesystem.

- *VFSPath* Represents a path on the file system to a given *VFSEntry*. It has a slightly looser coupling to the file system as a path does not imperatively need to exist.

- *FindInFolderCallback* The *find* and *findInFolder* methods on *VFSDiskManager* and *VFSEntry* require a callback object where the find mechanisms provided by the mock and the real implementations will notify the caller when a new entry is found. A client can start the search asynchronously and can handle the notifications for example with updating lists. The console implementation simply lists the absolute paths to the found entries.

The intention of those interfaces is to hide the real implementation of the virtual file system from a client. With that in mind it should be simple to add a network layer upon the real implementation without changing client code. The classes *VFSDiskManagerImpl, VFSEntryImpl (and its descendants) and VFSPathImpl* implement all the management for actually using the VFS on a virtual disk.

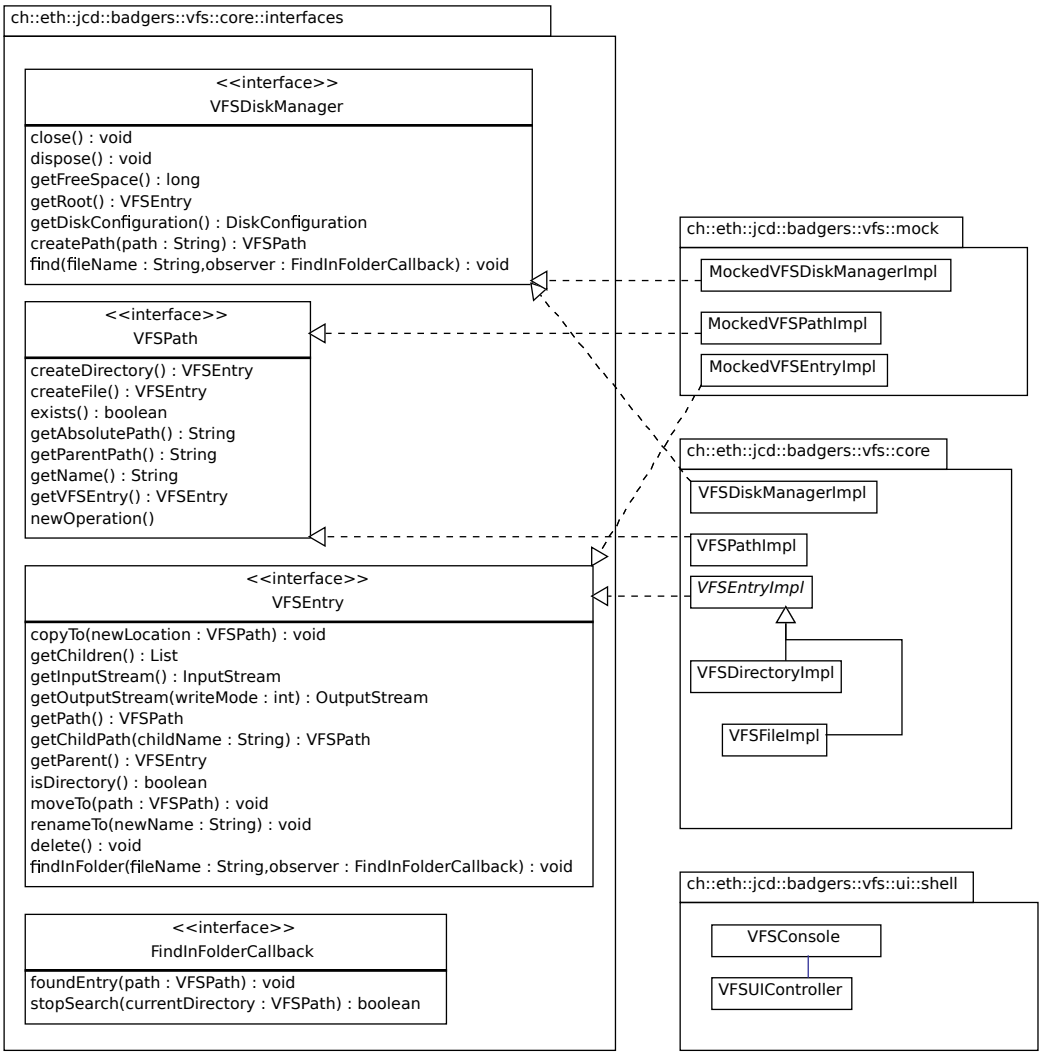

Figure 1: core classes

2.2.2 Design on compression and encryption

Both compression and encryption is only applied to file content. Neither file names nor folder structure is encrypted. Encrypting the whole virtual disk would have greatly increased complexity of the virtual disk core. Considering project deadlines it was decided to only compress and encrypt file content.

2.2.3 Mocking

For discussing the semantics of the virtual file system and the development of the interfaces explained in 2.2.1 it was decided to implement a mock that

works against the host file system. The mock classes implement all the interfaces and were very helpful for acquiring a common understanding of how the interface shall be used by clients. In a further step it was very useful to have the mock classes while developing the console application. Hence the console could be developed independently from the real implementation.

2.2.4 Test

During the development a bunch of test cases came to life. The tests solely depend on the interfaces and thus they can run against the mock classes and the real implementation. This was a huge help in finding bugs in the slightly more complicated real implementation.

2.2.5 The *real* implemenation

Eventually some code was developed that handles a virtual disk. This code and some details about the file format are described in this section.

The classes Figure 2 shows the overview over the classes implemented for the disk handling and file management.

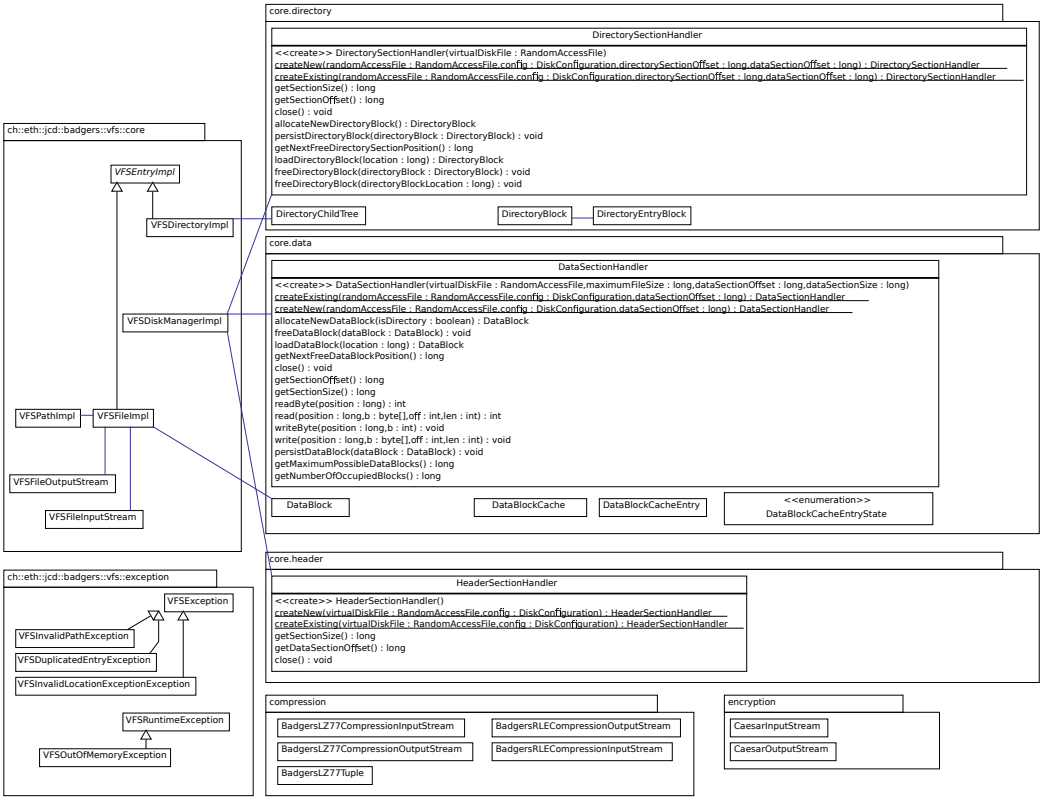

Figure 2: Implementation of the VFS core classes

The classes were divided into several packages which are explained here in more detail.

core The *core* package contains the implementation of the interfaces mentioned in section 2.2.1. Clients of the VFS core library operate on these classes to manipulate the virtual file system.

exception While manipulating the file system some exceptional behaviour might occur and thus some exceptions will be thrown by the core

classes. These exception are propagated to the clients of the VFS core libraries.

core.header As the virtual disk is divided into three sections (header, directory and data) the classes that handle the management of the header section are put in the package *core.header*.

core.data *core.data* contains the classes that manipulate the data section on the file system. This mainly means managing data blocks and actually writing/reading the raw data bytes of files to the virtual disk.

core.directory *core.directory* contains the classes that manage the index of the file system. That means that the whole directory structure is maintained in here. This is done in a B-tree that contains all references to directories and files currently saved in the file system. More about the internals of directory section can be found in section 2.2.5

encryption The encryption package shows some demo classes that implement a Ceasar cipher[1]. These classes are mainly to show how encrpytion in VFS core can be implemented and configured. To enable encryption in a new virtual disk one has to enable it in the *DiskConfiguration* that is passed to the *VFSDiskManagerImpl* at creation. Upon selecting the encryption algorithm the encryption streams will be wrapped around the *VFSFileInput- and VFSFileOutputStream*s.

compression To reduce the data volume within the virtual disk, compression on each file can be enabled. As mentioned earlier, compression is implemented as Input- and OutputStreams and thus is wrapped around the *VFSFileInput- and VFSFileOutputStreams*. Currently available compression algorithms are run length encoding [2] and LZ77 [3][1].

- *Run Length Encoding* The available 8bit run length encoding(rle) algorithm is a very simple form of data compression where multiple occurrence of the same byte were stored as a single byte value and the corresponding count. It is useful for simple graphic images like line drawings and icons.
- *LZ77* Abraham Lempel and Jacob Ziv introduced the LZ77 lossless compression algorithm in 1977. Newer compression methods such as

[1] http://en.wikipedia.org/wiki/Caesar_cipher

GZIP or DEFLATE often use LZ77-based algorithms. The compression is achieved by replacing the data with a reference to an earlier existing copy in the data input stream. For that a window of a certain size is held in memory where existing copies of the current data are searched.

The File Format This section describes the binary file format used by the file system inside a virtual disk. The file is separated into three major parts. The header, index and the data section. Each of them is described below. Figure 3 gives an overview, of how the sections are distributed in the virtual disk and what contents they have.

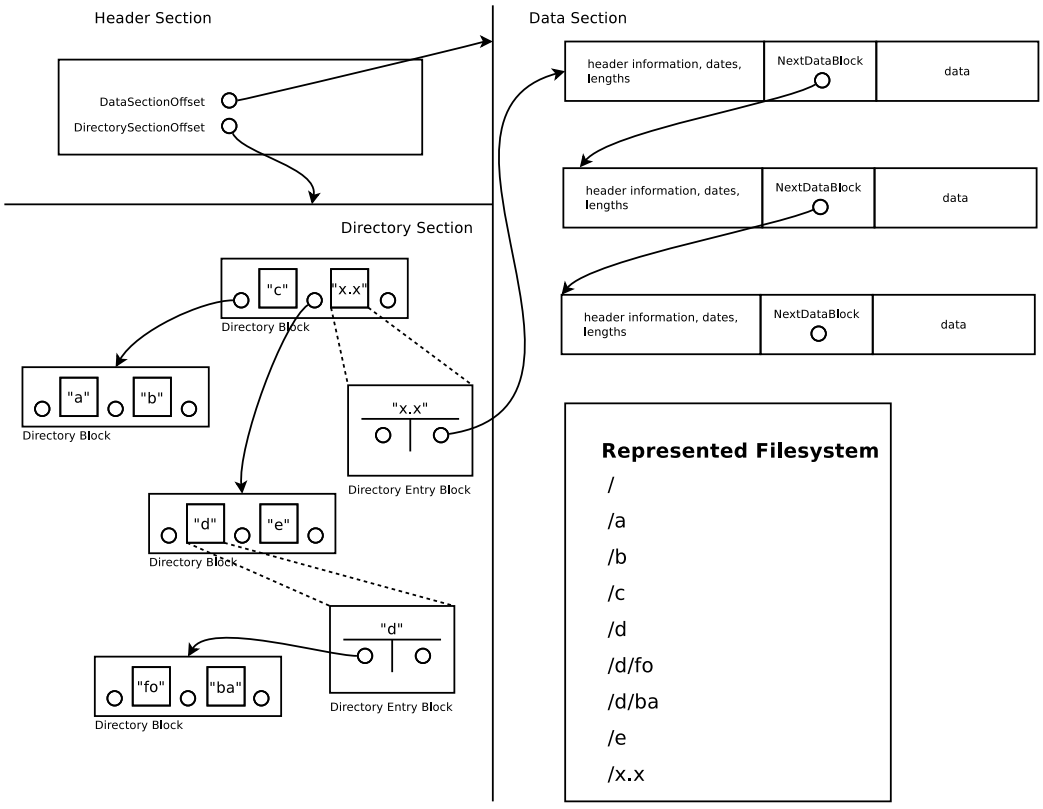

Figure 3: Overview of a disk

Header Section The header section contains some general information about the currently opened virtual disk. The details can be found in the following table:

Name	Length	Description
Info	50 byte UTF-8 String	Contains something like Badger VFS 2013
Version	10 byte UTF-8 String	Contains something like "1.0"
Compression used	20 byte UTF-8 String	null or indicates compression used for this file
Encryption used	20 byte UTF-8 String	null or indicates encryption used for this file
DirectorySectionOffset	long (8 byte)	File offset where the directory section starts
DataSectionOffset	long (8 byte)	File offset where the data section starts
SaltString	8 bytes	Salt used to hash user name and password randomly string generated while creating this file. (Not implemented yet)
Password	xxx bytes	CryptoHash (SHA-whatever) of Password+SaltString (Not implemented yet)

Directory Section The directory section describes which files and folders belong to which parent directory. This section has a fixed size and contains so called *DirectoryBlocks* which also have a fixed size. This makes management and manipulation easy. To each directory belongs a B-tree structure which lists all contained entries.

DirectoryBlock One *DirectoryBlock* represents a node in the B-tree of order 2.

Name	Length	Description
DirectoryHeader	1 byte	Header information. This header makes it easy to determine whether a *DirectoryBlock* is in use or not (a file delete operation fills this DirectoryHeader byte with zeros)
DirectoryEntryBlock1	128 byte	The smaller key inserted into the B-tree
DirectoryEntryBlock2	128 byte	The bigger key inserted into the B-tree
DirectoryBlockLink1	8 byte	Points to another DirectoryBlock which contains keys smaller than DirectoryEntryBlock1
DirectoryBlockLink2	8 byte	Points to another DirectoryBlock which contains keys bigger than DirectoryEntryBlock1 but smaller than DirectoryEntryBlock2
DirectoryBlockLink3	8 byte	Points to another DirectoryBlock which contains keys bigger than DirectoryEntryBlock2

DirectoryEntryBlock Represents a single directory or file.

Name	Length	Description
Filename	112 byte	UTF-8 file name String
DataBlockLocation	8 byte	Pointer to a DataBlock located in the Data Section. This DataBlock holds some meta information about the current directory
DirectoryEntryTreeRoot	8 bytes	Pointer to a DirectoryBlock located in the Directory Section. This referenced DirectoryBlock is the root block of a B-tree containing all entries of that directory specified by the current DirectoryEntryBlock. **This field containing a 0 indicates that this entry represents a file not a directory**

Data Section The data section is split into blocks where each of them is 1024 bytes long. Each block contains some amount of data and points to a subsequent block (simple linked list).

Block layout

Name	Length	Description
BlockHeader 0) Block Header-Bit (LSB) 1) Entry Header-Bit 2) not used 3) not used 4) not used 5) not used 6) not used 7) not used	1 byte	 If set to 1 this DataBlock is in use and contains real data. (Delete a DataBlock by setting this bit to 0) If set to 1 this is the first DataBlock of a file or folder.
NextDataBlock	8 byte	Points to the start address of the next DataBlock (linked list). 0 if this is the last DataBlock of a certain file or folder.
CreationDate	8 byte	UTC time when this file was created *This field only exists if File Header-Bit is set to 1*
LinkCount	2 byte	counter from 0 to 127 which indicates how many DirectoryBlocks point on this specific DataBlock. *This field only exists if File Header-Bit is set to 1*
DataLength	4 byte	Indicates the number of data saved on this DataBlock.
Data	n byte	user data (may be encrypted/compressed)

The root directory By definition the root directory's DataBlock and DirectoryBlock are located at the very first position in their correspond sections.

3 VFS Browser

VFS Browser is a JAVA Swing application atop the VFS core. The browser was developed during the second milestone of the project. The main goal of the browser application is to give users a convenient access to their virtual disks. It allows browsing files and folders, as well as importing and exporting from/to the host filesystems. Additionally it provides a convenient search interface, that allows quick finding of files.

3.1 Requirements

Below one will find a list of the requirements implemented in this project.

3.1.1 Basic Requirements

- *The browser should be implemented on one of the following platforms: desktop, web or mobile*
- *The browser should support all operations from Part 1 (VFS core). For example, users should be able to select a file/folder and copy it to another location without using console commands.*
- *The browser should support both single and multiple selection of files/folders.*
- *The browser should support keyboard navigation. The mandatory set of operations includes folder navigation, going to parent and child folders (this is optional for mobile applications due to limited keyboard functionality).*
- *The browser should support mouse navigation (or touch in case of the mobile platform). The required operations are the same as in requirement 4.*
- *The browser should support file-name search based on user-given keybwords. The search should provide options for: case sensitive/ case insensitive search; restrict search to folder; restrict search to folder and sub folders.*

These requirements are met with the classes

```
ch.eth.jcd.badgers.vfs.ui.desktop.view.BadgerMainFrame
ch.eth.jcd.badgers.vfs.ui.desktop.view.BadgerMenuBar
ch.eth.jcd.badgers.vfs.ui.desktop.view.BadgerTable
ch.eth.jcd.badgers.vfs.ui.desktop.view.DiskSpaceDialog
ch.eth.jcd.badgers.vfs.ui.desktop.view.ImportDialog
ch.eth.jcd.badgers.vfs.ui.desktop.view.NewDiskCreationDialog
```

provide the frontend functionalities to meet the requirements

```
ch.eth.jcd.badgers.vfs.ui.desktop.model.BadgerFileExtensionFilter
ch.eth.jcd.badgers.vfs.ui.desktop.model.EntryTableModel
ch.eth.jcd.badgers.vfs.ui.desktop.model.EntryUiModel
ch.eth.jcd.badgers.vfs.ui.desktop.model.ParentFolderEntryUiModel
```

are the model classes, that model table entries and ui models

`ch.eth.jcd.badgers.vfs.ui.desktop.controller.BadgerController.java`
`ch.eth.jcd.badgers.vfs.ui.desktop.controller.BadgerViewBase.java`
`ch.eth.jcd.badgers.vfs.ui.desktop.controller.DesktopController.java`
`ch.eth.jcd.badgers.vfs.ui.desktop.controller.SearchController.java`
`ch.eth.jcd.badgers.vfs.ui.desktop.controller.WorkerController.java`

provide the controller classes of a typical MVC design.

3.1.2 Implemented Bonus Requirements

- *Responsive UI, i.e. the browser does not stall during long-running operations (i.e. file search or import).* Is implemented with decoupling the disk work from the gui work. This is done with a single thread accessing the disk api. This thread has a work queue to which *Actions* are added.
- *Advanced search; For example, search with wildcards.* For this the API had to be extended slightly.
- *Drag-and-drop for manipulative operations (import).* For that we register a *DropTarget* on the *BadgerTable* that has a *DropTargetListener* which starts the import on the *DesktopController*.
- *Nice-to-have features like operation progress report (e.g. the number of files processed during export)* Is implemented as a small popup window, that kicks in after an action takes more than 500ms.

3.1.3 Not Implemented Bonus Requirements

- *Nice-to-have features like drag-and-drop for manipulative operations (move, copy).*
- *The browser is implemented for an additional platform*
- *Efficient full-text search (using some sort of indexing).*

3.2 Design

This section describes the main aspects of the VFS Browser application. It shows the implementation of the core interfaces and classes.

3.2.1 GUI classes

Figure 4 gives an overview of the main interfaces and classes that were implemented for the GUI. The implementation followed the rules of a MVC design dividing the aspects of Model View and Control to their respective classes. According to the partition of MVC the classes were partitioned into the packages *model, view and controller*.

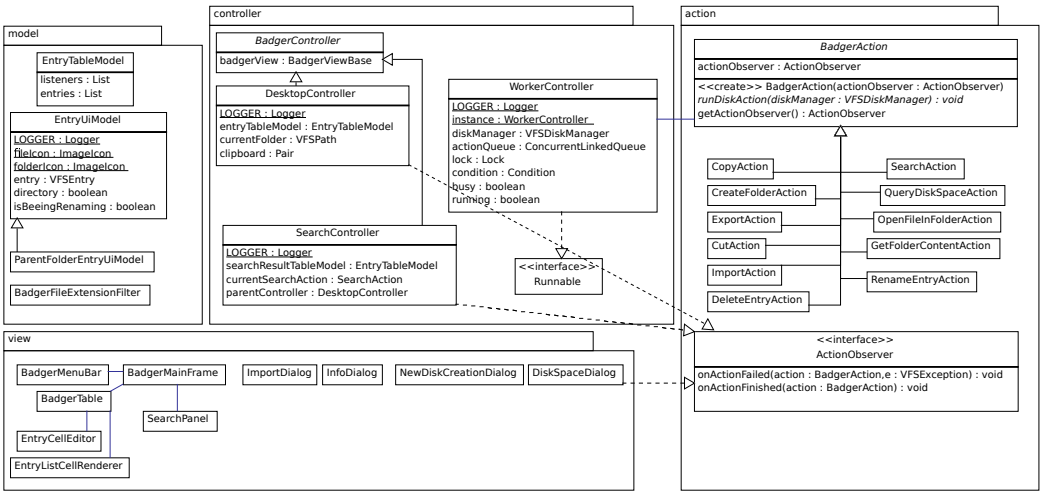

Figure 4: gui classes

3.2.2 Decoupling of GUI and working threads

This section describes the way how the decoupling from GUI and working threads was implemented. This decoupling made the GUI still responsive even though some long running tasks like import or search would be running. All operations performed on the core components accessing the virtual disk file are encapsulated in subclasses of *BadgerAction*. These *BadgerActions* are then performed single threaded by the *WorkerController*. By queuing these actions concurrent access to the virtual disk is prevented.

Figure 5 shows the sequence diagram of the example "Import": The *DesktopController* which lives in the GUI thread creates a new *ImportAction* that is enqueued in the *WorkerController*'s action queue. It is worth to

mention, that the *WorkerController* is a singleton, that has an *instance* field. The *WorkerController* has a thread that works on a blocking queue and every time it gets a *BadgerAction* the thread wakes up and performs the action on the VFS core API. After the API work is done (in this case the import of files), the *WorkerController* calls the corresponding *ActionObserver*, which is in most cases the *DesktopController* instance, so that the GUI can be updated. This design ensures single threaded access on the VFS core API with keeping the GUI responsive.

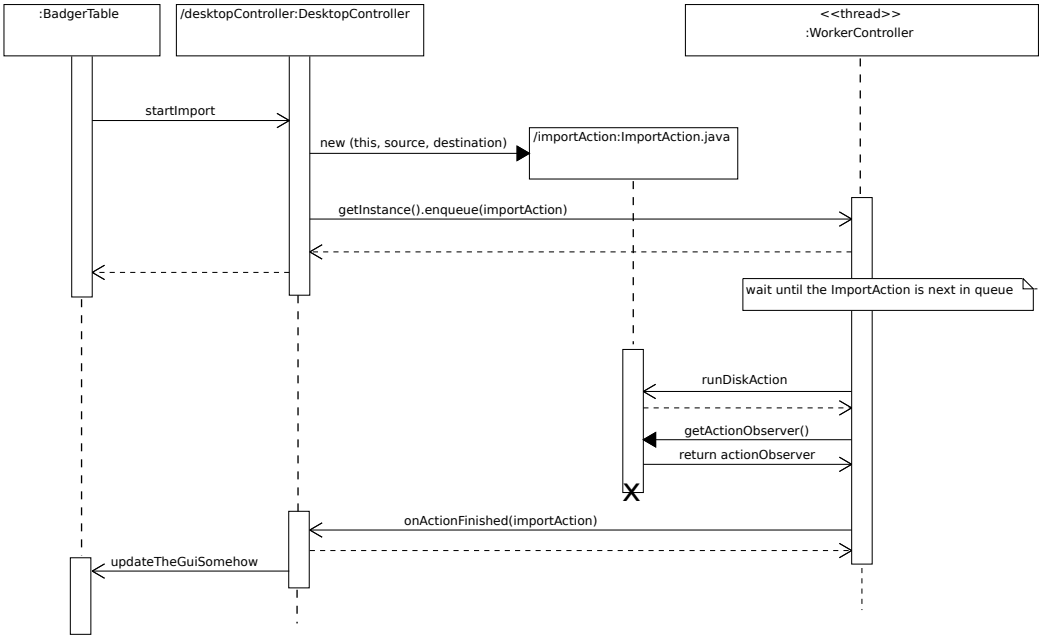

Figure 5: example of decoupling import from gui

Even when doing a seach the decoupling works pretty well: The *SearchAction* uses the VFS core's *FindInFolderCallback* to update the *SearchController* every time finds an entry. Upon such an update, the *SearchController* updates the GUI.

3.2.3 Search

Searching files can be done in a separate Panel: After entering the search string one is allowed to switch several flags like "case sensitive" or "search subfolder" and one can change the search folder. As mentioned before ? and * are supported as wildcards.

Search can be cancelled on the GUI at any time. However the search routine performed on the core checks for cancellation only once walking through

a directory (which should be often enough).

A double click (or keyboard enter) on a search result stops the current search and opens the containing folder in the browsing window.

3.2.4 Keyboard and Mouse support

One can navigate the list of entries by double clicking folders. The parent folder can be accessed by double clicking "..". By right clicking on an entry the user gets a menu depending on whether she clicked on an file, folder or the empty space. The right click in empty space reveals the same menu entries as the right click on the parent folder. The only difference between file and folder menus are the "new folder", "paste" and "import" actions, that are not allowed on files.

The user is also allowed to select several files and folders from the list and can then perform cut/copy or export on the selected items.

Keyboard Shortcuts For keyboard wizards there are several keyboard shortcuts for doing the actions on the selected entries. They keep to the general standards like F2 for renaming or Ctrl+C for copy. The menus indicate the available shortcuts.

Drag & Drop For doing quick import, the user is allowed to drag&drop files or folders from the host file system to the current folder. On doing such an import a small progress window is displayed, that show how much of the task is already finished, and how much has yet to be done.

3.3 Integration

For the integration of the VFS core into the VFS Browser application almost nothing had to be changed in the core. As the core is meant to be accessed by only one thread at a time, the user interaction from the GUI had to be bundled into the *WorkerController* and could be used from there without changes. The only improvements on the core were done for the search, which allows some wildcards that were not yet there in the first part of the project.

4 Synchronization Server

Synchronization Server is a command line server that allows synchronization of virtual disks. It allows basic user management via VFS Browser, that includes creating users and a login of created users. Logged in users can link local disks to an account on the server via VFS Browser and then can connect to such shared disks from an other client machine. Disks can also be created directly on the server. When VFS Browser has a connection to the server, changes that occur on the local disk will be propagated to the server as soon as the "sync" button is clicked. The server then propagates these changes to possibly other connected machines. If a user has no connection to the synchronization server but operates on a disk that is linked to a server instance the changes are recorded in journals and are propagated as soon as the connection is established next time. Thus synchronization server allows seamless synchronization between several VFS Browser instances running on different machines.

4.1 Requirements

Below one will find a list of the requirements implemented in this project.

4.1.1 Requirements for the browser

- *The browser should allow the user to create a new account or to log in to an existing account.*
- *The browser should offer to switch to an online mode, and be able to operate without a connection to the server.*
- *The browser should support binding an existing virtual disk to an active account.*

4.1.2 Requirements for the server

- *The server should support registration of unique accounts. Each account includes name & password.*
- *The server should track changes to linked virtual disks of registered accounts and synchronize the changes across the machines.*

4.1.3 Implemented Bonus Requirements

- *Provide a set of mocked unit tests for your implementation.*
- *Conflict Resolution: implement a conflict resolution scheme, so that concurrent changes to the same file are not lost (e.g. saving conflicting files as separate versions).*
- *The browser is updating automatically when changes to the disk occur.*
- *The server is able to synchronize changes, which are done simultaneously on the same account on different machines*

4.1.4 Not Implemented Bonus Requirements

- *File History: provide a history for files and an interface to restore a previous version*
- *Incremental Changes: minimize the communication between the browser and the server by only transferring the parts of a file that changed.*

4.2 Design

4.2.1 Scenarios

Since milestone 3 VFS Browser can run in two different modes. Classic mode6 basically is the usage of virtual disks without any server involved. Sync mode7 allows propagating changes on a local disk (step 1) to a server (step 2). In this mode changes are recorded in a journal, that gets sent over the wire when the "sync" button is hit. To use a disk in sync mode one has to link a disk to a server so that synchronization takes place between the local disk and the disk on the server. The mechanism of journaling allows offline usage of a linked disk, so that later on, when reconnecting to a server, local changes get propagated to the disk on server side. Such changes on the server side are sent to freshly attached clients that are working on the same linked disk.

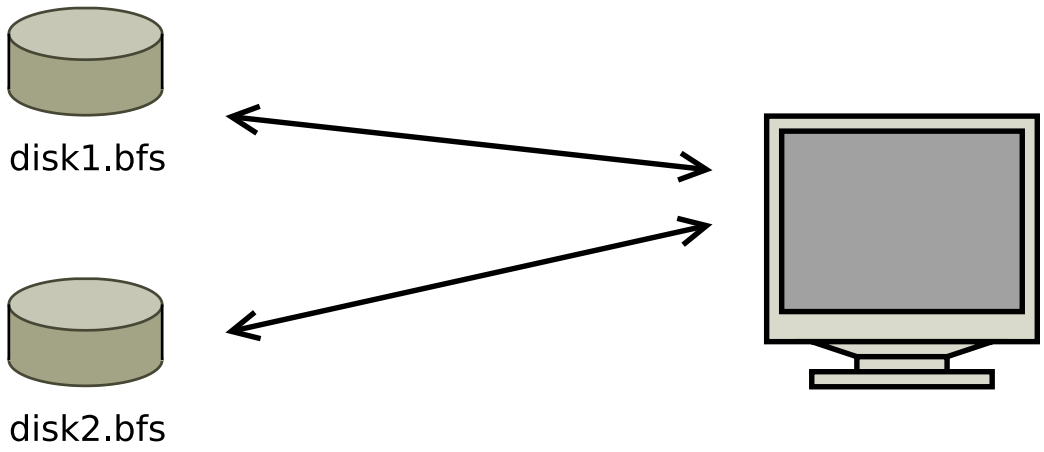

Figure 6: scenario classic mode

4.2.2 Technology

It was chosen to use JAVA RMI to establish client server communication. Figure 8 shows the interfaces and classes, that werde implemented to achieve communication. The three interfaces *LoginRemoteInterface, AdministrationRemoteInterface, DiskRemoteInterface* hide the functionality to login/logout, create/link disks, and push/pull journal updates for a disk. When a server starts up the *LoginRemoteInterfaceImpl* gets registered into the RMI registry and from now on clients are allowed to connect through its interface. Every time a client successfully logs in to the server a new *AdministrationRemoteInterfaceImpl* gets instantiated so that the client can link or open disks.

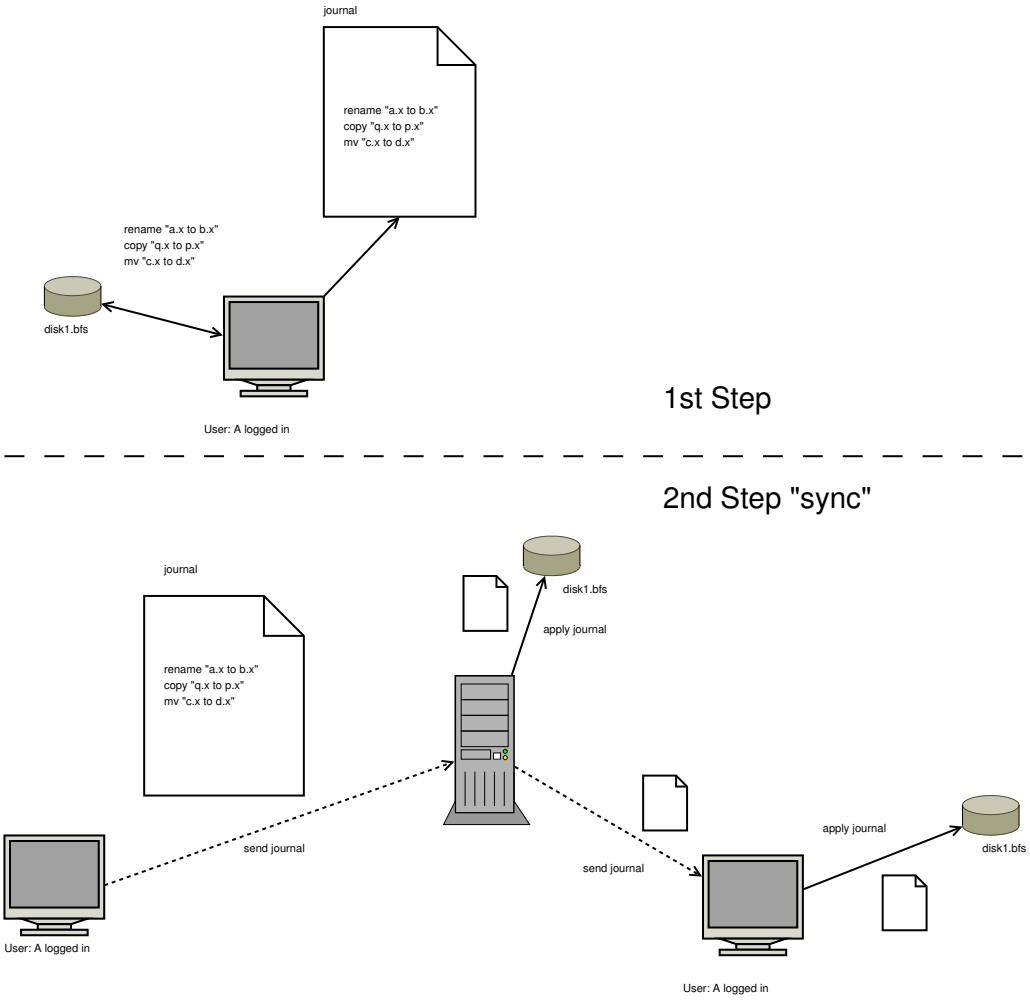

Figure 7: scenario sync mode

Every time a disks gets opened a new instance of *DiskRemoteInterfaceImpl* gets instantiated through which the client can synchronize local changes to the corresponding server disk.

4.2.3 User Management

User management is kept very simple. Upon connecting to a server one can create a new username/password or use a already existing one. Credentials are being saved on server side together with its linked disks (see *UserAccount*).

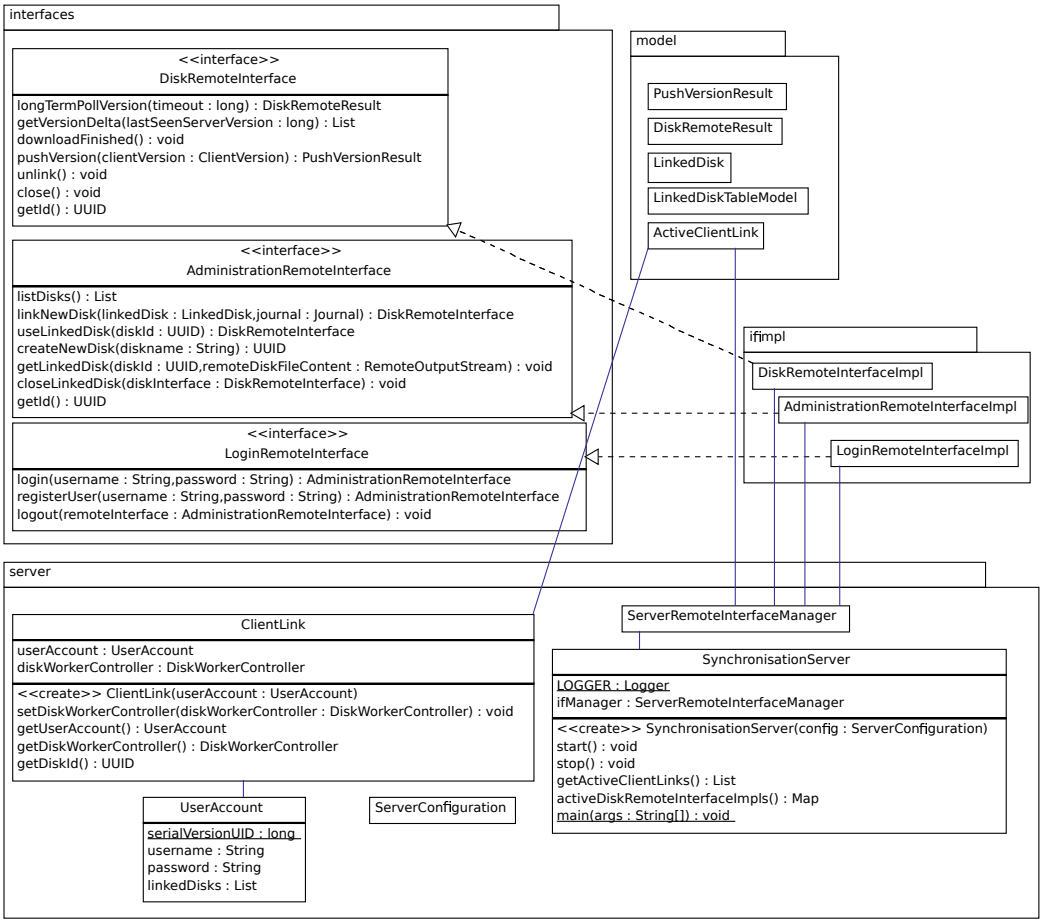

Figure 8: Server classes

4.2.4 Synchronization

Synchronization of disks is achieved by a simple journaling mechanism. Journals get recorded when changes on a linked disk are made. These recorded journals get sent to the server when "sync" button is hit. Changes that were done on a remote client get synchronized to the local client upon login or periodically with a long term polling mechanism. Figures 9 and 10 show roughly, what happens when the "sync" button is pressed: Through some indirection (*DiskWorkerController and RemoteWorkerController* that allow loose coupling of GUI, local disk and remote communication threads) the client uploads/downloads *Journal*s containing *JournalItem*s to/from the server which then get replayed on the other side. Notice in particular the *ModifyFileItem* that carries a special variant of *InputStream* to copy the binary data from a file to the other side. A small drawback of this design is

that while client is doing synchronization no other actions can take place on its local disk. This includes changing folder or simply all reading actions. Due to the lack of time it was decided not to investigate further on that topic.

Sync Button It was explicitly chosen to implement a sync button, to quickly test several scenarios with different clients. Having this button makes it easy to set up two or three clients that have a concurrent actions on files and folders and to check wheter the conclict resolution works. It would be very easy to add up/download on a periodic basis or even when a client action is finished. Of course such scenarios were also build with unit tests.

4.2.5 long term polling

To make other clients aware that changes happened on their disk, a long term polling mechanism was implemented. Clients start the *longTermPollVersion* method in the *DiskRemoteInterface*. Upon files get changed by a client the other clients that are currently connected to the same disk get notified by returning the *longTermPollVersion* method, containing the actual server version, to which the clients now have to update.

4.2.6 Journaling

Figure 11 gives an overview over the classes that were implemented for journaling. The main thing to notice are the *JournalItem*s that carry out the operations they represent (e.g. *RenameEntryItem* renames a given entry to a new one.). For every disk a logical version number is kept to make sure that no journals collide. Every time a client synchronizes the last known server version number gets sent to the server so that the server can decide which journals have to be propagated to the client. Journals are kept inside the disk in a folder `/.hidden/journals/` and get deleted on the client side, as soon as the local changes are propagated to the server. If file contents have to be transferred, they are copied and saved under a uuid in a path that looks like `/.hidden/journals/[journalnr]/file-[uuid]`. Copying is done very quickly via a shallow copy mechanism, that had to be implemented in VFS core. On the server side the disks mainly consist of the whole journal from the beginning of time. With more time left for development one could easily show a file history or even revert to any point in time.

Conflict resolution Based on the logical version numbers that are given to each journal the client is able to do basic conflict resolution when it detects

conflicting changes. Files that got new content on the server while they were changed locally will be renamed according to following scheme:

1. revert local changes
2. replay server journal (the client is in sync with server now)
3. redo local changes
4. push local changes to server

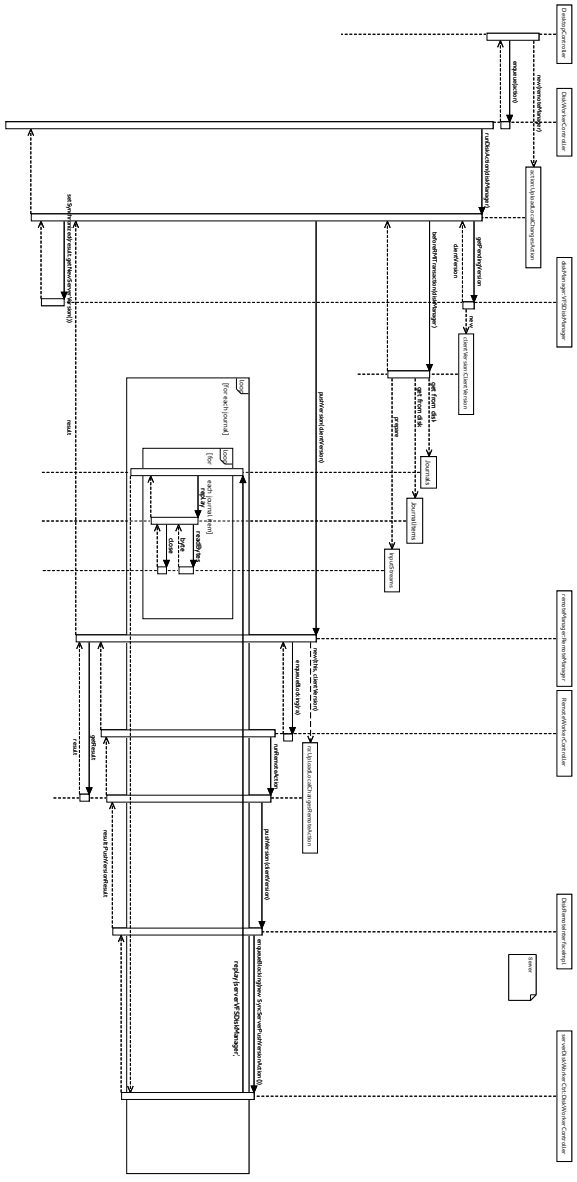

Figure 9: from click to upload changes to server

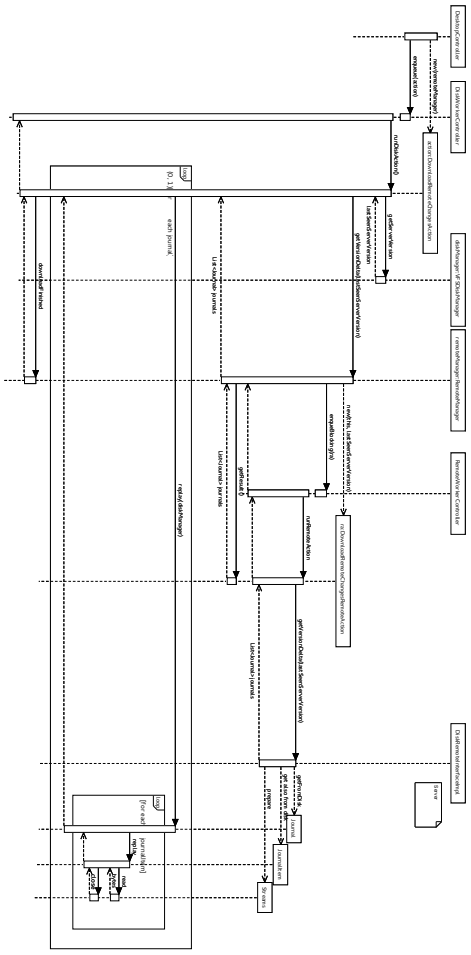

Figure 10: from click to download changes from server

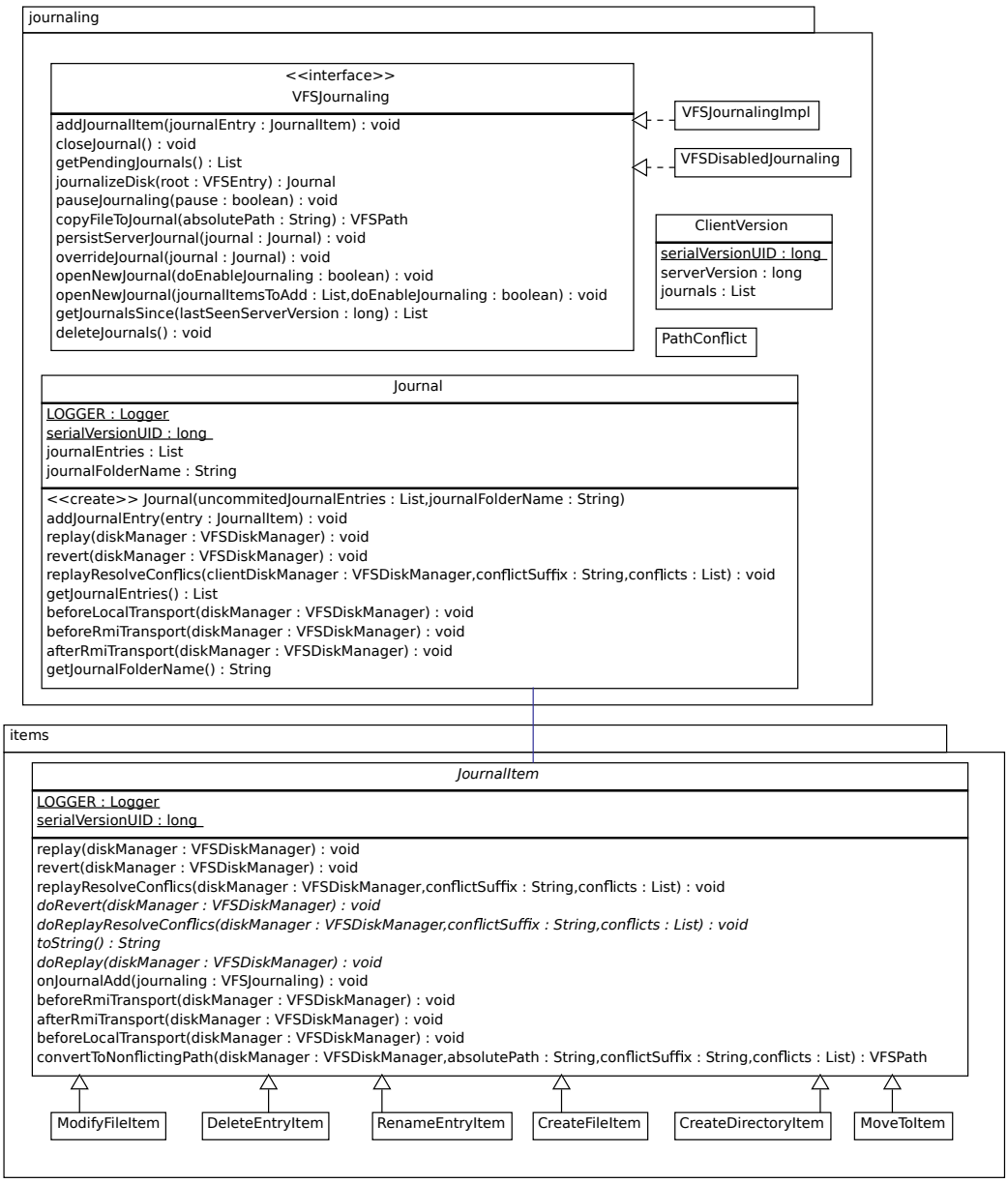

Figure 11: Journaling classes

4.3 Integration

To achieve goals like file synchronization, conflict management and history browsing of a file, the interfaces developed in the second part of the project were no longer good enough. Thus the journaling classes were integrated into the core library. These classes help keeping track of the actions that take place on the file system and are used to send changes on a local disk to the server. To have journaling running as efficiently as possible the shallow copy mechanism was developed. This mechanism does not copy the full file content but just has reference counting on the datablock. The counters get increased/decreased as copies are taken into the journal or deleted. This shallow-copy mechanism only works, because files cannot be changed on the file system, they have to be renamed or deleted before new content can be stored under the same name.

5 Quick Start Guide

5.1 the eclipse project

The project requires to be compiled with JAVA 7. It also depends on the maven plugin which pulls in all the required libraries.

5.2 Command line client

The command line client allows the usage of the VFS core and is mainly intended to test the basic functionalities. The console runs either in management mode or in file system mode. The management mode is entered automatically when starting the command line client. It allows creating and opening virtual disks. The file system mode is entered as soon as a virtual disk is opened.

5.2.1 startup

The command line client can be started as follows:

```
java -jar VFSCore.jar ch.eth.jcd.badgers.vfs.ui.shell.VFSConsole
```

or by starting `ch.eth.jcd.badgers.vfs.ui.shell.VFSConsole` in eclipse.

This gives a console prompt where the following commands can be used in.

5.2.2 commands

Following commands can be used with the command line client in management mode:

- **create c:\path\to\disk.bfs [size]** creates virtual disk with a maximum quota of [size] megabytes on the host system. The file may grow up to [size] megabytes. There is currently no way to change encryption or compression by using the console application. By default no encryption and the LZ77 compression will be used.
- **open c:\path\to\disk.bfs** opens file system mode for the given virtual disk
- **exit** exits the console program

following commands can be used in file system mode:

- **ls** lists the contents of the current directory
- **pwd** shows the path to the current directory
- **df** shows the usage of the current virtual disk space
- **cd dst** changes current directory to *dst* which must be either a child directory of the current path or ".."
- **find searchString** lists absolute paths of all files containing *searchString* in their file name
- **mkdir dirName** creates a new directory *dirName* in the current path
- **mkfile fileName** creates a new empty file *fileName* in the current path - this is rather not useful, as the "import" creates a file with content
- **rm file** deletes the entry denoted as *file*, it must be a child of the current path
- **cp src dst** copies the *src* file to *dst* as a child of the current path
- **mv src dst** moves the *src* file to *dst*
- **import ext_src dst** imports a *ext_src* from the host system to *dst*
- **export src ext_src** exports a *src* file to the host system *ext_dst*
- **find searchString** lists all file system entries below the current entry containing *searchString*
- **dispose** deletes the currently opened virtual disk
- **close** closes the file system mode, from now on management mode commands can be executed

5.3 VFS Browser

5.3.1 startup

The VFS Browser can be started as follows:

```
java -jar VFSCore.jar ch.eth.jcd.badgers.vfs.ui.desktop.view.BadgerMainFrame
```

or by starting `ch.eth.jcd.badgers.vfs.ui.desktop.view.BadgerMainFrame` in eclipse.

Figures 12, 13, 14, 15, 16 and 17 give a short overview of opening the browser and importing the first folder. From there on the UI is pretty much self explanatory of the features it exposes.

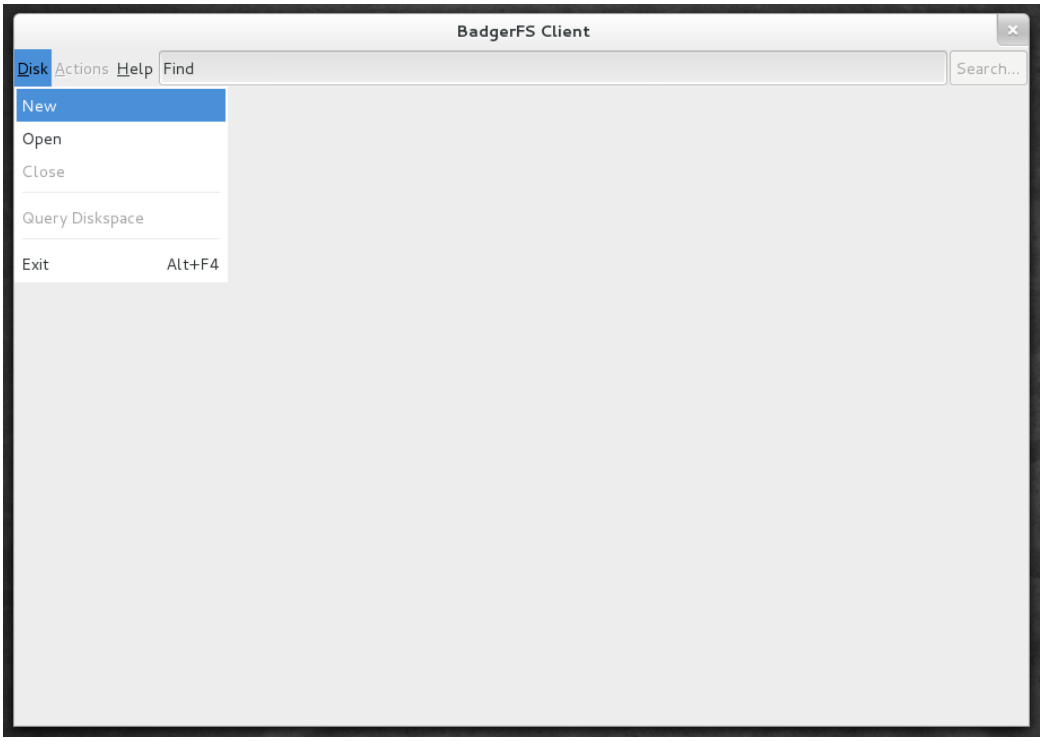

Figure 12: after startup, create a new disk

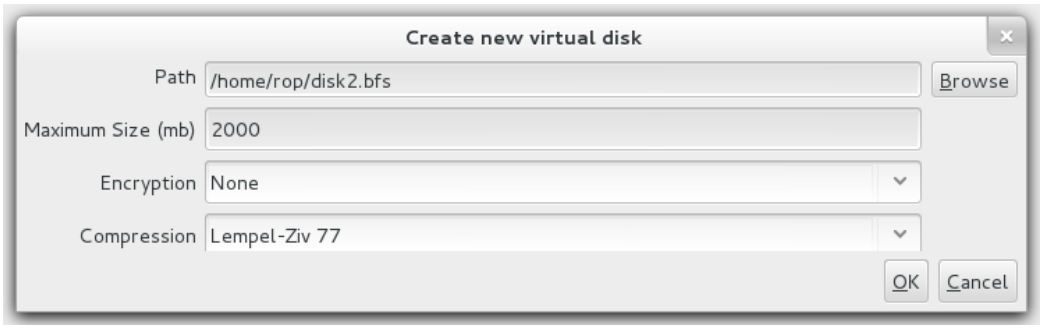

Figure 13: the disk creation wizard opens, one can choose the file, file size, encryption and compression algorithms

5.4 Disk sychronization

This section show how to create a new account, log in from two different machines, import a directory recursively on the first client and export the directory on the second client. Figures 18 through 30 give a short screenshot view of the steps taken.

Start synchronization server on localhost: Start *SynchronizationServer*

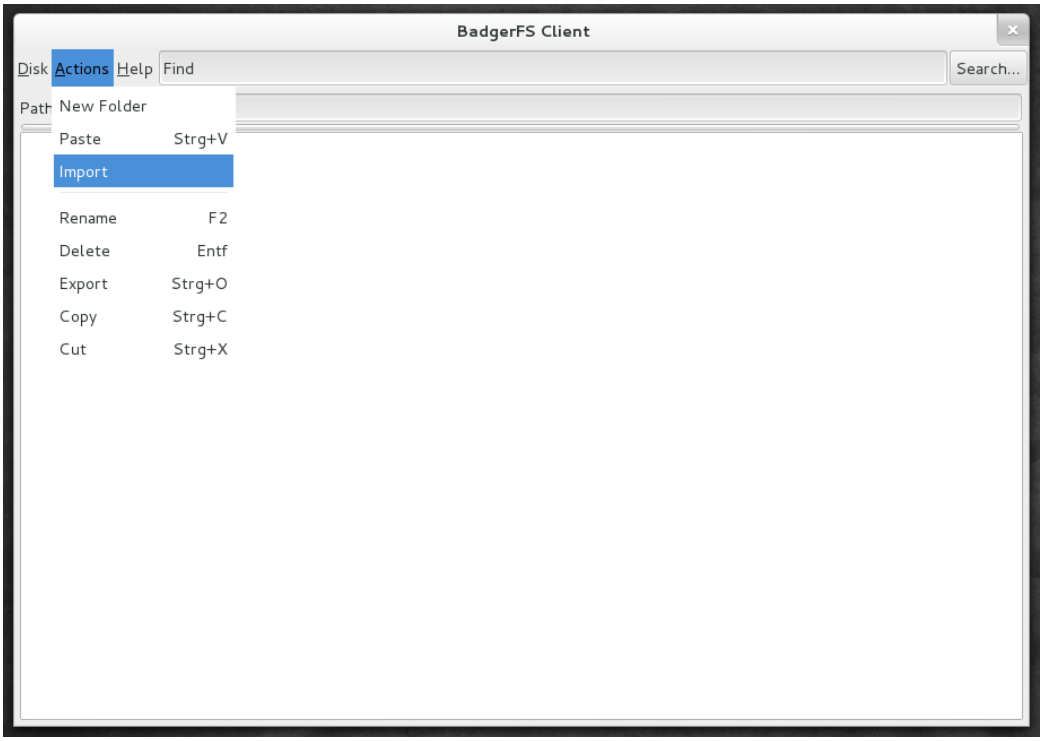

Figure 14: the new disk gets created on the host file system and the user can start importing files via Actions/Import...

via eclipse. Giving following program arguments allow some logging output and configuration of the server side files:
`-l log4j\_server.xml -c /home/rop/Desktop/badger/server/`

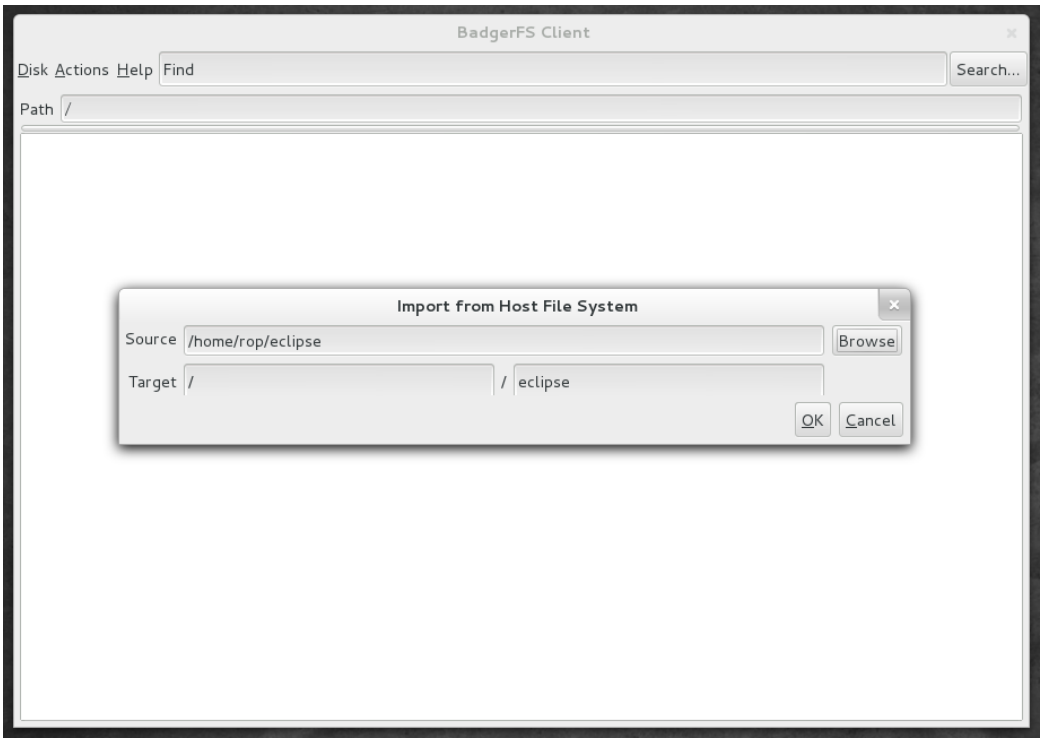

Figure 15: . . . which triggers the import dialog, where the user can choose a file or folder. . .

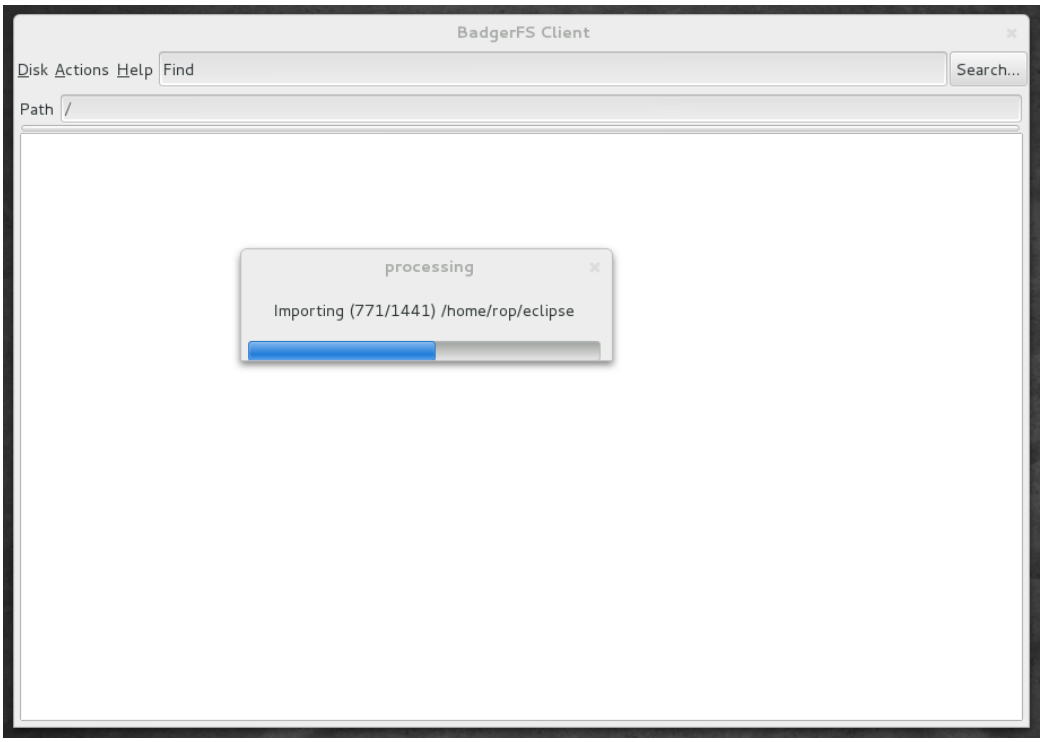

Figure 16: after clicking “OK” the progress window appears...

Figure 17: ... and finally the folder is imported after a couple of time.

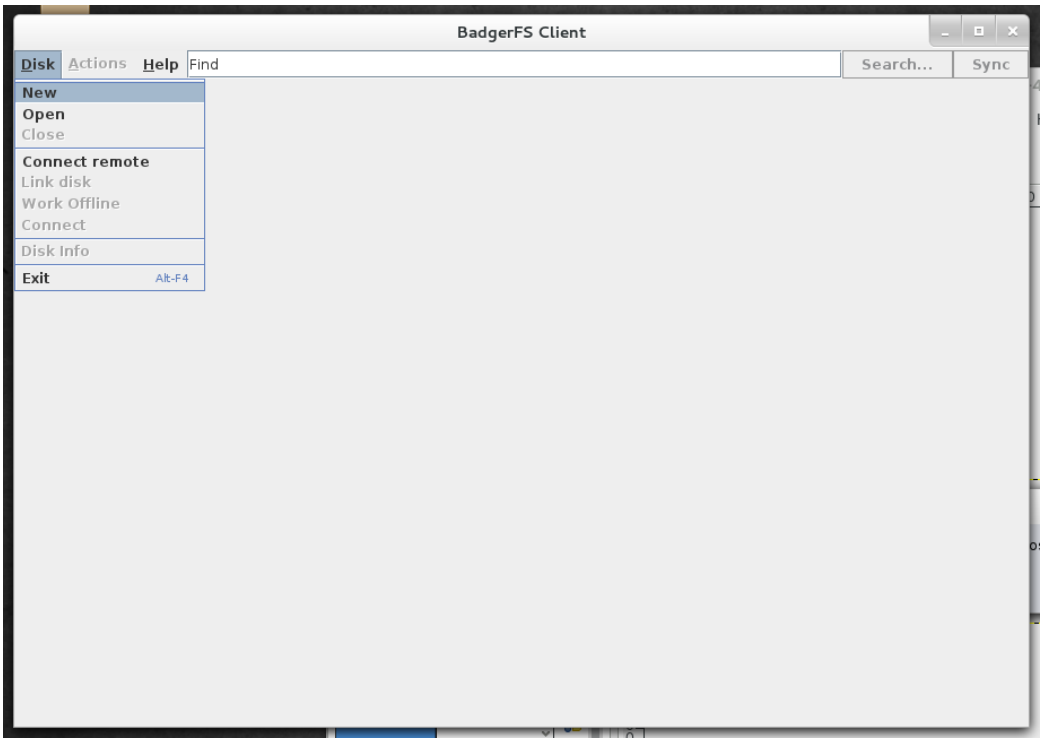

Figure 18: Create a new local disk according to figures ...

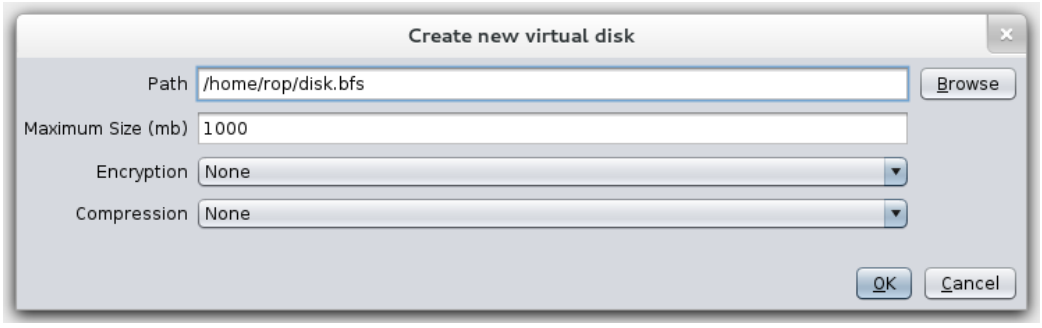

Figure 19: ... and set parameters ...

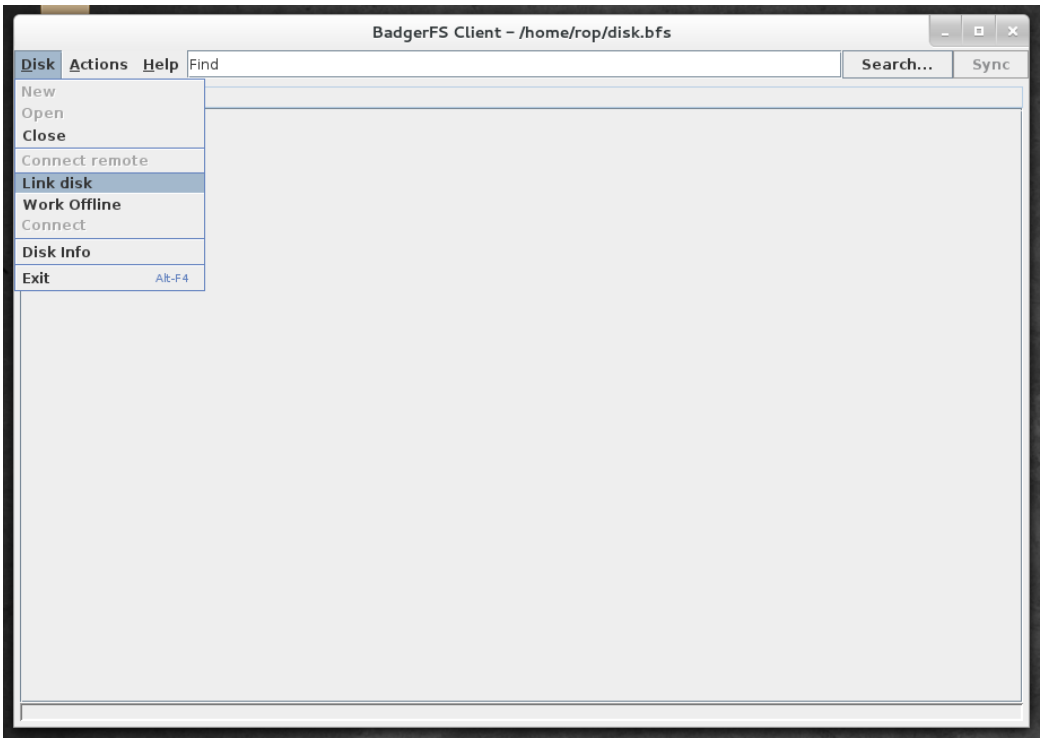

Figure 20: . . . open “link disk”. . .

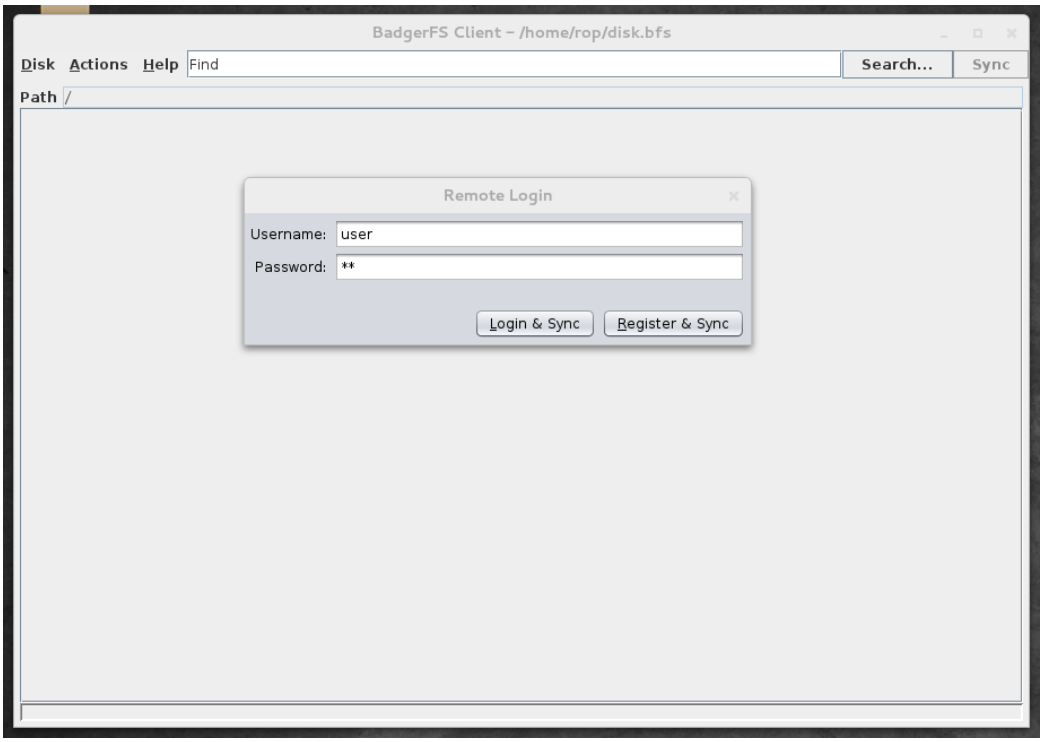

Figure 21: . . . enter new username and password, hit "Register & Sync". . .

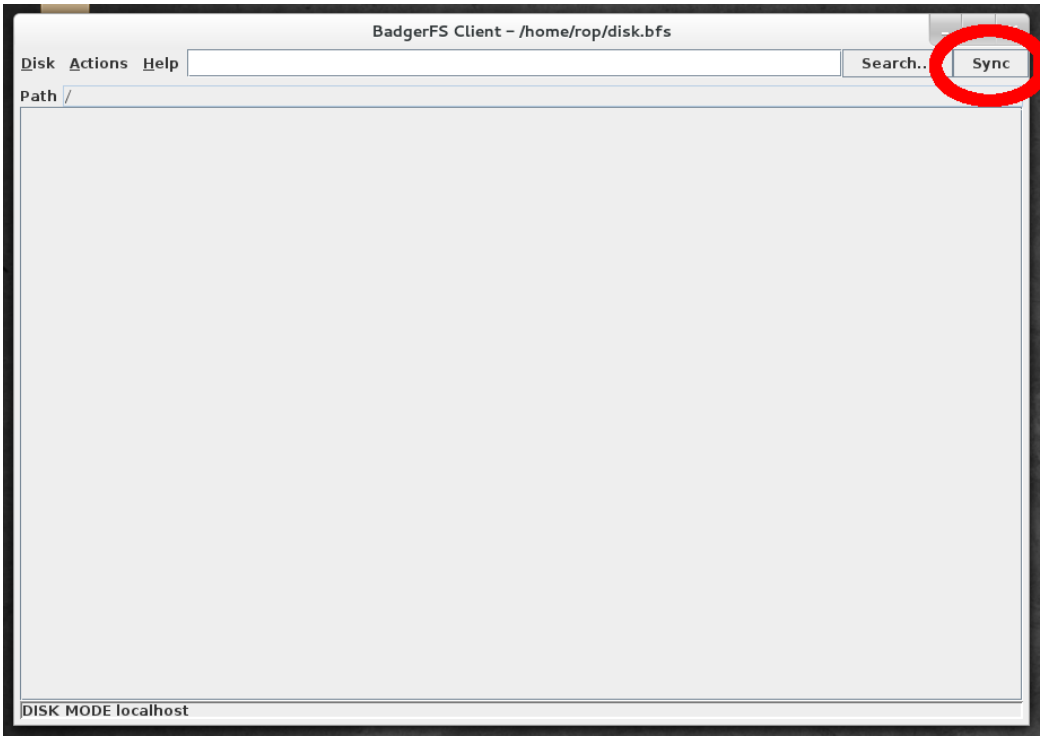

Figure 22: . . . disk is now linked, “Sync” button is available. . .

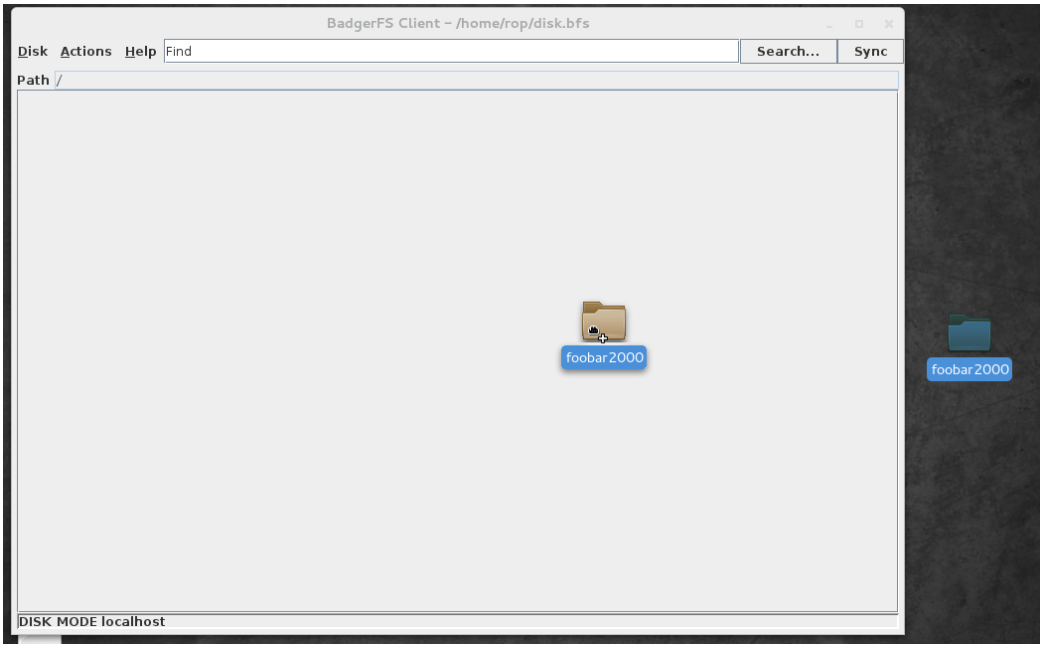

Figure 23: . . . drag and drop a folder into the grey area. . .

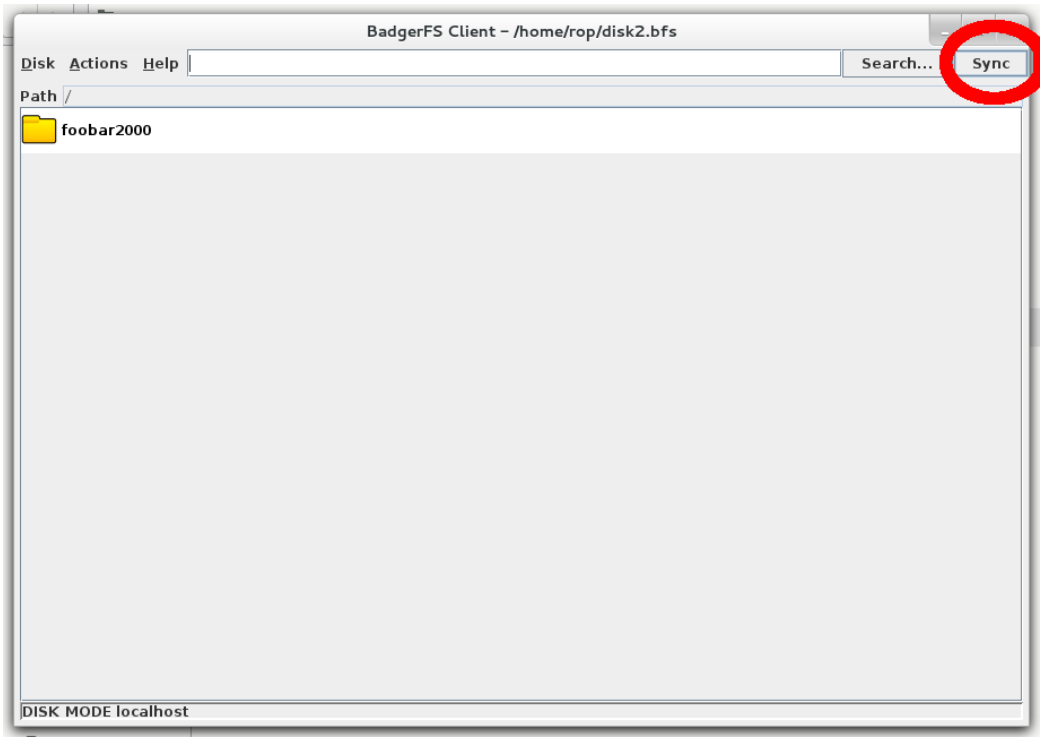

Figure 24: . . . the folder gets locally imported, and then hit "Sync" and the folder gets uploaded . . .

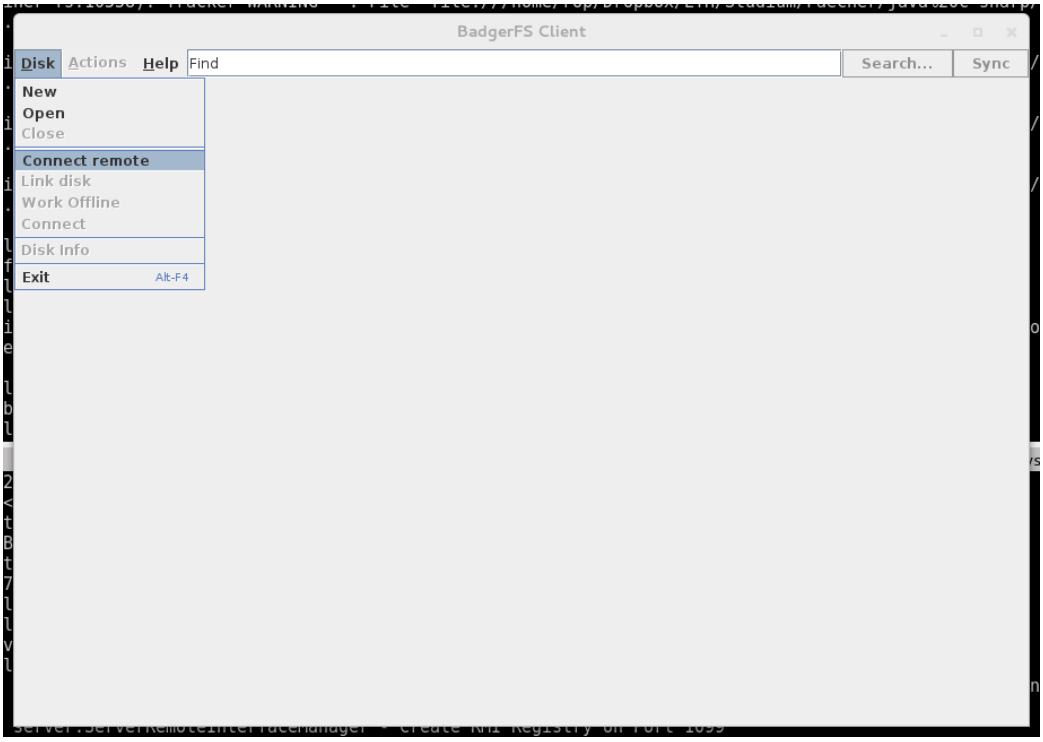

Figure 25: . . . in a new client open “connect remote” . . .

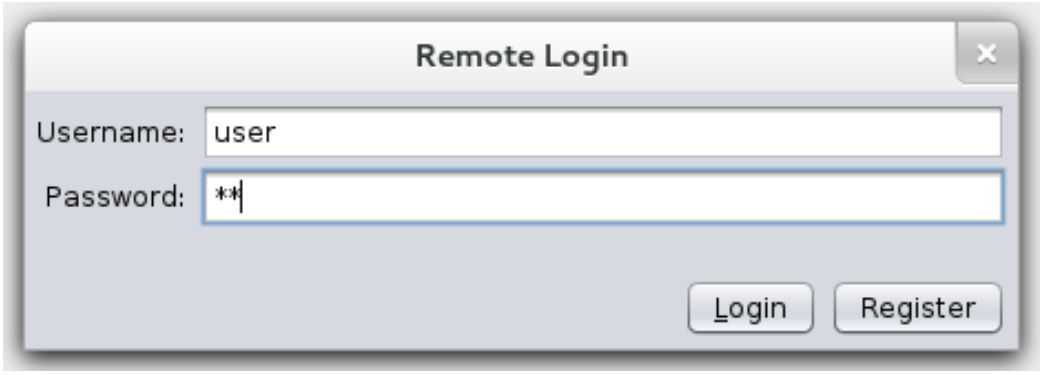

Figure 26: . . . do the login with the old credentials . . .

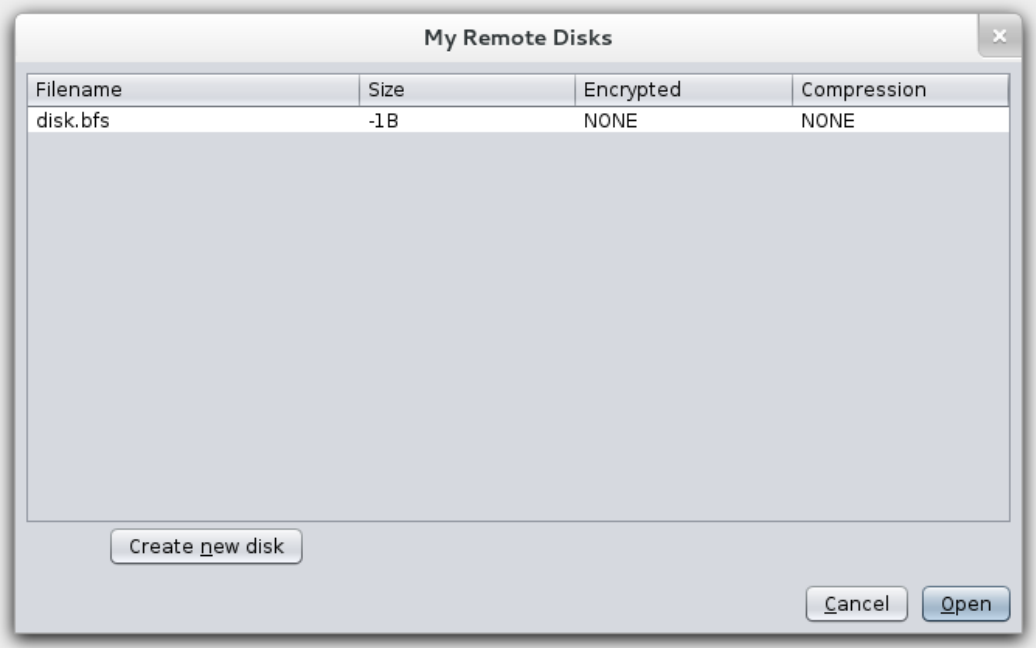

Figure 27: . . . now you can choose the disk created remotely and open it . . .

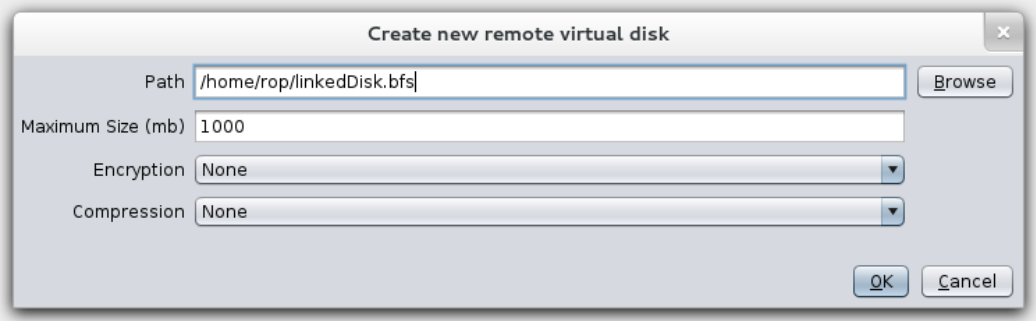

Figure 28: . . . here you create a new local disk to synchronize the server file to . . .

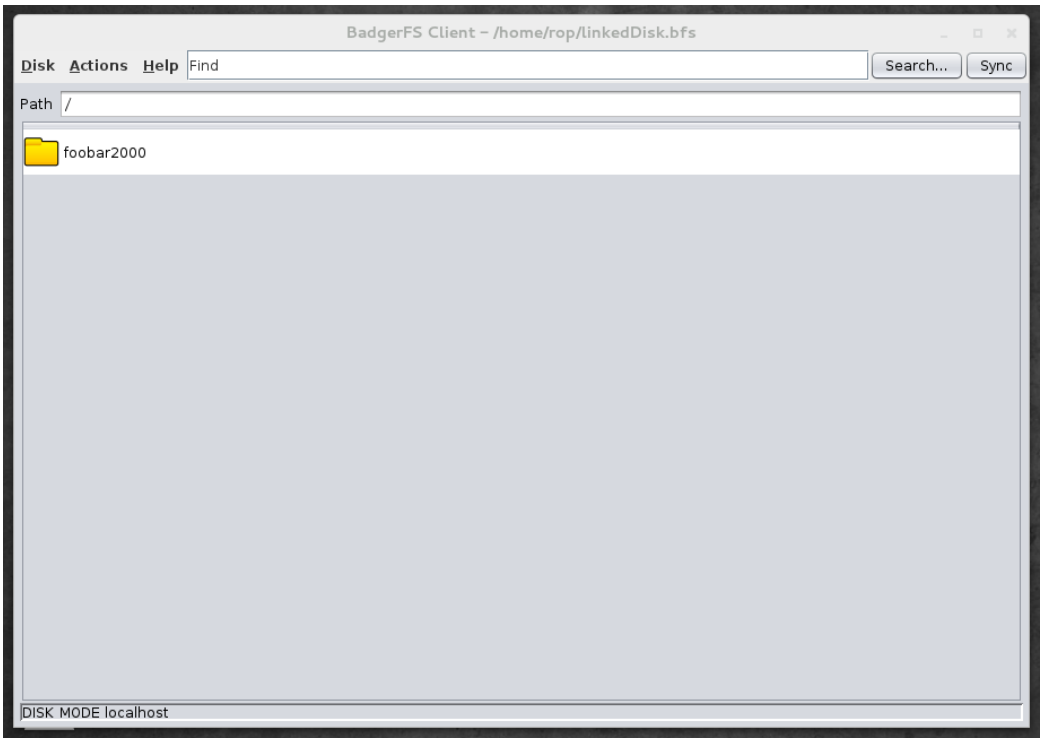

Figure 29: ... upon creation the remote files get synchronized down.

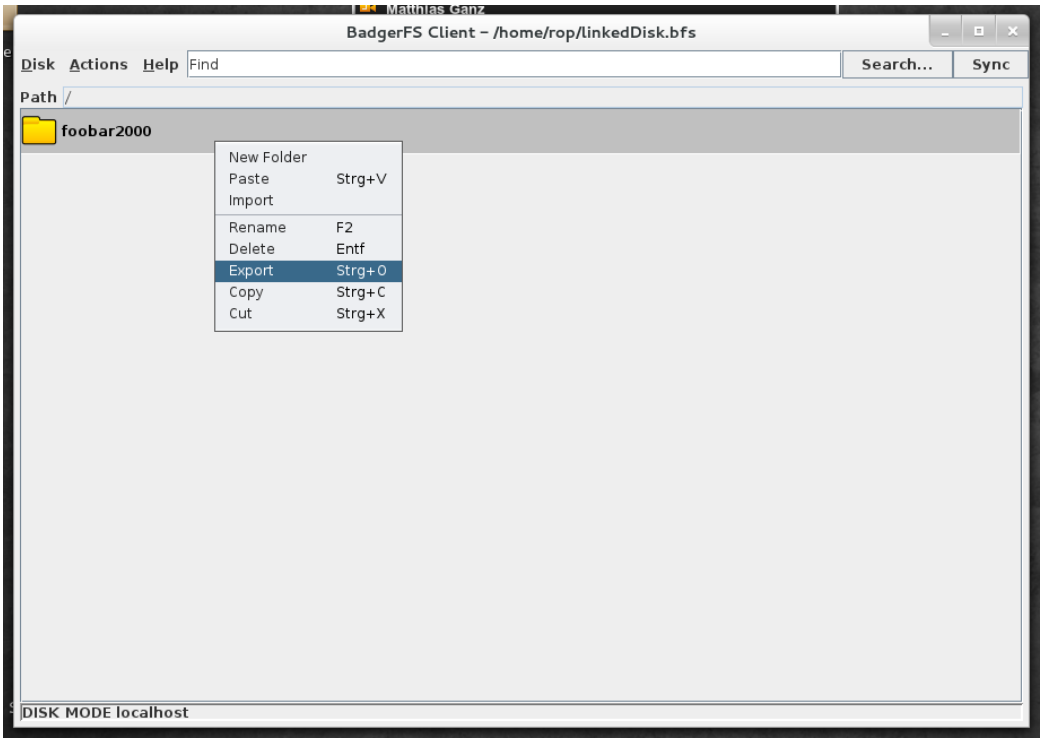

Figure 30: via right click - export one can export the freshly created folder.

6 Glossary

VFS core The main Java library, that handles all the interaction with virtual disks and importing/exporting/storing files. It is used by the command line client and the GUI.

Virtual Disk A virtual disk denotes a container file that is stored on the host file system. A virtual disk can be opened with the software that is developed during this project and stores the actual files. The file extension of the virtual disk is "*.bfs".

References

[1] Richard Baldwin. Understanding the lempel-ziv data compression algorithm in java. *http://www.developer.com/java/data/article.php/3586396/Understanding-the-Lempel-Ziv-Data-Compression-Algorithm-in-Java.htm*, 2006.

[2] Arturo San Emeterio Campos. Run length encoding. *http://www.arturocampos.com/ac_rle.html*, 1999.

[3] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23, 337-343*, 1997.