File Backup Server.

3 copies please

C.A.R. Hoare

October 1981. first draft.

first nowhere published.

Summary: A file backup server is designed to maintain extra copies of files stored on computer-readable media, so that the information can be restored in the event of failure, loss, or corruption of the original. This paper gives an abstract mathematical model of the specification of a file backup server. It then describes the abstract properties of the hardware recommended for its implementation. A complete set of operating instructions is given; and an implementation is designed.

Contents:

1. The Abstract Specification.

A file backup up server is intended to give service to a group of users, each identified by a value from the set

USERNAME.

It is of no concern to the server whether a username is shared among many persons, or whether a single person has many usernames. Informally, we shall regard a username as denoting a single male user.

Each user is the unique owner of a set of files, each identified by a value from the set

FILENAME

The information contained in a file is a value from the set

FILECONTENTS

The filename and the file contents may have a particular internal structure; but this is of no concern to the file backup server at the present abstract stage of its design.

We also requires to record approximately the date and time at which a file is backed up. This is a value from the set

DATETIME,

which is assumed to be totally and strictly ordered by the relation $<$. The unit of increment of the datetime is assumed to be short in comparison with the interval between operations of the backup server.

Each backed up file is indexed by a value from the set

$$\text{FILEINDEX} \triangleq \text{USERNAME} \times \text{FILENAME} \times \text{DATETIME}$$

Two different file contents with the same username and filename but different datetimes are said to be different <u>versions</u> of the same file. In order to prevent indefinite expansion of the stored material, the server

is permitted to purge older versions of files for which newer versions are available. We therefore introduce the function

$$longago: DATETIME \to DATETIME$$

which maps its argument onto a time which is sufficiently long ago to permit automatic purging; clearly

$$longago(t) < t$$

However, a file may not be purged unless there remains available a more recent version of the file, which was also stored long ago. This ensures that a stable version of a file is not lost as soon as a new version is backed up. However, an old file which contents:

$$empty \in FILECONTENTS$$

can be purged freely.

We now define the state of the backup server.

SERVER

store: FILEINDEX → FILECONTENTS;

now, lastpurge: DATETIME;

recent: P(FILEINDEX);

lastpurge ≤ now;

recent = {(u, f, t): FILEINDEX | t > longago (lastpurge)};

card (dom (store) ÷ recent) ≤ 1;

store^{-1}({empty}) − recent = {}

Notes, 'now: is the time of the most recent operation of the backup server.

lastpurge: is the time at which redundant files were most recently discarded.

Operations on the server are of type

OP = SERVER ↔ SERVER'

An important auxiliary operation is one that updates the variable "now" to the current date and time; it strictly increases the value of "now", if necessary, by waiting for the elapse of real time. For this, the implementation will require a "datetime server".

The most frequent operation of the (backup) server is

insertfile: USERNAME × FILENAME × FILECONTENTS

$$\rightarrow OP$$

$$insertfile = \lambda(u,n,c).\{OP \mid now' > now;\ lastpurge' = lastpurge;\ store' = store \oplus [(u,n,now') \rightarrow c]\}$$

A regular operation is

purge: OP;

$$purge = \{OP \mid now' > now \ \&\ lastpurge' = now' \ \&\ store' \subseteq store \ \&$$
$$\forall (u,n,t): (dom(store) - dom(store')).\ store(u,n,t) = empty$$
$$\vee \exists t': (t .. longago(now')).(u,n,t') \in dom(store')\}$$

The retrieval of a file from the backup server is nothing but an inspection of its "store" component. Another useful enquiry is one which gives the directory of files owned by a particular user. This should give the name and date of each version ~~of each version~~ of each file; and it is also convenient to give the length of the file. The directory may be used to check that the backup server has indeed carried out the operations which have been requested of it; it may also be used for accounting purposes.

directory: SERVER × USERNAME
→ TP(FILENAME × DATETIME × NAT)

directory(s, u) = {(n, t, l) | length(s.store(u, n, t)) = l}

There is no need to provide a "deletion" operation — indeed its use would be very risky. The recommended method is to set the file content to empty, so that (actual) deletion is postponed until a later purge.

2. The abstract hardware.

This section serves as a specification of a lower level software package which administers the physical hardware driving the storage media. We shall therefore assume that this package will contains reasonable error checking and recovery or retry procedures for hardware and operator errors; and these need not be replicated by the backup server.

In order to achieve immunity from hardware and software error, a backup server should use storage media (such as tapes or disc) that are physically removable from the processor, and can be stored in a secure remote environment, for example, a fire-proof safe. The unit of removable storage is known as a pack. Each pack must be clearly labelled, both externally and in computer-readable form, by

an element of the set

PACKNAME

The contents of a pack are a sequence of values from the set

BLOCK.

PACKCONTENTS = seq BLOCK.

The value of the complete library of dismounted packs stored away from the computer is a value from the set.

LIBRARY = PACKNAME $\not\to$ PACKCONTENT.

A handler is a device which may be used to read or write the content of a pack which is loaded onto it. The important components of its states are described as follows.

"empty": means there is no pack loaded, in which case none of the remaining variables have relevant values.

pn : is the name of the currently loaded pack.

pc : is the content of the currently loaded pack

rw = "read": means that the pack is loaded for rewinding and reading only.

rw = "write": means that the pack is loaded for writing only.

pos: is the index of the next block to be read or written.

space: if rw = "write", pos + space is a conservative estimate of the (total) number of blocks that can be physically accommodated on the medium.

~~broken: means that the hardware has been unable to carry out~~ (an operation)

In order to allow magnetic tape as a storage medium, we stipulate that the content of the tape beyond the writing heads is null.

The operations provided by a handler are listed below; their execution may require human operator assistance; but the necessary communication and administration is concealed beneath the abstract hardware interface.

openread: loads the handler with a write-protected pack with given name and content

openwrite: loads the handler with a blank tape with given name; and permits writing and closing only.

close: unloads the handler.

read: reads the current block,

step: steps the reading head one block forward.

write: writes a given block at the end of the current content of the pack.

more: a Boolean indicating that there are more blocks to read, or more space to write.

search: positions the handler to read the block with the specified index.

STATUS = {normal < nearend < atend}

HANDLER =

empty: BOOL;
pn: PACKNAME;
pc: PACKCONTENT;
pos: NAT;
space: NAT;
rw: {"read", "write"};

---

pos ≤ length(pc);
rw = "write" ⇒ pos = length(pc)

HOP = HANDLER ↔ HANDLER'

openread: PACKNAME × PACKCONTENT → HOP

openread = λ(n,c).{HOP | ¬empty ⇒ pn' = n & pc' = c &
rw' = "read" & pos' = 0 &
¬empty'}

openwrite: PACKNAME → HOP;

openwrite = λn.{HOP | empty ⇒ pn' = n & pc' = <> &
rw' = "write" & pos' = 0 &
¬empty'}

close = {HOP | ¬empty'}

read: HANDLER → BLOCK;

read = λ(h | h.rw = "read" & ¬empty) . . . h.pc(pos)

step: HOP

step = {HOP | rw = "read" & ¬empty
& pos < length(pc)^ ⇒
¬empty' & pn' = pn & pc' = pc
& pos' = pos + 1 & rw' = rw }

write: BLOCK → HOP

write = λb. {HOP | ¬empty & rw = "write" ⇒
¬empty' & pn' = pn & pc' = pc^<b>
& pos := pos + 1 & space' ≥ space
& rw' = rw }

more: ℙ(HANDLER)

more = {HANDLER | ¬empty & (rw = "read" & pos < length(pn)
∨ rw = "write" & pos < space)}

3. Operating Instructions

The security of a file backup server depends critically on the security of the library of packs stored away from the computer, perhaps in a fire-proof safe. It also depends on the reliability and simplicity of the operating instructions; these must be especially simple in the case when something has gone wrong, since then recovery must be made from a situation which is already critical. One simplification is to distinguish between the handler used for reading and that used for writing.

The first most essential rule for operation is that each pack has an external label, on which is displayed its pack name, and the dates and times at which it has been written to. For extra security, the date-time should be inscribed on the label both before the pack has been inserted in the writing handler, and again after removal. The second rule is that the library of dismounted packs should be stored in such a way that the

most recently written tape is easily identifiable, and also the pack with the earliest date of last writing. This (latter) pack is known as the "victim", because it is the one which will next be written to. Thirdly, each pack must be held in a fixed place in a rack, and each pack must be returned to this place in the rack from which it came; so that it can be readily found again.

One way of achieving these aims is to store the packs in a circular rack like a carousel, with a rotating pointer which always points to the gap between the victim and the most recently written tape.

Let us now consider a worst case, in which the operating computer has developed a fault, and some of the library packs have been corrupted or destroyed. (When the computer fault is mended, the following sequence of operations should be carried out.

(1) Sort all available library packs into ascending order of writing date.

(2) Load the most recently written pack in the reading handler.

(3) The computer will display the date and the expected content of the library. Check this against the actual library; if any of the packnames or dates do not correspond, this may mean that you have failed to load the most recently written pack. If so, go back to step (2), and restart the program.

(4) If the computer finds the loaded pack unreadable, it will inform you; mark the label of the pack "UNREADABLE", and load the next most recent pack, and repeat as necessary from step (2)

(5) If the whole library is lost or unreadable, follow the instructions for system initialisation. Otherwise, remove the (first) successfully read pack, and return it to the library, with the status of the most recently written pack.

(6) Load the victim pack in the writing handler.

If the operating computer has no form of non-volatile internal storage, these above procedures must be carried out whenever the computer is switched on; otherwise, all that is necessary is to load the victim pack in the write handler.

There are also operating procedures for introducing and labelling new packs, for purging library packs, for changing the pack being written when it is full, and for loading a pack onto the reading handler so that a file stored on it can be recovered. All these operations are called for by the computer when required; and their correct execution is checked by the computer. Thus the design of these operating procedures is not so critical and can be postponed. The need to design the recovery operation first was pointed out by A. S. Fraser.