Project Report
Group Pandas

Java and C# in depth, Spring 2013

Christian Klauser
Sander Schaffner
Roger Walt

May 13, 2013

1 Introduction

This document describes the design and implementation of the *Panda Virtual File System* of group *Pandas*. 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.

2 VFS Core

The VFS Core (assembly Panda.Core) is the actual implementation of the virtual file system. It is a library that manages directories and files within virtual as we are used to it from other file systems. Clients are presented with a simple API that models the virtual file system as a hierarchical structure. Implementation details are well hidden and the libraries design lends itself to isolated testing of its individual components.

2.1 Design

The VFS Core is divided in three layers. This makes the library easier to understand and maintain, as the individual layers need not concern themselves with the entire complexity of virtual file systems as a whole.

We chose to simplify storage management by dividing the available space in a virtual disk into evenly sized *blocks*. One of our layers, the block API, is exclusively task with the reading and writing of these blocks to and from “raw” storage space. The boundaries of the block layer define the two other layers: the file system layer, implemented on top of the block API, and the I/O layer, which manages the actual storage (reading/writing from the host file system)

2.1.1 I/O layer

On this level we store raw data. Implementations of the I/O layer implement the `IPersistenceSpace` and `IRawPersistenceSpace` interfaces in the `Panda.Core.IO` namespace. Our main implementation uses *memory mapped files* to read from and write to the disk. From the perspective of the block layer, a *storage space* is simply a large block of main memory that gets “magically” persisted to disk. The I/O abstraction is not perfect, as it hands out raw (unsafe) pointers and therefore must be backed by contiguous memory.

For debugging and unit testing purposes, we also implemented a persistence space that is not backed by a file on disk. This has proven useful as we did not have to deal with the creation and clean up of temporary files.

2.1.2 Block layer

The purpose of the block layer is to present a structured view of the raw storage space below. It divides the underlying space into evenly sized block (block size can be configured per disk), which it uses to store both file system meta information as well as the actual file contents themselves.

Blocks are addressed by their index (how many blocks come before them) as opposed to their absolute byte offset. This addressing has the advantage that it is independent of the underlying block size.

The block layer is exposed via the interface `Panda.Core.Blocks.IBlockManager`. Implementations of this interface hand out instances of `Panda.Core.Blocks.IBlock` (or derived interfaces), which are each responsible for a single block of the virtual disk.

Given such an `IBlock`, the user can read/write fields of the block in a safe manner. Pointer arithmetic and serialization/deserialization of data is handled by the block layer and thus completely hidden from upper layers.

Generally, the block layer only manipulates a single block at a time. Operations that span multiple blocks need to be coordinated on a higher level. The one exception to this rule is empty space management. Block managers are expected to keep track of empty blocks (blocks that are ready for allocation) and they generally use blocks themselves to store this information.

As with the I/O layer, we created two implementations of the block layer. Our actual virtual disks use the `Panda.Core.IO.RawBlockManager`, which is backed by a `IRawPersistenceSpace` (in memory or based on a memory mapped file). That implementation persists blocks into the unmanaged memory provided by the raw block manager.

The second implementation, `Panda.Test.InMemory.Blocks`, simply models blocks as a collection plain old C# objects. Again, this second implementation proved very useful, because we were able to test the implementation of the upper layer before our `RawBlockManager` was ready for prime time.

2.1.3 Filesystem layer

This is the most abstract level. Here we implement the most of the actual functionality of the file system in terms of the block API.

This layer needs to worry about how the many different blocks relate to one another. For instance, if the block that is backing a large directory has not enough room for another directory entry, the file system is responsible for allocating an overflow or "continuation" block and spill the new entry there.

When implementing the file system API, we chose to follow the composite pattern. We have `VirtualFile` as the leaf and `VirtualDirectory` as the inner node of our virtual directory tree (both inherit from `VirtualNode`). Common operations like move, rename, delete are of course available on the `VirtualNode`-level.

The API that users of the Core consume is located in the `Panda` namespace whereas our implementation can be found in the `Panda.Core.Internal` namespace.

The entire system is wired up (instantiating persistence spaces, supplying them to block managers and injecting those into virtual disk implementations) by a pair of factory methods in `VirtualDisk`.

3 VFS Core Requirements

Each requirement which is listed is also implemented. The software elements which are involved can be found in the test class Panda.Test/Integration/Specification.cs. The test method is noted to each requirement.

1. *The virtual disk must be stored in a single file in the working directory in the host file system.*
   The whole VFS is stored in the instance disk which is created at the beginning.

While our system defaults to creating disks in the process's working directory, it is not limited to that directory. All information about a virtual disk is stored as part of that virtual disk. As a consequence, we also do not have a central list of existing virtual disks.
The actual storage is handled by `Panda.Core.IO.MemoryMapped.MemoryMappedFileSpace` whereas the file format is implemented by `Panda.Core.IO. RawBlockManager`.
Test method: Req2_1_1_and_2

2. *VFS must support the creation of a new disk with the specified maximum size at the specified location in the host file system.*
   Possible since we can create a VFS.
   Our implementation currently stores the maximum disk capacity as part of the disk, but also pre-allocates the virtual disk file to that size. Test method: Req2_1_1_and_2

3. *VFS must support several virtual disks in the host file system.*
   We can create any number of different disks each with its own file.
   Disks are completely isolated from one another. Exchanging files and directories means exporting from one disk and importing into the other.
   Test method: Req2_1_3

4. *VFS must support disposing of the virtual disk.*
   Possible with Dispose().
   Test method: Req2_1_4

5. *VFS must support creating/deleting/renaming directories and files.*
   Files and directories support all these methods after they are created.
   See `VirtualFileImpl` and `VirtualDirectoryImpl`.
   Test method: Req2_1_5

6. *VFS must support navigation: listing of files and folders, and going to a location expressed by a concrete path.*
   Since files and folders inherit from nodes they are stored in an list and we can loop trough them. With the method `Navigate()` on `VirtualDirectory` (implemented in `VirtualDirectoryImpl`) we can go to any concrete path, absolute or relative.
   Test method: Req2_1_6

7. *VFS must support moving/copying directories and files, including hierarchy.*
   `Move()` and `Copy()` are implemented as methods on `VirtualNode`.
   Test method: Req2_1_7

8. *VFS must support importing files and directories from the host file system.*
   `Import()` is implemented on `VirtualDirectoryImpl`. It imports host directories recursively and uses our `System.IO.Stream`-based `CreateFile` method to import files.
   Test method: Req2_1_8_and_9

9. *VFS must support exporting files and directories to the host file system.*
   `Export()` is implemented as a method on `VirtualNode` and works recursively for virtual directories and via our own (read-only) `Stream` for virtual files.
   Test method: Req2_1_8_and_9

10. *VFS must support querying of free/occupied space in the virtual disk.*
    Size is implemented as by recursively querying the individual directories. To get the occupied space just ask root.
    Test method: Req2_1_10

11. **Bonus Basic:** *Elastic disk: Virtual disk can dynamically grow or shrink, depending on its occupied space.*
    The internal data structure starts out with just the minimal space necessary and then grows incrementally. Section 4.3 discusses how keep track of how much space our file system actually needs. Note that we make use of sparse files to implement this features. This means that for most windows APIs, the file appears to be pre-allocated to the entire disk capacity. Accessing the disk file's properties, however, will reveal what the file's "Size on disk" is. Also note that this dynamic growth only applies to "local" disks. Disks downloaded from a server will not benefit from this.

12. **Bonus Advanced:** *Large data: This means, that VFS core can store & operate amount of data, that can't fit to PC RAM (typically, more than 4GB).*
    We use the operating system's virtual memory mechanism to get the relevant pieces of our file system in memory. We were careful not to hold on to data structures (blocks, virtual nodes) but instead re-create them whenever required. That way, our file system implementation only uses as much RAM as the user wielding it.

    We do, however, require a 64bit virtual address space to operate on large files. While it would be technically possible to map just parts of a large file into the much smaller 32bit address space, we chose not implement re-mapping.

4 Implementation details

This section lists the detailed specification of the block system of the virtual file system.

4.1 General Remarks

- Offsets & lengths in bytes.
- The VFS is organized in blocks with fixed BLOCK_SIZE.
- All addresses are in number of blocks from 0 and of length 4 bytes (unsigned 32bit integer).
- Only single links to blocks (not more than one hard-link) are allowed. This means that one file or directory can only be in one directory.
- Block address 0 is illegal, it means absence of a block.
- B_S := BLOCK_SIZE & d-t := data-type
- Offsets are absolute
- Strings are encoded in UTF-8

4.2 Metadata

Metadata of the whole VFS starts at address 0.

Offset	Length	C# d-t	Description
0	4	UInt32	Number of blocks in entire VFS
4	4	UInt32	BLOCK_SIZE in bytes
8	4	UInt32	Address of root directory node
12	4	UInt32	Address of empty page block. Must never be 0
16	4	UInt32	"break" in number of blocks, see empty space management.
20	4	UInt32	Address of the journal block.
24	8	UInt64	Time of last synchronization.
32	*	string	Name of this disk on the server (prefixed by byte indicating the byte length of this string)

Normal blocks are everywhere but at address 0.

Block Types

- Directory blocks
- File blocks
- Data blocks
- Empty space blocks
- Journal blocks

In addition, for files and directories there are dedicated "continuation" (overflow) blocks for when one block cannot hold all necessary information.

Directory blocks

Contain file / directory names of current directory and their block addresses.

Offset	Length	C# d-t	Description
0	?	-	Arbitrary number of directory entries
B_S - 4	4	UInt32	Link to directory continuation block. 0 here marks absence of continuation blocks.

Directory continuation blocks look the same as directory blocks and can link to other directory continuation blocks.

Directory entry

Offset	Length	C# d-t	Description
0	1	UInt8	Number of bytes in file name. 0 here marks end of directory block.
1	1	UInt8	If first bit (the least significant) set (== 1), following address points to directory. Else to file.
2	4	UInt32	Address to file / directory block
3	X	String	File / directory name

File blocks

Contain addresses to data blocks.

Offset	Length	C# d-t	Description
0	8	UInt64	File size in bytes (to manage file sizes not equal to a multiple of the block size)
8	?	UInt32	Arbitrary number of addresses to data blocks
B_S - 4	4	UInt32	Link to file continuation block. 0 here marks absence of continuation blocks.

File continuation blocks have file size 0 and can link to other file continuation blocks.

File blocks

Contain only plain binary data.

Empty space block

Contains addresses to empty blocks.

Offset	Length	C# d-t	Description
0	4	UInt32	Number of empty blocks in number of blocks
4	?	UInt32	Arbitrary number of addresses to empty blocks
B_S - 4	4	UInt32	Link to empty space continuation block. 0 here marks absence of continuation blocks.

4.3 Empty space management

The VFS is designed to maintain an index of unused blocks. The addresses of the unused blocks are stored in the empty space block. Its address is stored in the VFS meta-data. This empty space block may also have empty space continuation blocks. But not every address to an empty block in the whole VFS can be stored in this empty space block. Instead, only addresses of empty blocks up to a maximum address, which is called “break”, is stored in this block. If there are no empty blocks left, the “break” must be increased by 1, and the new empty block addresses must be added to the empty space block. If the block next to “break” is freed, decrease the “break”, otherwise the address of this block to the empty space block or its last continuation block.

Unfortunately, our organisation of empty blocks does not allow us to reclaim blocks efficiently that have been freed and lie next to the break. This means that our disk only grows dynamically, but remains constant in size when its contents are being deleted.

5 VFS Browser

The VFS Browser is a Graphical User Interface which allows the user to use all functions which are implemented in VFS Core.

5.1 Design

Since our implementation is written in C# we used WPF for our GUI. The interface of our application is fairly straightforward with a menu bar on top and a status bar at the bottom. A tree view (provided by WPF) takes the center stage. It doubles as the list of currently open disks as well as a hierarchical view of each disk’s directory structure. The user can access operations on open disks, files and directories via a context menu on each of these elements.

An additional tab, introduced in the last phase of the project, allows the user to connect to a synchronization server, view a list of disks available on that server and download any of theses disk (if they are not yet present locally).

5.2 Integration

We were able to leave our Core largely intact for it’s integration into the GUI. We can partly attribute this to our use of standard .NET interfaces such as IReadOnlyCollection. This meant that we could simply the WPF tree view at one of our disks and it would recursively bind to our file system hierarchy.

However, in order to update the view in an efficient and fine grained fashion, we decided to implement INotifyPropertyChanged and INotifyCollectionChanged in our Core where appropriate. These interfaces allow any client (not necessarily GUIs, they are not part of a GUI assembly) to receive notifications (event handler callbacks) when a property or collection in our file system API changes its value.

5.3 VFS Browser Requirements

1. *The browser should be implemented on one of the following platforms: desktop, web or mobile.*
   Implemented as an WPF application which runs on a desktop.

2. *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.*
   Possible.
   Most operations are accessible via context menus. Where appropriate, we also provide keyboard short cuts. Users can copy, paste, rename, delete, import and export files and directories. Note that copy and paste is not supported across different disks. If the user wants to transfer a file from one disk to another, they have to export the file or folder from one and then re-import them into the other disk.

3. *The browser should support mouse navigation (or touch in case of the mobile platform). The required operations are the same as in requirement 4.*
   All operations can be initiated with the mouse via context menus or buttons. The file system can be navigated by clicking on the small white arrows next to disks or directories.

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 subfolders.*
   The user can access search via the context menu on directories or disks. A search dialogue will appear that allows the user to specify the search string and whether the search is to be conducted recursively, case-sensitively and/or using regular expressions. The system displays the search results in a list box in the lower half of the search window.

5. **Bonus Basic:** *Responsive UI, i.e. the browser does not stall during long-running operations (i.e. file search or import).*
   Many operations are implemented in an asynchronous fashion (using the async features of C# 5.0). For instance when the user imports a directory the system performs the import in the background and the imported files and directories will appear in the directory hierarchy before the user's eyes.

5.4 Implementation details

We tried to separate browser logic out of the view as much as possible by using a Model-View-ViewModel or Model-View-Presenter pattern. The state of our application is largely held in BrowserViewModel and DiskViewModel. The actual view (MainWindow.xaml) communicates with the view model only via two-way property binding. This turned out to be a very convenient way of programming, since our application logic could operate in terms of the domain model (it manipulates virtual disk objects and string properties) whereas WPF will automatically display changes that we make to the view model.

We also made extensive use of the async keyword in C# 5.0, which allowed us to express the synchronization work flow in a very succinct yet completely non-blocking way. The user can see the progress of their up- or download in the status bar at the bottom and still interact with the application.

6 Synchronization Server

The synchronization server allows users to keep a disk synchronized across multiple machines. The user can choose to associate any of their disks with the server, uploading a copy of that disk to the server in the process. After that, the system can keep the server's and client's disk in sync by only exchanging the parts of the disk that have actually changed. The user can also have their changes replicated to linked disks on other machines or revert a disk with diverging changes to the state of the authorative copy on the server.

6.1 Requirements

Note: Since team Panda fell apart on the way to the final milestone, me and Julian Tschannen agreed to reduce the scope of the final milestone. We decided to drop user authentication and authorisation, focusing on the more interesting aspect of synchronizing disk state instead.

6.3.1.1 *dropped*

6.3.1.2 *The browser should offer to switch to an offline mode, and be able to operate without a connection to the server.*
The default behaviour of disks is to record all changes. The user decides when to synchronize changes. To initiate synchronization, the user opens the context menu of a disk and selects the "Synchronise disk with server" option.

6.3.1.3 *The browser should support binding an existing virtual disk to [the server].*
The user can associate any locally created disk with the server. To do so, the user selects the "Associate with server" option from a disk's context menu. The system will then check with the server to see if that disk already exists on the server. If that is not the case, the browser will upload the disk to the server and then perform the association. Servers don't have an identity, so technically a disk can be synchronized with multiple servers.

6.3.2.1 *dropped*

6.3.2.2 *The server should track changes to linked virtual disks [...] and synchronize the changes across the machines.*
Clients can query our server for a list of changes made to a disc since a client-specified point in time. With this list in hand, they can then have the server transfer only those parts (the parts that changed). It is not the server, however, that initiates synchronization, but the clients. The server simply serves as a central authoritative copy of the disk.

Reverting conflicting changes When our implementation detects a conflict, our client will make sure that their disks become consistent with the server's version. This means that in addition to requesting the changes that other clients have made to the disk, our client also orders the server to send it the original versions of blocks that it changed, effectively reverting the local disk to the last synchronization before incorporating the new changes. This feature is limited to conflict resolution on synchronization. It cannot be triggered manually.

6.2 Design

We decided to perform our synchronization on the level of disk blocks. Our system keeps track of the offset of each block that has been changed as well as when it was changed last. We call the list that tracks these changes our "journal". Like the list of empty blocks, it is implemented as a singly linked list of blocks ("journal blocks") as part of the disk it describes.

The job of this journal is to answer one simple question: "Which blocks have changed since time *X*". The answer comes in the form of a set of block offsets that can then be used to send or receive these changes to and from the server. Together with a mechanism that allows reading from and writing to individual blocks, synchronization of changes is pretty straightforward.

6.2.1 Communication

We decided to implement the synchronization server as a RESTful web service. This decision was primarily motivated by the fact that basic HTTP requests are easier to debug and issue “by hand” than SOAP or custom exchange protocols. To help us we used ServiceStack, a relatively young web service framework that has the advantage that servers built with it can easily be hosted in a standalone application.

ServiceStack’s programming model revolves around DTOs (data transfer objects). Each DTO, a distinct C# type, represents one kind of message understood by the server. Request DTOs can be bound to URLs and have one or more handler methods associated with them, for instance one per HTTP verb. So you could have that DTO type represent your REST resource and perform different actions depending on the HTTP method used in the various requests. Our server actions are implemented in the Panda.Server.ServiceInterface.DiskService class.

ServiceStack also comes with client-side abstractions which will automatically assemble the correct request URL depending on what kind of request object they are supplied with. For this to work, both server and client need the same definitions of these DTOs. That is why we have a separate assembly called Panda.ServiceModel, which provides the common DTOs to both the server and the client. These are completely “dumb” objects with no application logic. Combined with the ServiceStack’s client’s support for asynchronous operations meant that communication was actually rather straightforward to implement. However, ServiceStack is still pretty new and it’s documentation is lacking in places, making it’s use an adventure in of itself.

6.2.2 Global journal

Like with distributed version control systems, our journal is the exact same on all synchronized machines. While this isn’t technically necessary for clients in our solution, it is vital that the server has a change journal reaching all the way back to the initial upload of the disk. The server cannot know how many copies of the virtual disk are floating around and at what stage (“revision”) they are. After all these virtual disks are just file that the user can copy at will. So in order to synchronize with all copies of a disk, regardless of how old the disk is, the server needs to keep the entire journal.

So could we have dropped unnecessary parts of the journal from the clients? Not while the journal is part of the disk itself, no. While the client could function with a journal that only reaches back to the last time it synchronized, the missing journal blocks would still have to be reserved.

This is because the journal really tracks changes to *all* blocks in disk, including the journal blocks and the empty block list blocks. So if the client were able to re-allocate no-longer-needed journal block, it would quickly diverge from the server.

One advantage that client-side global journals have, is that every copy of the disk can serve as a new starting point for a server copy. So if your server dies with a catastrophic hard drive failure, you can re-upload any copy of the synchronized disks without losing compatibility with older versions of the disk.

6.2.3 Duplicate entries

As the journal keeps track of *all* blocks changed on a disk and is itself located on the disk, it also keeps track of itself. So every time a change to any block on a disk happens, the journal actually writes two entries, one for the changed block and one for the changed journal? No, at runtime, the block manager keeps track of blocks that have been changed since the last synchronization in a hash set. Repeated changes to a single block without an intermittent synchronization are not re-recorded. However, this data structure is *not* stored on the disk. So if the browser is shut down, it forgets about the changes that have already been recorded since the last synchronization.

Another tricky aspect of journal keeping is the space for the journal blocks themselves. When the current journal page runs out of space, the system allocates a new one (likely changing the empty block list) and updates the pointer in the meta data block (to point to the new "head" of the journal list), incurring yet another change.

The journal keeping algorithm is implemented in OnBlockChanged in class RawBlockManager.

6.3 Integration

Unfortunately, our synchronization works at a lower level of abstraction (blocks) than our file system API (disks, files and directories). This meant that we had to "punch a hole" through the file system API in order to operate our synchronization mechanism. We did this by implementing the newly created interface ISynchronizingDisk. We have not encapsulated the synchronization logic as part of the VFS Core. Most of the synchronization logic is located in Synchronize in class BrowserViewModel.

The interface that the user would use to manipulate the disk has not been affected by this change. One downside of this approach is that the client cannot tell in advance whether a disk supports synchronization as that is a property of the block manager underlying the disk. Our mock implementation of IO, against which we unit test our file system layer — the MemBlockManager — for instance does not support synchronization, because it is not based on blocks.

7 Quick Start Guide

The server, called Panda.Server, and the VFS browser, called Panda.UI are separate assemblies. The respective bin folders contain all their dependencies (apart from the .NET framework 4.5).

By default, the VFS browser ("client") operates on files located in the process working directory. If you want to run multiple client instances on the same machine, either create multiple copies of the client binaries in different locations or make sure that the client instances are run with different working directories.

7.1 Server setup

The server uses HttpListener, which takes advantage of communication infrastructure built into Windows XP SP3, and Vista upwards. Depending on how your system is configured (user account privileges) you might need to some preparatory work.

In the simplest case, you can just double-click/run Panda.Server to start a server on port 8997. Pressing any key will shut the server down.

If you want to run the server on another port, you can pass the server's base URL as a command line parameter:

```
> Panda.Server.exe http://*:9080/
# or
> Panda.Server.exe http://*:80/pandas/
```

The * instructs the HttpListener to bind to all network interfaces. You can optionally specify a service prefix after the port number, but you need to make sure that all clients also use that same prefix. *Important*: the service base URL *must* end with a forward slash (/).

7.1.1 Troubleshooting

Depending on your system configuration, you might need to first allow the server to bind to that address. You can do this via the netsh from privileged (Administrator) command line. Type:

```
> netsh http add urlacl url=<the-url-to-allow> \
              user=<your-username> listen=yes
```

(The server detects this scenario and provide the concrete command line for you to type)

If you want the server to accessible from other machines, you might have to add a corresponding exception to the windows firewall. One way to do this, is to run the following command, again on an elevated command line:

```
> netsh advfirewall firewall add rule \
     name="Panda.Server" dir=in action=allow \
     protocol=TCP localport=<the-port-to-open>
```

and later delete the firewall exception via

```
> netsh advfirewall firewall delete rule \
             name="Panda.Server"
```

7.2 Example interaction

1. Start the server, for instance bound to http://*:8997/
2. Create two copies of the client binaries in two different directories (or run it from two different working directories).
3. Launch both clients, we'll call them client A and client B.
4. In client A, choose "New Disk..." from the "File" menu in the upper right corner (or press Ctrl+N).
5. Choose a maximum capacity and a disk name. We'll leave it at 10MB and call our disk "disk1".
6. Click "Ok". You can now see your empty disk in the "Browser" view.
7. Switch to the "Server" view by clicking on the "Server" tab at the top.
8. Enter the base URL of the server (it defaults to http://localhost:8997/, which is exactly what we need)
9. Click the "Connect to server" button below
10. Switch back to the "Browser" view by clicking on the "Browser" tab above.
11. Right-click the disk you created before, "disk1" and click "Associate with server". The disk is now being uploaded to the server. This only happens the first time you associate a disk with a server.

12. Switch to client B.
13. In client B, switch to the “Server” view.
14. Connect to the server by clicking the “Connect to server” button. Your disk, “disk1” will appear in the table below.
15. Right-click on “disk1” and click “Download disk”. This creates an initial copy of the disk for client B. This only happens if you don’t already have an associated copy of that disk for that client.
16. In client B, switch to the “Browser” view. You’ll see “disk1” also appear here.
17. Switch back to client A.
18. In client A, right-click on “disk1” and click on “Import”.
19. Choose a file or folder to import (automatically happens recursively). Make sure that your disk’s capacity is large enough for the folder.
20. Still in client, right-click on “disk1” and click “Synchronize disk with server”. Wait until the disk has been synchronized.
21. Close client A. If you want, you can also delete the disk1.panda file in the working directory of client A.
22. Switch to client B.
23. In the “Browser” view , right-click “disk1” and click “Synchronize disk with server”. Wait until synchronization is complete.
24. The directory you imported in client A is now in your local copy of disk1. Click on the white arrow next to your disk to reveal it.
25. Right-click on your directory and select “Export”.
26. Choose where the client should export your directory to.
27. Close client B.
28. Press any key on the server to shut it down.

8 References and acknowledgements

Panda.Core/Core/Internal/Extensions.cs A collection of helper methods that I, Christian Klauser, have written and use for many of my C# projects. On the web:
https://github.com/SealedSun/prx/blob/master/Prexonite/Helper/Extensions.cs

EditableTextBox The editable text box custom control (used to make renaming files and directories work inline, like in Explorer) was taken from a CodeProject contribution:
http://www.codeproject.com/Articles/31592/Editable-TextBlock-in-WPF-for-In-place-Editing

Sparse files The C# bindings for the NTFS sparse files API was taken from a Microsoft provided example called CSSparseFile
http://code.msdn.microsoft.com/windowsdesktop/CSSparseFile-6e26dc97

Panda application icon Taken from http://www.visualpharm.com/ in exchange for this link (licensed as "Linkware")