Transactional File in Java

USER GUIDE

author: Ioannis Ganotis
email: Ioannis.Ganotis@ncl.ac.uk
Newcastle upon Tyne,
JBoss, a division of Red Hat

Index

1 Introduction

1.1 About this Guide

This Guide describes a library that can be used to support transactions in File I/O in Java. It provides information on how an application programmer can use the classes included in the library to benefit from transactional support when applying operations on the contents of files or directories. It also describes how the operations can be called in a distributed environment (e.g. different Virtual Machines) without violating any of the ACID (Atomicity, Consistency, Isolation, Durability) properties.

1.2 Acknowledgements

Jonathan Halliday (jonathan.halliday@redhat.com) has contributed providing truly inspiring dedication, guidance and knowledge to the development of the “Transactional File in Java” project.

1.3 Prerequisites

This Guide assumes familiarity with Object-Oriented (OO) programming (preferably Java), basic knowledge on Transactions (Transaction Managers) and a basic understanding of multi-threaded applications. A general understanding of UNIX operating system will be also useful.

1.4 Documentation

(add link/references here...)

2 Transactional File in Java

2.1 Brief Description

Transactions are used in software to increase accuracy and reliability. They perform as indivisible, complete units of work to access and modify critical business information. It is really important to ensure that no data loss will happen while modifying such critical information. From the scope of programming, what an application programmer normally does, is to call start and end operations on Transaction Managers (TMs) and then include his business logic within the transaction boundaries (begin – commit/rollback). The business logic may involve operations like accessing transactional resources such as Databases and Messaging systems. However, replacing a transactional resource with a filesystem will not give the expected behaviour. Current versions of Java do not support transactions in their File I/O library. This means that if, for example, an application programmer uses a TM and within the boundaries of a transaction invokes operations to modify the bytes of a file, the updates will be written to disk no matter if the transaction has committed yet or not. This of course is not the desired behaviour and if such an application was used by a big organisation (e.g. bank) modifying bytes in a file while an error in the application had previously occurred, would rather end up with catastrophic results for the organisation and customers complaining or, be happy for the rest of their lives.

At the moment, programmers that need to avoid the previously described behaviour, have to implement transactional support in their code. This costs in terms of time needed for the implementation and also is a bad practice as the application programmer mixes-up his business logic with transactions. As an approach to this problem, the Transactional File I/O in Java project has been proposed. It provides a library specially “tailored” to application programmers that wish to have access to a filesystem with full coverage on ACID semantics. Like in the examples that follow in this Guide, the library provides an implementation that allows application programmers to manipulate both contents of files and directories transactionally.

Another great advantage offered by the library on both approaches (files and directories) is the recovery support. Even if the system crashes due to some failure in the software or hardware, after successful restore the library is more than able to reconstruct its state just before the system's failure and continue any incomplete tasks.

2.2 How to Build and Install

The library, as mentioned earlier, can be used for both managing contents of files and directories. There are two different packages for this: the *txfiles*: and *txdirs*. This gives the ability to application programmers to compile and use only the package they wish. It would also be a problem compiling the whole library for those who want to use only the *txfiles*, as the *txdirs* requires an extra installation of the Apache commons transaction project.

Requirements for *txfiles*:

- JDK – at least version 1.5 (http://java.sun.com/javase/)
- Ant build tool – at least version 1.7.0 (http://ant.apache.org/)
- JBossTS 4.4.CR1 or later (http://jboss.org/jbosstm/)

Requirements for *txdirs*:

- All the requirements of *txfiles*
- Apache Commons Transaction project – version 1.2
  (http://commons.apache.org/transaction/)

In order to compile successfully, the provided "build.xml" file must be edited to include the home directories of JBoss and Apache Commons Transaction (only if the *txdirs* is to be compiled). To do this, the two properties on the top of the build file must be modified accordingly.

Example

```
<property name="jbossts.home" value="/home/ioannis/JBoss/install"/>
<property name="apache.home" value="/home/ioannis/Apache/commons-tx"/>
```

The *build* file is now ready to be used with the *ant* tool. There are several target-options available for either compiling the whole library or only one of its packages (*txfiles* or *txdirs*) depending on the future usage of the library. There is also a "jar" target which will package the compiled code in a *jar* file under a *build* directory. The *jar* file can then be included in other projects that need the support of this library.

2.3 Running the Demo

No matter which of the two packages you use, you will find a *demo* package with a runnable class in each of them. There is an ant target to run each of these. An output in each case will be produced to demonstrate the transactional behaviour.

Example

```
[ioannis@localhost ~]$ ant txfiles-demo
```

3 Available API and Examples

3.1 Part I: Manage Contents of a File Transactionally

Operations that will allow managing the contents of file are *read/write* operations that can be invoked through the XAFile class. This class implements *DataInput* and *DataOutput* interfaces and uses a *RandomAccessFile* object for reading and writing to a file. Generally speaking the API of this class, regarding file operations (e.g. read/write), is an implementation of all the methods included in the two interfaces mentioned above, plus other operations available by a *RandomAccessFile* object.

3.1.1 Create / Open a File

In order to open a file, an object of the XAFile class must be instantiated. If the file does not exist, the XAFile constructor automatically creates and opens a new one. Although the XAFile has been designed and developed with the aim to behave transactionally, the constructor allows the creation of objects that can be used to read or write directly to files without transaction support. This is quite similar to using a RandomAccessFile object to access the contents of a file. The constructor is of the following form:

```
public XAFile(String filename,
              String mode,
              boolean transactionsEnabled)
       throws IOException
```

Figure-1.0

The first parameter in the constructor is the name (including path) of the file that contains the business data entries. The second parameter specifies the mode in which the file must be opened and it can be one of the modes used in the *RandomAccessFile* class. The most usual ones are "*r*" or "*rw*" which opens a file only for reading or for both reading and writing to it, respectively. Like mentioned before the XAFile can be used in two ways: transactionally or not. This is specified by the third parameter which is of type boolean. If it is true it means that transaction support on this file is enabled.

Example

There are three different ways to instantiate XAFile objects.

```
1. XAFile xaFile = new XAFile("entries.txt", "rw", false);

2. DataInput in = new XAFile("entries.txt", "r", false);

3. DataOutput out = new XAFile("entries.txt", "rw", false);
```

Figure-1.1

Obviously, using one of the two interfaces does not give access to methods needed for transaction support. Even if the transactions are enabled (3rd parameter is set to *true*) transactional behaviour will not work and an IOException will be thrown instead.

3.1.2 Invoke File Operations (Transaction Support: Disabled)

After creating the instance, operations such as *read* and *write* can be applied to the file. There is also a file pointer which progresses according to *read* and *write* operations or can be set manually using the *seek* method provided by the XAFile class. An example of this (using the first way to instantiate an XAFile object) would look like:

Example

```
public static void main(String[] args) throws Exception {
  XAFile xaFile = new XAFile("entries.txt", "rw", false);
  xaFile.writeInt(1);           // write Integer(1); position: 1
  xaFile.seek(0);               // reset position to: 0
  int result = xaFile.readInt();// read Integer; position: 1
  System.out.println("Read Integer= " + result);
  // other code...
}
```

Figure-1.2

This will open the "*entries.txt*" file in "*rw*" access mode so both reading and writing can be applied. Transaction support is set to be disabled. At the time of creation of the *xaFile* the position of the file's pointer is set to *0* which points to the start of the file. A *write* operation is then performed through that *xaFile* object and a value of type *Integer* is written directly to the file. The *seek* method will move the file's pointer to the beginning of the file so as a *read* operation can then read the previously written value. The returned *result* in the above example will be the number 1.

3.1.3 Invoke File Operations (Transaction Support: Enabled)

As mentioned before, to make *XAFile* behave transactionally the boolean in its constructor must be set to *true*. In addition, a method that will create a new Resource Manager (RM) and will associate it with both a TM and the current thread must be invoked through the *xaFile o*bject. A TM object is needed here in order to have access to a transaction on which the RM will be enlisted (transparently to the application programmer).

Example

```
public static void main(String[] args) throws Exception {
  XAFile xaFile = new XAFile("entries.txt", "rw", true);
  TransactionManager txMngr = new TransactionManagerImple();
```

```
  txMngr.begin();
  { // business logic - start
    xaFile.newTransaction(txMngr);
    xaFile.writeInt(1);         // write Integer(1); position: 1
    xaFile.seek(0);             // reset position to: 0
    int result = xaFile.readInt();// read Integer; position: 1
    System.out.println("Read Integer= " + result);
  } // business logic - end
  txMngr.commit();
  // other code...
}
```

Figure-1.3

The code above demonstrates a simple case with file operations aiming to access the file the same way the code in *Figure-1.2* does, but also transactionally. Even if the file operations between the two examples look exactly the same the overall behaviour and the outcome is different.
As you see, the boundaries of the transaction are denoted by the *txMngr.begin()* and *txMngr.commit()* statements. Within these boundaries the application programmer's business logic is written. An instance of TM is passed as a parameter to the *newTransaction()* provided by the *xaFile* object. What this method actually does, is to create a new XAResourceManager (XAREM), get the transaction object from the TM (passed as parameter) and enlist the XAREM object to the retrieved transaction. It will also associate the XAREM with the current thread id, but the reason of this will be explained later on. The program in *Figure-1.3* will then try to apply some file operations with the first one being to write the Integer value *1* to the file. As the file operation is within the boundaries of a transaction, the value will not be written to the file yet. Instead, it will be kept in memory and log files (durable storage in case of system crash). Any read operation within the same transaction will return the modified bytes (if they exist) instead of the actual bytes in the file. Consider a scenario where an Integer of value “2” is already written at position 0 in the file used in the example above. The *read* operation used after having the Integer *1* written within the transaction will return the value *1*. Now if somebody else, outside of this transaction tries to read an Integer at position *0* in the file while the transaction is still in progress, he will get the actual value (an Integer number “2”) rather than the modified one (Integer value “1”). If no errors appear and the *commit* statement is reached, any modifications happened within the transaction will be written to the file so anybody else can see them. If the *commit* statement is not reached for some reason the TM should be informed about this, so the following model is encouraged to be used:

Example

```
public static void main(String[] args) throws Exception {
  XAFile xaFile = new XAFile("entries.txt", "rw", true);
  TransactionManager txMngr = new TransactionManagerImple();
```

```
  try {
    txMngr.begin();
    {
      xaFile.newTransaction(txMngr);
      // do business work here...
    }
    txMngr.commit();
  } catch (Exception e) {
    txMngr.rollback();
  }
  // other code...
}
```

Figure-1.4

With this model if any error occurs while reading or writing to the file, the thrown exception will be caught and the TM will rollback any changes made so far.

The application programmer must be careful here when invoking the *newTransaction()* method. As in *Figure-1.3* the method is invoked as the first statement within the boundaries of the transaction and before any *read*/*write* operations. If the method is called in the wrong place it will be reported to the application programmer by throwing a relevant exception. Importantly, the *newTransaction()* method cannot be called twice by the same thread as in the background each thread's id is associated with an XAREM object.

3.1.4 Multiple Concurrent Transactions

In this part, when managing contents of a file transactionally it is possible to have multiple concurrent transactions updating parts on the same file. This concurrency can be achieved by creating multiple threads Each thread will have its own TM associated with it and obviously its own *read/write* operations.

Example

```
final XAFile xaFile = new XAFile("entries.txt", "rw", true);

// thread-code bellow
Thread th = new Thread(new Runnable () {
  public void run () {
    try {
      TransactionManager txMngr = new TransactionManagerImple();
      try {
        txMngr.begin();
        {
          xaFile.newTransaction(txMngr);
          // do business work here...
        }
        txMngr.commit();
      } catch (Exception e) {
        txMngr.rollback();
```

```
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
});
th.start();
```

Figure-1.5

The “thread” code in the above example can be repeated to meet the number of threads that want to access the file simultaneously. When using multiple threads interferences may appear while accessing the same parts of the file. It is not necessary for the application programmer to build an application that will avoid such interferences. The locking mechanism, also included in the library, will take care of this and if access is denied while processing a part of the file due to some lock conflict, it will be reported by throwing a build-in *LockRefusedException*. The application programmer can detect such exceptions and retry the file operations until the locks are released. Although the library will protect concurrent access on the same data to avoid inconsistency issues, it may lead to reduced concurrency control. So, it would be of better performance if the application programmer could provide a model by which interferences could be avoided to the maximum.

3.1.5 Multiple VMs - Concurrent Transactions

A file can be managed transactionally using different VMs. All of the VMs have a shared space to persist or retrieve locks. The idea is to let every VM know which parts of a file are currently managed by other VMs. There is nothing special for the application programmer to do or to take care of in this case as everything happens transparently. The only issue again is the concurrency control which will be reduced if many conflicts appear while the VMs are accessing the same data.

3.1.6 Close the File

Like in every Input or Output class in Java's I/O library there is a *close()* method which closes the stream used for reading/writing and releases any system resources associated with it. Similarly, invoking the *close()* operation provided by the *XAFile* class will close the stream (actually the *RandomAccessFile* in this case) and will remove unnecessary files created by the library during the execution of transactions. If the method is called while a transaction is still in progress it will cause an IOException to be thrown complaining about incomplete transactions. So the application programmer must ensure that all the transactions have either committed or rolled back and then close the *XAFile.*

Example

```
XAFile xaFile = new XAFile("entries.txt", "rw", true);
// do work with xaFile here...
xaFile.close();
```

Figure-1.6

3.2 Part II: Manage Contents of a Directory Transactionally

This is quite similar to *Part I* but deals with manipulation of directory contents rather than contents of files. The *XADir* class represents the directory the contents of which are going to be modified. Its contents are represented by *XADirFile* objects each of which provides operations like renaming, deleting and creating a new file. Unlike *Part I*, the use of multiple VMs will result in unexpected behaviour as new instances of *XADir* will try to remove pre-existing transactions. However, you may use concurrent transactions to manage contents of a directory as soon as they do not interfere. If they do, the library will throw an exception to report the existence of conflicts.

3.2.1 Create / Open a Directory

The first step is to create an *XADir* object. The already provided constructor is of the following form:

```
public XADir(java.io.File storeDir)
    throws java.io.IOException
```

Figure-1.7

The *storeDir* parameter is passed by the application programmer and must point to a directory the contents of which are to be managed transactionally. If the *File* passed by the user does not represent a directory an appropriate exception will be thrown. Also if the *File* does not exist, the constructor will create a new one.

Example

```
XADir xadir = new XADir(new File("/home/ioannis/businesstxdir"));
```

Figure-1.8

Suppose the transactional directory we want to work with is called “businesstxdir”. Also assume the directory is initially empty. As soon as the directory contains no files, the only possible operation is to create a new file. Like in *Part I* above, in order to benefit from transaction support a TM is needed to define the boundaries of a transaction.

Example

```
XADir xadir = new XADir(new File("/home/ioannis/businesstxdir"));
TransactionManager txMngr = new TransactionManagerImple();

txMngr.begin();
{
```

```
    xadir.startTransactionOn(txMngr);
    // business logic (e.g. create, delete, rename files)
  }
  txMngr.commit();
```

Figure-1.9

There is also an additional statement which needs to appear within the scope of a transaction and right after the TM has begun. The *startTransactionOn()* method needs to be called through the *xadir* object. This method gets a TM object as a parameter and uses it to retrieve a transaction. A new XAREM will be created and enlisted to the transaction.

3.2.2 Invoke Operations on Directory's Files

To create a new file a construction of an *XADirFile* object is necessary. The constructor of this class is of the form:

```
public XADirFile(java.io.File file,
                 XADir xadir)
```

The first parameter will take the *File* instance which contains the name of the new file. The second is the *xadir* object created earlier. This is to specify that we want to create a new *file* under the *xadir* directory.

Example

```
XADir xadir = new XADir(new File("/home/ioannis/businesstxdir"));
TransactionManager txMngr = new TransactionManagerImple();

txMngr.begin();
{
  xadir.startTransactionOn(txMngr);
  XADirFile newFile = new XADirFile(new File("resource.txt"),
                                                        xadir);
  // business logic (e.g. create, delete, rename files)
  newFile.createNewFile();
}
txMngr.commit();
```

Figure-1.10

The “resource.txt” file will only be created to disk after the *commit* statement has been reached. Now lets assume that the *xadir* contains a number of files. To get the files from the directory the *listTXFiles()* method can be used. This method will return an array of *XADirFile* objects which represent the files in the *xadir*. Operations are exposed to the application programmer through each of the objects in the array. Consider a case where the programmer wants to apply an operation in one of the existing files in the *xadir*, for example rename the first of the files in the array list. The example below demonstrates how it can be achieved:

Example

```
XADir xadir = new XADir(new File("/home/ioannis/businesstxdir"));
TransactionManager txMngr = new TransactionManagerImple();

txMngr.begin();
{
  xadir.startTransactionOn(txMngr);
  // business logic (e.g. create, delete, rename files)
  XADirFile[] files = xadir.listTXFiles();
  files[0].renameTo(new File("new_name.txt"));
}
txMngr.commit();
```

Figure-1.11

Again, the first file in the array list will only be affected if the TM has committed. Operations other than *renameTo()* can be applied on existing files.

3.2.3 Multiple Concurrent Transactions

Like in the first part, it is possible to set up a number of threads each of which is associated with a new transaction. The transactions can then perform operations on the contents of the directory which will succeed only upon successful completion of the transactions.

Example

```
final XADir xadir = new XADir(new
                            File("/home/ioannis/businesstxdir"));
TransactionManager txMngr = new TransactionManagerImple();

// thread-code bellow
Thread th = new Thread(new Runnable () {
  public void run () {
    try {
       TransactionManager txMngr = new TransactionManagerImple();
      try {
        txMngr.begin();
        {
          xadir.startTransactionOn(txMngr);
          // business logic (e.g. create, delete, rename files)
        }
        txMngr.commit();
      } catch (Exception e) {
        txMngr.rollback();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
});
```

Figure-1.12

The “thread” code in *Figure-1.12* can be repeated depending on the number of concurrent operations the application programmer wants to perform on the files of the directory. Using multiple threads is good as tasks can be completed quicker. A Locking mechanism will ensure that access to the same data will not cause any inconsistencies. However, a repeated number of lock conflicts while accessing the same file may result in bad concurrency control. It is encouraged that the application programmer will provide a sufficiently good model to avoid as many conflicts as possible.

3.2.4 Close the Directory

Importantly, the application programmer must invoke the *close()* method when all the transaction work and work related to the directory is complete. This will remove any temporary folders used during the execution of the transaction to keep “shadow” files and logs that are not needed any more.

Example

```
XADir xadir = new XADir(new File("/home/ioannis/businesstxdir"));
// do work with xadir here...
xadir.close();
```

Figure-1.13

3.3 Failure Recovery

As there is no absolute guarantee for today's systems that their software or hardware will never fail, ability to recover from failures gives a powerful advantage to this library. Both of the parts described earlier use XAREMs that implement Java's standard XAResource interface as well as the Serializable interface so their state can be stored and restored. In case of a system failure a Recovery Manager (RECM) is needed to help both TMs and XAREMs to reconstruct their state just before the system crash. The following lines of code present such a RECM which uses a thread to detect interrupted transactions. The following code must also be placed before doing any other job, at the very beginning of the application.

```
RecoveryManager rm = RecoveryManager.manager();
rm.startRecoveryManagerThread();
```

Figure-1.14

Alternatively, you can use the build-in *RecoverManager* class under the *recovery* package of this library.

4 API Overview

Below are the methods available to the application programmer. For more information and detailed description on the methods below please refer to the relative documentation mentioned in the Introduction section of this Guide.

4.1 Part I: Transactional File Methods

void	**close**()	Closes the XAFile
void	**flush**()	Forces data to be written to disk by instantly closing and re-opening the random access file.
String	**getFilename**()	Returns the name of the file
long	**getFilePointer**()	Returns the current offset in this file.
String	**getMode**()	Returns the access mode in which the `XAFile` was created
RandomAccessFile	**getRAF**()	Returns the `RandomAccessFile` object used to read/write
long	**length**()	Returns the length of this file.
void	**newTransaction**(javax.transaction.TransactionManager txnMngr)	Method to create a new Transaction and enlist XAResources.
int	**read**(byte[] bytes)	Reads up to `bytes.length` bytes of data from this file (or memory if there are uncommitted byte updates) into an array of bytes.
int	**read**(byte[] bytes, int off, int len)	Reads exactly `len` bytes of data from this file (or memory if there are uncommitted byte updates) into an array of bytes.
boolean	**readBoolean**()	Reads a `boolean` from this file.
byte	**readByte**()	Reads a signed eight-bit value from this file (or memory if there are uncommitted byte updates).
char	**readChar**()	Reads a character from this file.
String	**readChars**(char[] chars)	Reads up to `chars.length` characters from this file (or memory if there are uncommitted byte updates) into an array of characters.
int[]	**readDirectlyFromFile**(int len)	Method to read exactly `len` bytes directly from the file and starting at the current file pointer.
double	**readDouble**()	Reads a `double` from this file.
float	**readFloat**()	Reads a `float` from this file (or memory if there are uncommitted byte updates).
void	**readFully**(byte[] bytes)	Reads `bytes.length` bytes from this file(or memory if there are uncommitted updated bytes) into the byte array, starting at the current file pointer.
void	**readFully**(byte[]	Reads exactly `len` bytes from this file(or

	bytes, int off, int len)	memory if there are uncommitted updated bytes) into the byte array, starting at the current file pointer.
int	**readInt**()	Reads a signed 32-bit integer from this file (or memory if there are uncommitted byte updates).
String	**readLine**()	Reads the next line of text from this file.
long	**readLong**()	Reads a signed 64-bit integer from this file.
short	**readShort**()	Reads a signed 16-bit number from this file.
int	**readUnsignedByte**()	Reads an unsigned eight-bit number from this file (or memory if there are uncommitted byte updates).
int	**readUnsignedShort**()	Reads an unsigned 16-bit number from this file.
String	**readUTF**()	Reads in a string from this file (or memory if there are uncommitted byte updates).
void	**seek**(long position)	Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	**setTransactionsEnabled**(boolean transactionsEnabled)	Method to enable Transactional support in the file
int	**skipBytes**(int n)	Attempts to skip over `n` bytes of input discarding the skipped bytes.
void	**write**(byte[] bytes)	Attempts to write `bytes.length` bytes from the specified byte array to this file, starting at current file pointer.
void	**write**(byte[] bytes, int off, int len)	Attempts to write `len` bytes from the specified byte array starting at offset `off` to this file.
void	**write**(int b)	Attempts to write the specified byte to this file.
void	**writeBoolean**(boolean b)	Attempts to write a `boolean` to the file as a one-byte value.
void	**writeByte**(int b)	Attempts to write a `byte` to the file as a one-byte value.
void	**writeBytes**(String bytes)	Attempts to write the string to the file as a sequence of bytes.
void	**writeChar**(int ch)	Attempts to write a `char` to the file as a two-byte value, high byte first.
void	**writeChars**(String s)	Attempts to write a string to the file as a sequence of characters.
void	**writeDouble**(double d)	Converts the double argument to a `long` using the `doubleToLongBits` method in class `Double`, and then attempts to write that `long` value to the file as an eight-byte quantity, high byte first.
void	**writeFloat**(float f)	Converts the float argument to an `int` using the `floatToIntBits` method in class `Float`, and then attempts to write that `int` value to the

		file as a four-byte quantity, high byte first.
void	**writeInt**(int n)	Attempts to write an `int` to the file as four bytes, high byte first.
void	**writeLong**(long l)	Attempts to write a `long` to the file as eight bytes, high byte first.
void	**writeShort**(int s)	Attempts to write a `short` to the file as two bytes, high byte first.
void	**writeUTF**(String str)	Attempts to write a string to the file using modified UTF-8 encoding in a machine-independent manner.

4.2 Part II: Transactional Directory Methods

4.2.1 XADir class

void	**close**()	As this class represents a transactional directory, the application programmer must call this method after his transactional work is over.
long	**length**()	Returns the number of files (not directories) under this transactional directory.
XADirFile[]	**listTXFiles**()	This method lists all the files under the transactional directory.
void	**startTransactionOn**(javax.transaction.TransactionManager txnMngr)	This method must be used after a `TransactionManager` has begun and within the boundaries of a transaction (`begin, commit/rollback`).

4.2.2 XADirFile class

boolean	**createNewFile**()	Create a new file in the disk.
boolean	**delete**()	This method will delete the file.
String	**getName**()	Returns the name of the file.
InputStream	**readResource**()	Returns an InputStream containing the bytes of this `XADirFile` file.
boolean	**renameTo**(java.io.File file)	Renames this file to the name of the file given by `file`.
OutputStream	**writeResource**()	Returns an OutputStream containing the bytes written to this `XADirFile` file.