THE DISTRIBUTED COMPUTING SOFTWARE PROJECT

by

Roger Gimson
Carroll Morgan

Technical Monograph PRG-50

July 1985

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Introduction

Work on the Distributed Computing Software Project began at Oxford University Programming Research Group in 1982. The goal of the project is to construct and publish the specification of a loosely-coupled distributed operating system, based on the model of autonomous clients having access to a number of shared services.

A fundamental objective of the project is to make use of mathematical techniques of program specification to assist the design, development and presentation of distributed system services.

In this monograph we present some of the results of the first stage of the project.

In the first chapter we include a paper which gives an overview of the use of mathematics in system design, and its application to the specification of an example file service. It illustrates how abstraction from details of implementation can allow the exploration of novel system designs.

The following chapters contain the user documentation for some of the services which have been implemented to date. They illustrate how it has been possible to make use of mathematical techniques to provide precise manuals for users of the services.

The project is funded by a grant from the Science and Engineering Research Council.

Contents

Chapter 1

The Role of Mathematical Specifications

This chapter appeared in the book "Distributed Computing Systems Programme" (ed. D.A.Duce), published by Peter Peregrinus Ltd. 1984, under the title "Ease of use through proper specification".

1.1 Introduction

The aim of the Distributed Computing Software Project is to explore the new possibilities of distributed operating system design which have been made possible by the low cost of distributed processing hardware. The mathematical techniques of program specification and development play a crucial part in this aim because:

> we can use *mathematical specifications* to explore designs motivated purely by ease-of-use rather than by ease-of-implementation (since specification allows abstraction from implementation constraints);
>
> we will have a precise notation in which such designs can be reliably communicated to others, and which will assist the discovery and discussion of the designs' implications;
>
> it will be possible to present the specifications directly in the user manuals of the distributed operating system, thus increasing their precision while decreasing their size; and
>
> we will be able to use the mathematical techniques of *refinement* to produce implementations which are highly likely to satisfy their specifications (and hence also will be accurately described by their user manuals).

It is especially important that those benefits should be realised in the construction of a *distributed* operating system - because distributed operating systems offer the rare opportunity for the user to control the system, rather than vice versa. The high bandwidth of current local area networks allows an efficient modularity; for example, a structure consisting of largely autonomous services and clients is entirely feasible. In such a system, the choice between (rival) services, and the manner in which they are used, would be entirely up to the clients. This is the basis of the *open systems* approach: provided services are well-specified, clients are free to make use of them in whatever manner is consistent with their specification.

1.2 A first example

One of the most visible parts of any operating system is its file system. Even today, the design of these range in quality from excellent to horrific. But others of course may think instead that they range from horrific to excellent: the features one user cannot do without, another may abhor. It is through such *features* that an operating system controls (even the thoughts of) its clients, and this is exactly what we hope to avoid.

A file service in a distributed operating system is there to be shared by as many clients as possible. To achieve this, it must be unopinionated: it must have so *few* features that there is nothing anyone could object to. It is only in the context of specification that we can propose such a radical design; any less abstract context introduces efficiency constraints. Some of these, of course, will have to be met eventually, but perhaps not all of the ones that might conventionally be presumed. We must not introduce such constraints simply because we could not express ourselves without them: first we state what we would like - then we can compromise.

As an example, let us consider the simplest file system design one could imagine. We describe it as a partial function "files" from the set "NAME" of file names to the set "FILE" of all possible files; and we say nothing about the structure of the sets NAME and FILE themselves:

$$files : NAME \pfun FILE$$

The mathematical notation above introduces the variable files, and gives its type as $NAME \pfun FILE$. The English text states that this variable is to describe the file system. Our style of mathematical specification is an example of the Z specification technique, and we will continue to use it below. It is not possible for us to explain Z itself in any detail in this document, but we hope its flavour will be evident; and in any case the bulk of the meaning will be conveyed by the English. Sufrin [8] and Morgan [3,4] together give an introduction to Z. A glossary of the mathematical symbols used is provided at the end of this chapter.

We propose two operations only on the file system: StoreFile stores a (whole) file, and RetrieveFile (destructively) retrieves it.

StoreFile

Let files be the state of the file system before the operation, and let files' be the state afterwards. Let file? be the file to be stored, and let name! be some filename, chosen by the filesystem, which will refer to the newly stored file (we conventionally use words ending in ? for inputs, and in ! for outputs). That is, given

$$\begin{aligned} &files, files' : NAME \pfun FILE \\ &file? \quad\quad : FILE \\ &name! \quad\quad : NAME \end{aligned}$$

the effect of StoreFile is to choose a new name, which is not currently in use

$$name! \notin dom\ files$$

and to update the partial function by overriding its current value, so that after the operation it maps the new name to the newly-stored file

$$files' = files \oplus [name! \mapsto file?]$$

(We notice as an immediate advantage of our abstraction that we have given the implementor the freedom to store identical but differently named files using shared or separate storage, as he chooses.)

RetrieveFile

Let files be the state of the file system before the operation, and let files' be the state afterwards. Let name? be the name of the file to be retrieved, and let file! be the file itself. That is, given

$$\begin{aligned} &files, files' : NAME \pfun FILE \\ &name? \quad\quad : NAME \\ &file! \quad\quad : FILE \end{aligned}$$

the effect of RetrieveFile is to return the named file to the client

$$file! = files\ (name?)$$

provided it exists

```
name? ∈ dom files
```

and to remove the name (and hence the file) from the partial function which represents the file system

```
files' = files \ {name?}
```

The description above is "of course" not feasible with today's technology - which is a pity. It would be too impractical to have to retrieve a whole large file if we wished, say, just to read one small piece of it. But how wonderful it would be if a file system could be so simple! At least we were able to describe it.

1.3 The first compromises

The best we can do with our simple file system is to use it as the basis for a development of a more practical design - and the description above provides a context into which the necessary compromises can be introduced. Here are some of them (in no particular order):

Compromise	Reason
It must be possible to read the file without deleting it.	The communication medium is not entirely reliable - a breakdown during retrieval could destroy the file without returning its contents.
Clients must be prevented from destroying the files of others (remember, a file can't be updated).	Mistakes are inevitable - even honest clients could accidentally destroy other clients' files.
Files must be given a limited lifetime, and clients must be charged for their storage.	Any implementation of the file system, however capacious, will still be finite.

We introduce these compromises in a revised design (again using the notation of Z). First, we name three new sets

CLIENT - the set of client identifications,

TIME - the set of instants (e.g. seconds from 1st January 1980 - but we need not be specific here),

COST - the set of costs (e.g. pence).

The definition of a file is extended to include the identification of its owner, and its time of creation and (eventual) expiry. DATA is a fourth new set which contains all the possible values a client could store in a file (its contents). We will collect these attributes in a schema FILE, and state at the same time that in any file, the creation time must precede the expiry time:

FILE
owner : CLIENT
created,
expires : TIME
contents : DATA
---
created ≤ expires

The schema FS below describes the state of the file storage system itself:

FS
files: NAME ⇸ FILE

and the schema ΔFS describes the general aspects of any operation on it:

ΔFS
files, files': NAME ⇸ FILE
who : CLIENT
when : TIME

who is the identity of the client performing the operation, and when is the time at which it is performed. We can abbreviate ΔFS (without changing its meaning) by building it from the schema FS instead of directly from the variable files:

```
ΔFS
    FS
    FS'
    who : CLIENT
    when: TIME
```

StoreFile

The (revised) StoreFile operation we will present as a schema including the variables files, files', who, and when (supplied by ΔFS), as well as the data to be stored (contents?), the expiry time (expires?), the new name chosen by the service (name!), and the charge made in advance (cost!):

```
StoreFile
    ΔFS
    contents?: DATA
    expires? : TIME
    name!    : NAME
    cost!    : COST
    ---------------------------
    (∃FILE'.
        owner'    = who
        created'  = when
        expires'  = expires?
        contents' = contents?

        name! ∉ dom files
        files' = files ⊕ [name! ↦ FILE']
        cost!  = Tariff (FILE'))
```

A new file FILE' is constructed which is owned by the client storing it, which records its creation time as the time it was stored, which will expire at the time the client specified (then becoming inaccessible), and whose contents the client supplies.

A new name name! is chosen, not currently in use, and the file is stored under that name. The charge made is some function Tariff of the file (hence of its owner, creation and expiry times, and contents). Here is a possible definition of Tariff (which depends in turn on some function Size):

```
Tariff = (λFILE. (expires - created) * Size(contents))
```

ReadFile

The ReadFile operation returns the expiry time and the contents of the file stored under a given name. Its parameters are the name of the file to be returned (name?), when it will expire (expires!), and its contents (contents!):

```
ReadFile
    ΔFS
    name?    : NAME
    expires! : TIME
    contents!: DATA
    ─────────────────
    FS' = FS

    (∃FILE.
        FILE      = files (name?)
        expires   > when
        expires!  = expires
        contents! = contents)
```

ReadFile does not change the state of the service. The map files is applied to the name, to determine the file's value FILE, which must not have expired. Its expiry time and contents are returned.

DeleteFile

The DeleteFile operation removes a file from the service. A rebate is offered as an incentive to deletion before expiry. name? is the name of the file to be deleted, and cost! is the (possibly negative) charge made for doing so (we assume negation "-" is defined on COST):

```
DeleteFile
    ΔFS
    name?: NAME
    cost!: COST
    ---------------------------------
    (∃FILE.
        FILE    = files (name?)
        expires > when
        owner   = who
        files'  = files \ {name?}
        cost!   = - Rebate (FILE, when))
```

The map files is applied to the name, to determine the file's value FILE, which must not have expired. It must be owned by the deleting client. The file's name name? (and hence the file itself) are removed from the partial function which represents the stored files, and the cost is determined by a function Rebate of the file and its deletion time. Here is a possible definition of Rebate:

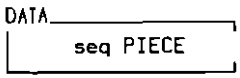
```
Rebate = (λFILE; when: TIME.  (expires - when) * Size(contents))
```

Naturally, there are other compromises which could be made, in addition to or instead of those above. In the next section, however, we discuss a compromise which we suggest should *not* be made.

1.4 A compromise avoided

One glaring inefficiency remains in our proposal: that we must transfer whole files at once. Many clients will not have time or the resources (e.g. local memory) to do this. But here we will not compromise by modifying our file storage service to cater for this inefficiency - rather we insist that the business of the file storage service will be file storage exclusively. Partial examination and updating will be the business of a file updating service.

To propose a service which treats the contents of files as having structure, we must propose a structure. The proposal we make is the very simple view that the contents of a file is a sequence of *pieces*. (Sequences are functions from the natural numbers N to their base type, and begin at index 1.) We do not wish to say what a piece is, however, for this description.



The file updating service in fact has no state; all its work is done in the calculation of its outputs from its inputs. Its two operations are ReadData and UpdateData.

ReadData

ReadData takes the contents of a file contents?, a starting position start?, and a number of pieces to be read number?, and returns the data pieces! at that position within contents?. (#pieces! is the length of the sequence pieces!, and 1..#pieces! is the set {i: N | 1 ≤ i ≤ #pieces!}.)

```
ReadData
    contents?: DATA
    start?,
    number?  : ℕ
    pieces!  : DATA
    ─────────────────────────────────────
    #pieces! = min (number?, (#contents? - start?))

    (∀i: 1..#pieces!. pieces!(i) = contents?(i + start?))
```

The length of the data returned is equal to the number of pieces requested, if possible; otherwise, it is as large as the length of contents? will allow. The i^{th} piece of pieces! returned is equal to the $(i+start?)^{th}$ piece of contents?.

UpdateData

UpdateData takes the contents of a file contents?, a position start?, and some data pieces?, and returns an updated contents contents!.

```
UpdateData
    contents?,
    contents! : DATA
    start?    : ℕ
    pieces?   : DATA
    ─────────────────────────────────────
    #contents! = max (#contents?, (start? + #pieces?))
    start?     ≤ #contents?

    (∀i: 1..#contents!.
       (i - start?) ∈ 1..#pieces?
          ⇒ contents!(i) = pieces?(i - start?)

       (i - start?) ∉ 1..#pieces?
          ⇒ contents!(i) = contents?(i))
```

The length of the new contents is equal to its original length, unless an extension was necessary to accommodate the new data; however, the new data must begin within the original contents or immediately at its end. The i^{th} piece of contents! is equal to the $(i\text{-}start?)^{th}$ piece of pieces?, if this is defined; otherwise, it is equal to the i^{th} piece of contents?.

Our proposal is of course only one of the many possible (for a different proposal, see the definition of these operations in Morgan and Sufrin [5]). We could, of course, propose *several* updating services, each providing its own set of facilities. Moreover, the original operations which transferred whole files would still be available to those clients able to use them (see figure 1.1).

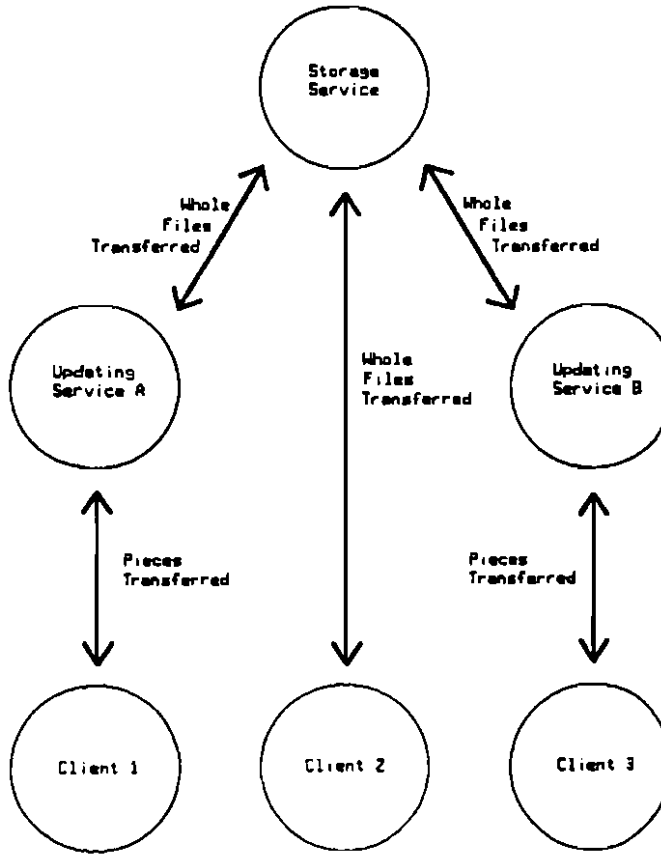


Figure 1.1 Separate updating and storage services

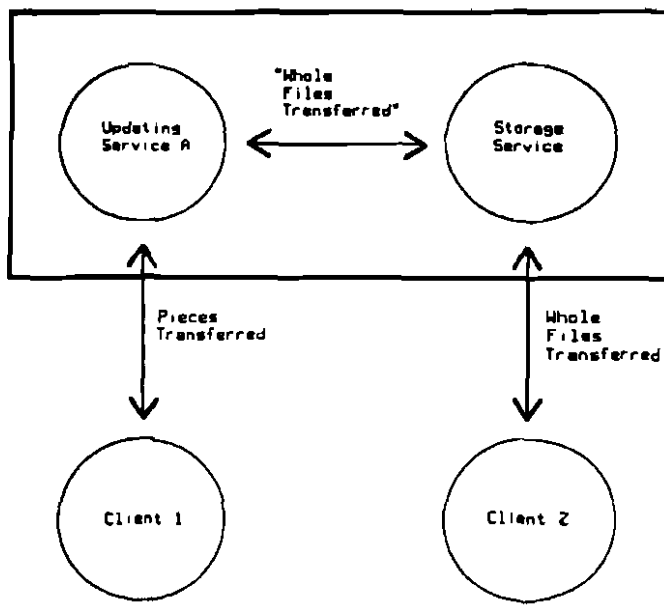


Figure 1.2 Combined updating and storage service

1.5 Modularity and composition of services

The structure we have presented above separates the issues of how files should be stored from how they should be manipulated. As a result, we have offered the user an unusual freedom of choice - he can read just one piece of a file, or he can treat a file as a single object (with the corresponding conceptual simplification; Stoy and Strachey [7] for example allow this in their operating system OS6).

Still, it is likely that a further compromise will be necessary: for large files, the time taken to transfer the file between the two services (storage and updating) may not be tolerable. We solve this not by changing our design, but by an engineering decision: for applications that require it, we will provide the two services together *in one box*, and the transfers will be internal to it (see figure 1.2). Its specification we construct by combining the material already available.

StoreFile, ReadFile, and DeleteFile will be available as before. However, we introduce two new operations, ReadStoredFile and UpdateStoredFile, whose specifications will be formed by composing the specifications given above. (The schema composition operator "$\fatsemi$", used for this, is defined in [3]. Here, we will explain it informally.)

ReadStoredFile

Reading a stored file is performed by first reading the whole file with ReadFile, and then reading the required portion of its contents using ReadData. In Z we write this

$$\mathit{ReadStoredFile} \;\widehat{=}\; \mathit{ReadFile} \fatsemi \mathit{ReadData}$$

If we were to expand this definition of ReadStoredFile, the result would be as below:

```
ReadStoredFile
ΔFS
name?   : NAME
start?,
number? : ℕ
expires!: TIME
pieces! : DATA
---
FS' = FS

(∃FILE.
  FILE    = files (name?)
  expires > when
  expires! = expires

  #pieces! = min (number?, (#contents - start?))
  (∀i: 1..#pieces!.
     pieces!(i) = contents(i + start?)))
```

ReadStoredFile takes a file name name?, a starting position start?, and a number of pieces number?, and returns the expiry time of the file expires!, and the data pieces! found at the position specified. (expires! is returned by ReadStoredFile because ReadFile returns it; we could have dropped this extra output, but choose not to introduce the Z notation for doing so.)

UpdateStoredFile

The complementary operation UpdeteStoredFile is a more difficult composition, since we must accumulate the costs of the component operations, and we must ensure the updated file is (re-)stored under its original name. For the sake of honesty, we will give the definition, but we will not expand it:

```
UpdateStoredFile  ≙

        ReadFile ⨾
        DeleteFile [dcost!/cost!] ⨾
        UpdateData ⨾
        StoreFile [name?/name!, scost!/cost!] ⨾
        (dcost?, scost?, cost!: COST | cost! = dcost? + scost?)
```

UpdateStoredFile first reads the whole file, then deletes it, then updates it, and then stores its new value under its original name. Finally, it presents as its overall cost the sum of the two charges made by DeleteFile (which may well be negative) and StoreFile.

What we have done is to compose two simple but infeasible operations to produce a more complicated but feasible one (rather like the use of complex numbers in electrical engineering, for example). Naturally, the implementor need not transfer whole files back and forth within his black box on every read and update operation - but nevertheless the updating and storage service provided by the box *must* behave as if he does: that is, it must behave as we have specified. Our decomposition was chosen for economy of concept; the implementor's must be chosen for economy of time and equipment, and the whole range of engineering techniques are available to him to do so (caches, update-in-place, etc.).

1.6 Experience so far

While the project has followed the general principles above, it has in fact adapted to constraints in different ways. Its storage service, which we have implemented in prototype, stores *blocks* of a fixed size (rather like the service described by Biekert and Janssen[1]). This distinguishes it as a "universal" storage service from, say, the one implemented at Cambridge (described by Needham and Herbert[6]). Organisation of blocks into files, the keeping of directories, etc. is done by software in the clients' own machines (for example, using a "File Package" as described by Gimson [2]). This allows clients freedom in the choice of what file structure they build, but of course makes the sharing of files more difficult. If one package should become popular, however, it could be placed in a machine of its own, and so become a service.

There are many aspects of the project that it has not been possible to cover. For example, the specification of the errors that may occur in use is an essential part of the full specification of a service. We include such details in the user manuals of the services we have implemented. The manuals follow the style of specification presented here, combining formal text and English narrative to give a precise yet easily understandable description of the user interface to a service.

So far, the pressure of simplicity in our mathematical descriptions has kept the designs correspondingly simple. At present, they are perhaps too much so; but by using mathematical specification techniques we have built basic services which genuinely are simple. And that is where one must begin.

1.7 Future plans

The styles of specification, and of presentation of user manuals, has to some extent been developed in parallel with the software to which they have been applied. These styles are now more stable, and further services will be specified, designed, and implemented in the same way.

The goal of the project is to produce a suite of designs from which implementations can be built on a variety of machines. Each design will be documented, in a mathematical style, both for the user and for the implementor. Thus the primary goal is to construct a distributed system on paper.

For a paper construction to have any value, the designs proposed in it must be widely applicable, and genuinely useful. Machine-independent techniques of description will take care of the first requirement. To ensure that the second is met, prototype implementations must be constructed of each of the designs, and experience must be gained of their use.

1.8 Glossary of symbols

$\in$	"is an element of"
$\notin$	"is not an element of"
$\exists$	"there exists"
$\forall$	"for all"
$\triangleq$	"is syntactically equivalent to"
$\mathbb{N}$	The set of natural numbers (non-negative integers)
$m..n$	The set of natural numbers between m and n inclusive $m..n \triangleq \{k: \mathbb{N} \mid m \leq k \leq n\}$
$\{sig \mid pred\}$	The set of sig such that $pred$
$A \nrightarrow B$	The set of partial functions from A to B
$[a \mapsto b]$	The function $\{(a, b)\}$ which takes a to b
$f(x)$	The function f applied to x
dom	The domain of a relation (or function) for $f: A \nrightarrow B$, $\mathrm{dom}\ f \triangleq \{a: A \mid (\exists b: B \cdot b = f(a))\}$
/	Domain restriction for $f: A \nrightarrow B$; $\underline{A} \subseteq A$, $f / \underline{A} \triangleq \{(a,b): f \mid a \in \underline{A}\}$
\	Domain co-restriction for $f: A \nrightarrow B$; $\underline{A} \subseteq A$, $f \setminus \underline{A} \triangleq \{(a,b): f \mid a \notin \underline{A}\}$
$\oplus$	Functional overriding for $f, g: A \nrightarrow B$, $f \oplus g \triangleq (f \setminus \mathrm{dom}\ g) \cup g$

seq A	The set of sequences whose elements are drawn from A seq A ≙ {s: ℕ ⇸ A | (∃n: ℕ . dom s = 1..n)}
#s	The length of sequence s dom s = 1..#s
[new/old]	Schema variable renaming
⨾	Schema forward composition

1.9 References

[1] Biekert, R., and Janssen, B., 1983, "The implementation of a file system for the open distributed operating system Amoeba", Informatica Rapport, Vrije Universitiet, Amsterdam.

[2] Gimson, R. B., 1983, "A File Package - User Manual", Distributed Computing Project Working Paper, Programming Research Group, Oxford University

[3] Morgan, C. C., 1984, "Schemas in Z - a preliminary reference manual", Distributed Computing Project Working Paper, Programming Research Group, Oxford University

[4] Morgan, C. C., 1984, "Schemas in Z - an example", Distributed Computing Project Working Paper, Programming Research Group, Oxford University

[5] Morgan, C. C., and Sufrin, B., 1984, "Specification of the Unix File System", IEEE Trans. Soft. Eng., March 1984

[6] Needham, R., and Herbert, A., 1982, "The Cambridge file service", in "The Cambridge Distributed Computing System", Addison-Wesley, 41-63

[7] Stoy, J. E., and Strachey, C., 1972, "OS6 - An operating system for a small computer", Comp. J. 15, 2, 195-203

[8] Sufrin, B., 1983, "Mathematics for system specification", Lecture Notes 1983-1984, Programming Research Group, Oxford University

Chapter 2

Authentication of User Names

2.1 Nicknames and usernames
2.2 Authentication
2.3 Guest user

2.1 Nicknames and usernames

As a short-term measure, a very simple scheme has been chosen to make it difficult for one client to impersonate another.

Each registered client has a *nickname* and a *username*. Nicknames are allocated from a set Nickname, and the allocation is public - that is, it is common for clients to know each others' nicknames. It is expected that nicknames will change only rarely, if at all.

Usernames are allocated privately, from a set User; a client should not reveal his username to anyone else. Since usernames may become compromised (known by too many people) or forgotten (known by too few!), it might be necessary to change a client's username from time to time.

2.2 Authentication

Authentication is achieved by the existence of a (secret) partial function

$$GetNickname : User \pfun Nickname$$

which gives for any username the nickname of the client who should be its sole possessor. Since the set User of usernames has been made very large, and the set

$$dom\ GetNickname$$

of *authentic* usernames has been made a relatively small part of it, it will be hard for clients to guess the usernames of others. Services therefore may use the function GetNickname to authenticate their clients; they might reject requests for which

$$client? \notin dom\ GetNickname$$

2.3 Guest user

There is a *guest* username GuestUser which some services might recognise as a special case. This username is public, and is expected to be used by clients temporarily without a private username of their own. It is guaranteed that the guest username is not the authentic username of any client

$$\mathsf{GuestUser} \in \mathsf{User} - \mathsf{dom}\ \mathsf{GetNickname}$$

Chapter 3

Time Service - User Mannal

3.1 Time service operation

The time service provides only one user operation, *GetTime*, which returns the current time

in seconds since 00:00:00 1 January 1980.

The description of the operation has three sections, titled **Abstract, Definition** and **Reports.**

The Abstract section gives a procedure heading for the operation, with formal parameters, as it might appear in some programming language. The correspondence between this procedure heading and an implementation of it in some real programming language must be obvious and direct.

Each formal parameter is given a name ending with either ? or !. Those ending with ? are inputs, and those ending with ! are outputs.

The Definition section defines the meaning of the operation.

The Reports section lists the possible (success or failure reporting) values which the `report!` formal parameter can assume. Reports are discussed in more detail in section 3.2.

GETTIME

Abstract

```
GetTime (client?: User;
         now!   : Time;
         cost!  : Money;
         report!: Report)
```

Definition

The current time now! is returned, measured in seconds from 00:00:00 1st January 1980. The cost cost! is fixed. The client's username client? must be authentic (see Chapter 2).

```
client? ∈ dom GetNickname
```

Reports

```
Success
ServiceError
```

3.2 Error reports

The report parameter `report!` indicates whether the operation succeeded or failed. The value `Success` indicates the operation succeeded.

The value `ServiceError` indicates the operation failed; in this case, no reliance should be placed on any other values returned. Possible reasons for this report are:

The service isn't running
There was a communication error

3.3 UCSD Pascal interface

```
UNIT TI;

INTERFACE {UNIT TI  28-Nov-83}

{Time Service  -  UCSD-Pascal Interface}

  USES {$U SVTYPES.CODE} SV_Types;


TYPE
  TI_Report = (TI_Success, TI_ServiceError);


PROCEDURE TI_GetTime (    InClient  : SV_User;
                      VAR OutNow    : SV_Time;
                      VAR OutCost   : SV_Money;
                      VAR OutReport : TI_Report);

  {return current time}
```

3.4 Modula-2 interface

```
DEFINITION MODULE TI;  (* Roger Gimson  22-Feb-84 *)

(* Time Service  -  Modula-2 Interface *)

FROM SVTypes IMPORT User, Time, Money;

EXPORT QUALIFIED Report, GetTime;

TYPE
  Report = (Success,
            ServiceError);

PROCEDURE GetTime (    InClient  : User;
                   VAR OutNow    : Time;
                   VAR OutCost   : Money;
                   VAR OutReport : Report);
  (* return current time *)

END TI.
```

Chapter 4

Reservation Service - User Manual

4.1 Introduction

The distributed operating system at the Programming Research Group is made up of various services which are largely independent. In particular, it's possible that one service can be turned on or turned off while other services and clients continue to run.

When a service is turned off (*shutdown*), there should not be any client who is at that moment involved in some *series* of interactions with it - because interruption of such a series could be quite inconvenient (for the client). If these series (or *transactions*) can be recognised by the service, it is possible to avoid this inconvenience as follows.

Shutdown procedure:

1. The operator *requests* shutdown of the service.
2. The service rejects any attempt to begin a new transaction, but allows current transactions to continue.
3. When all transactions have completed, the service notifies the operator that shutdown is complete.

However, there are some problems; for example, a client might *himself* fail to complete a transaction (presumably due to accidental failure of his own software). If this happened, the service would *never* shutdown. A more serious problem is that for some services (*e.g.* the block storage service) there is no recognisable transaction structure, and so the above scheme cannot be used at all. We solve both problems with an independent *Reservation Service*.

The reservation service does not interact at all with the service it reserves; it interacts only with its own clients, and with the operator. It allows clients to state when, and for how long, they would like to use the reserved service, and it allows the operator to state a *shutdown time* beyond which all reservations are to be rejected. It becomes the clients' responsibility to protect themselves from sudden shutdown of the service (by making reservations), and the operator's responsibility to turn off the service only after the shutdown time (which he may set). Thus a shutdown can be unexpected only by those clients who have made no reservation.

A typical use of the reservation service would be for clients to include a reservation request at the start of every program using the reserved service. The duration of the reservation should be long enough to allow the program to complete, but short enough to allow the operator to make a reasonably spontaneous decision to shutdown.

The state of the reservation service has two components:

expires	a map from clients' nicknames (their public identities - see Chapter 2) to the time at which their current reservation expires
shutdown	the shutdown time most recently set by the operator.

RS

```
expires : Nickname ⇸ Time
shutdown: Time
```

Each operation requested by clients includes the three values:

client?	the username of the client
cost!	the cost of the operation
report!	a report indicating whether the operation succeeded or failed.

ΔRS

```
RS
RS'

client? : User
nickname: Nickname

cost!   : Money
report! : Report
---
client? ∈ dom GetNickname
nickname = GetNickname (client?)
```

The username client? is supplied by the user; it is his *private* username (as distinct from his public nickname). client? must be authentic (client? ∈ dom GetNickname) if the service is not to ignore the request.

The client's nickname is calculated by the service. cost! and report! are returned to the user by the service.

4.2 Reservation service operations

Three operations are described in this section: *Reserve*, which is requested by clients, *SetShutdown*, which is requested by the operator, and *Scavenge*, which is performed by the service itself (at its discretion). The latter two operations are included here only as an aid to the reader's intuition.

The description of each operation can have up to four sections, titled **Abstract**, **Definition**, **External Calls** and **Reports**.

The Abstract section gives a procedure heading for the operation, with formal parameters, as it might appear in some programming language. The correspondence between this procedure heading and an implementation of it in some real programming language must be obvious and direct.

Each formal parameter is given a name ending with either ? or !. Those ending with ? are inputs, and those ending with ! are outputs.

A short description may accompany the procedure heading.

The Definition section mathematically defines the operation, by giving a schema which includes as a component every formal parameter of the procedure heading; within the schema also appear subschema(s) whose components include the service state before and after the operation (this can be more (RS, RS') or less (ΔRS) explicit). Any other components appearing in the schema are either local to the operation (that is, temporary) or represent values exchanged with other services (invisibly to the client).

Only the formal parameters of the procedure heading are exchanged directly between client and service.

A short description may accompany the schema.

The External Calls section lists the calls this service may make on other services, in order to complete the requested operation. These appear as procedure calls which match the procedure headings given in the description of the operation called. (These are found in the user manual for the called service.) The correspondence between formal and actual parameters is positional, with missing (i.e. irrelevant) actual parameters indicated by commas.

The Reports section lists the possible (success or failure reporting) values which the report! formal parameter can assume. If such a value is followed by a component in parentheses and/or a predicate, it is to suggest that the reported value would occur because that component satisfied the predicate. The component and predicate are therefore a hint to the cause of the report.

Reports are discussed in more detail in section 4.3.

RESERVE

Abstract

```
Reserve (client?  : User;
         interval?: Interval;
         until!   : Time;
         cost!    : Money;
         report!  : Report)
```

A reservation is made for a period of interval? seconds. until! returns the expiry time of the new reservation.

A client can cancel his reservation by making a new reservation in which interval? is zero; see *Scavenge* below.

Definition

Reserve

```
ΔRS
interval?: Interval
until!,
now      : Time
─────────────────────
until! = now + interval?
until! ≤ shutdown

expires'  = expires ⊕ [nickname ↦ until!]
shutdown' = shutdown

cost! = 20
```

The reservation must expire before the shutdown time. The current time now is obtained from the time service (see Chapter 3).

External Calls

Time Service

```
GetTime (,now,,Success)
```

now is obtained by a successful call of *GetTime*. It is measured in seconds from 00:00:00 1st January 1980.

Reports

```
Success
ServiceError

NotAvailable        ⇒    shutdown < now + interval?
                         shutdown = until!

TooManyUsers        ⇒    #expires = Capacity
```

SETSHUTDOWN

Abstract

```
SetShutdown (shutdown? : Time;
             threatens!: Boolean)
```

The operator may set a new shutdown time. He is informed if the new time threatens existing reservations; if it does, it is his responsibility to negotiate with the clients affected.

Definition

SetShutdown

```
RS
RS'
shutdown? : Time
threatens!: Boolean
---
shutdown' = shutdown?
threatens! ⇔ (∃expiry: ran expires. expiry > shutdown')

expires' = expires
```

The shutdown time is changed to the new value *regardless of existing reservations*. Reservations are unaffected.

SCAVENGE

Abstract

Scavenge()

The service can at any time remove reservations whose expiry time is in the past. This is in fact the *only* way in which reservations are removed (by client, operator or service).

Definition

Scavenge

RS
RS'
now: Time

---

shutdown' = shutdown

expires' ⊆ expires

(∀removed: dom expires. removed ∉ dom expires'
⇒ expires(removed) ≤ now)

Scavenge does not change the shutdown time.

Scavenge can remove reservations, but it never makes new ones. A reservation is removed only if its expiry time is in the past.

External Calls

Time Service

GetTime (, now, , Success)

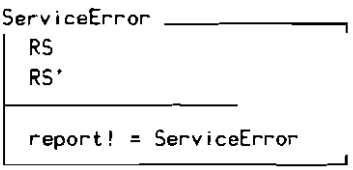
now is obtained by a successful call of *GetTime*. It is measured in seconds from 00:00:00 1st January 1980.

4.3 Error reports

The report! parameter of each operation indicates either that the operation succeeded or suggests why it failed; in most cases, failure leaves the service unchanged.

An operation can return only the report values listed in the Reports section of its description. If it returns the value Success, it must satisfy its defining schema. If it returns any other value, it must satisfy instead the appropriate schema below.

ServiceError



ServiceError indicates an unexpected failure, which might not be the client's fault. These are typical causes:

Service not running

Network (hardware or protocol) failure
Service hardware fault
Service software error

NotAvailable

```
NotAvailable ──────────────────────
  ΔRS
  interval?: Interval
  until!,
  now      : Time
  ──────────────────
  report! = NotAvailable

  shutdown < now + interval?
  until! = shutdown

  RS' = RS
```

If the reservation cannot be made due to early shutdown, the shutdown time itself is returned in until!.

now is obtained from the time service.

TooManyUsers

TooManyUsers

ΔRS

now: Time

---

report! = TooManyUsers

#expires = Capacity

nickname $\notin$ dom expires

RS' = RS

The service has finite capacity Capacity for recording reservations; this report occurs when that capacity would be exceeded. The report *cannot* occur if the client has a reservation (since it is overwritten by the new one).

now is obtained from the time service.

Clients who can't themselves make reservations might be able to rely temporarily on the reservations of others.

4.4 UCSD Pascal interface

```
UNIT RI;

INTERFACE {UNIT RI  28-Nov-83}

{Reservation Service  -  UCSD-Pascal Interface}

  USES  {$U SVTYPES.CODE} SV_Types;


TYPE
  RI_Report = (RI_Success, RI_ServiceError, RI_NotAvailable,
               RI_TooManyUsers);


PROCEDURE RI_Reserve (     InClient    : SV_User;
                           InInterval  : SV_Interval;
                      VAR  OutUntil    : SV_Time;
                      VAR  OutCost     : SV_Money;
                      VAR  OutReport   : RI_Report);

  {reserve use of the service for InInterval, terminating at
   OutUntil, otherwise return the time at which the service
   becomes unavaileble in OutUntil}
```

4.5 Modula-2 interface

```
DEFINITION MODULE RI;  (* Roger Gimson  22-Feb-84 *)

(* Reservation Service  -  Modula-2 Interface *)

FROM SVTypes IMPORT User, Time, Interval, Money;

EXPORT QUALIFIED Report, Reserve;

TYPE
  Report = (Success,
            ServiceError,
            NotAvailable,
            TooManyUsers);

PROCEDURE Reserve (     InClient    : User;
                        InInterval  : Interval;
                  VAR OutLimit      : Time;
                  VAR OutCost       : Money;
                  VAR OutReport     : Report);
  (* reserve use of the service for InInterval, terminating at
     OutLimit, otherwise return the time at which the service
     becomes unavailable in OutLimit *)

END RI.
```

Chapter 5

Block Storage Service - User Manual

5.1 Introduction

The block storage service stores blocks on behalf of its clients. A client may submit some *data*

$$data : Data$$

which the service will store within a block

$$\begin{array}{|l} \textbf{Block} \\ \hline owner : Nickname \\ created : Time \\ expires : Time \\ data : Data \\ \hline \end{array}$$

As well as containing the client's data, the block records as its *owner* the nickname (see Chapter 2) of the client who submitted it, and it records the time of its creation. Whenever a block is created, a *lifetime* must be given by the client; it is the number of seconds for which the service is obliged to store the block. After its lifetime, a block is said to have *expired*, and can be discarded by the service without notification of the client. A *name* will be issued by the service when the block is created

$$name : Name$$

which becomes the client's reference to the block. Any subsequent operations on the block will require this name.

The service contains a mapping from block names to blocks; it contains also a finite set of *new* block names which it has not yet issued. When a new name is issued, it is taken from this set.

$$\begin{array}{|l} SS \\ \hline blocks : Name \pfun Block \\ newnames : \mathbb{F}\ Name \\ \hline newnames \cap dom\ blocks = \{\} \\ NullName \notin newnames \\ \hline \end{array}$$

The service guarantees never to issue the special name NullName; this name can therefore be used by clients' applications to indicate "no block" (similarly to the use of the *nil* pointer in a programming language).

There are eight operations the client may ask the service to perform:

Submit	-	create a new block and store it.
Read	-	read the data of a stored block.
Status	-	obtain the complete status of a stored block.
Destroy	-	remove a stored block from the service.
Replace	-	replace one stored block with another.
Extend	-	change the lifetime of a block.
BlockNames	-	obtain the names of blocks currently owned by the client.
BlockCount	-	obtain the number of blocks currently owned by the client.

There is also a Scavenge operation which the service may perform at any time: it can't, however, be requested by clients:

Scavenge	-	remove an expired block.

Every operation the service can perform for a client must receive the client's identification as input, and it must provide the cost and a report as output; normally, the report will be Success:

```
client?: User

cost!  : Money
report!: Report
```

The client's username must be authentic (see Chapter 2) ; if it is, he will have a nickname:

```
nickname: Nickname

nickname = GetNickneme (client?)
```

During an operation, the service can ask the time service (Chapter 3) for the current time:

```
now?: Time
```

Finally, any name issued by an operation is removed from the set of new names, and so can never be issued again:

```
newnames' = newnames - dom blocks'
```

This schema describes these general aspects of operations on the storage service.

```
ΔSS ──────────────────────────────
  SS
  SS'

  client?     : User
  now?        : Time

  nickname    : Nickname

  cost!       : Money
  report!     : Report
──────────────────────────────────
  nickname  = GetNickname (client?)

  newnames' = newnames - dom blocks'
──────────────────────────────────
```

5.2 Accounting

Each client will be responsible for the expense of his using the service.

Costs

Every operation has a cost , which may have two components. One is the expense of performing the operation itself:

SubmitCost
ReadCost
StatusCost
DestroyCost
ReplaceCost
ExtendCost
BlockNamesCost
BlockCountCost

The other, if present, is related to the service requested by the operation. For example, the submit operation charges in advance for the storage of the block submitted, and the destroy operation may give a rebate (negative expense) if the block is destroyed before its expiry time.

The expense of storing a block is determined by applying a *tariff* function to the block's creation and destruction times. This is a typical tariff function:

Tariff (created, destroyed) =

if destroyed ⩾ created **then** destroyed - created
else 0

Accounting policy

The values of SubmitCost etc. and of the tariff function may be varied; their precise values at any time will be published separately. Expenditure will be recorded in a log, and clients will be expected to observe any limits placed upon them.

5.3 Security

The service provides limited security in two areas; in both cases it depends on certain values being chosen from such a large set that they are hard to guess.

A client may not access a block unless he knows its name, and block names are hard to guess. The name of any block is initially known only to its creator; the service will never tell any client the name of a block he doesn't own.

Blocks may be replaced or destroyed only by their owners, and user names are hard to guess (see Chapter 2).

5.4 Storage service operations

On the following pages appear descriptions of the storage service operations. Each description has three sections, titled **Abstract, Definition** and **Reports.**

The Abstract section gives a procedure heading for the operation, with formal parameters, as it might appear in a programming language. The correspondence between this procedure, and an implementation of it in a real programming language, must be obvious and direct.

Each formal parameter is given a name ending with either ? or !. Those ending with ? are inputs, and those ending with ! are outputs.

A short description accompanies the procedure heading.

The Definition section mathematically defines the operation, by giving a schema which includes as a component every formal parameter of the procedure heading; within the schema also appears a subschema (ΔSS) whose components include the service state before (SS) and after (SS') the operation. Any other components appearing in the schema are local to the operation (that is, temporary) and may assume any values consistent with the predicates.

The client is directly aware only of the components which are formal parameters of the procedure heading.

A short description accompanies the schema.

The Reports section lists the possible (success or failure reporting) values which the report! formal parameter may assume. If such a value is followed by a predicate, it is to suggest that the value would occur only if the predicate were true. The predicate is therefore a hint to the cause of the report.

Section 5.5 discusses report values in more detail by giving a mathematical definition of each of their occurrences.

SUBMIT

Abstract

```
Submit ( client?   : User;

         lifetime?: Time;
         data?     : Data;

         name!     : Name;

         cost!     : Money;
         report!   : Report)
```

A block is formed from the lifetime? and data? values given, and is stored by the service under the new name name!. The cost includes the expense of storing the block until its expiry time.

Definition

```
Submit
  ΔSS
  Block'
  lifetime?: Time
  data?    : Data
  name!    : Name
  ─────────────────
  owner'   = nickname
  created' = now?
  expires' = created' + lifetime?
  data'    = data?

  name!   ∈ newnames
  blocks' = blocks ⊕ [name! ↦ Block']

  cost! = SubmitCost + Tariff (created', expires')
```

The owner of the block is the client submitting it.

A new name is chosen which has never before been issued, and the new block is stored under that name.

Reports

```
Success
ServiceError

NoSpace      ⇒   #blocks = StorageCapacity
```

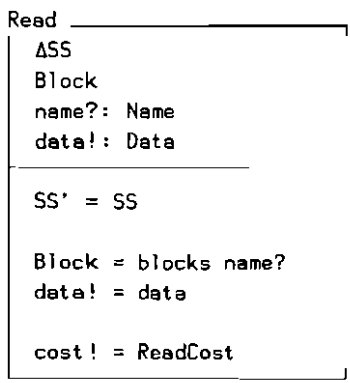
READ

Abstract

```
Read (client?: User;
      name?  : Name;
      data!  : Data;
      cost!  : Money;
      report!: Report)
```

The data is returned for the block stored under name?.

Definition



The service is unchanged by this operation.

Reports

Success
ServiceError

NoSuchBlock ⇒ name? ∉ dom blocks

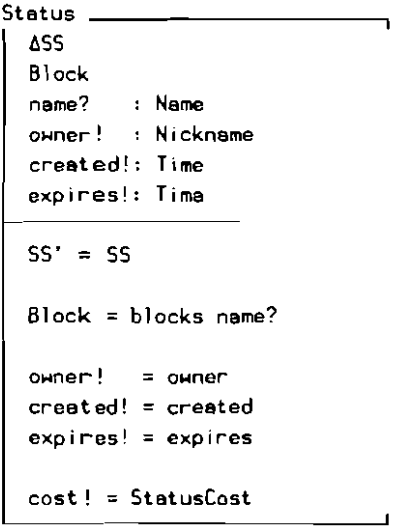
STATUS

Abstract

```
Status (client? : User;

        name?   : Name;

        owner!  : Nickname;
        created!: Time;
        expires!: Time;

        cost!   : Money;
        report! : Report)
```

The status is returned of the block stored under name?.

Definition



The service is unchanged by this operation.

Reports

Success
ServiceError

NoSuchBlock ⇒ name? ∉ dom blocks

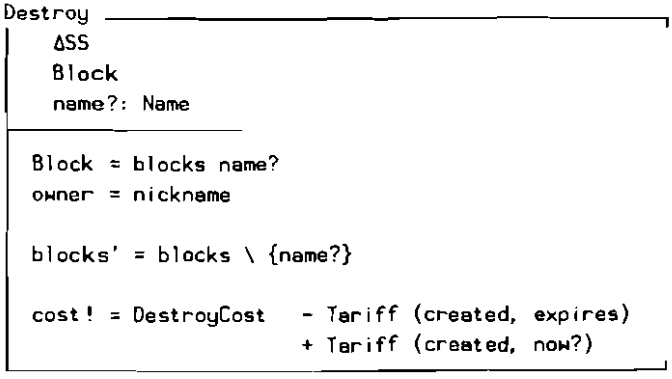
DESTROY

Abstract

```
Destroy (client?: User;

         name?  : Name;

         cost!  : Money;
         report!: Report)
```

The block stored under name? is removed from the service; there may be a rebate if it is destroyed before its expiry time.

Definition



A block may be destroyed only by its owner.

Reports

Success
ServiceError

NotOwner ⇒ owner ≠ nickname
NoSuchBlock ⇒ name? ∉ dom blocks

REPLACE

Abstract

```
Replace (client?: User;

         name?   : Name;
         data?   : Data;

         name!   : Name;

         cost!   : Money;
         report!: Report)
```

The data part of the block stored under name? is replaced by data?. *The block is given a new name.*

Definition

```
Replace
  ΔSS
  Block
  Block'
  name?: Name
  data?: Data
  name!: Name
  ─────────────────────────────────────
  Block = blocks name?
  owner = nickname

  owner'   = owner
  created' = now?
  expires' = expires
  data'    = data?

  name!   ∈ newnames
  blocks' = blocks \ {name?} ⊕ [name! ↦ Block']

  cost! = ReplaceCost   - Tariff (created , expires)
                        + Tariff (created , now?)
                        + Tariff (created', expires')
```

A block may be replaced only by its owner. The new block contains the new data, and its creation time is the time of the replace operation. Its owner and expiry are taken from the old block.
A new name is chosen which has never before been issued, and the new block is stored under that name.

Reports

Success
ServiceError

NotOwner ⇒ owner ≠ nickname
NoSuchBlock ⇒ name? ∉ dom blocks

EXTEND

Abstract

```
Extend (client?  : User;

        name?    : Name;
        lifetime?: Time

        cost!    : Money;
        report!  : Report)
```

The expiry time of the block stored under name? is changed to now? + lifetime?; there may be a rebate if its new expiry time is earlier than before.

Definition

```
Extend
  ΔSS
  Block
  Block'
  name?     : Name
  lifetime?: Time
──────────────────
  Block = blocks name?

  owner     = nickname

  owner'    = owner
  created'  = created
  expires'  = now? + lifetime?
  data'     = data

  blocks' = blocks ⊕ [name? ↦ Block']

  cost! = ExtendCost     - Tariff (created , expires)
                         + Tariff (created', expires')
```

A block may be extended only by its owner.

Reports

```
Success
ServiceError

NoSuchBlock     ⇒   name? ∉ dom blocks
NotOwner        ⇒   owner ≠ nickname
```

BLOCKNAMES

Abstract

```
BlockNames (client? : User;

            key?    : Key;
            count?  : N

            key!    : Key;
            nameset!: F Name

            cost!   : Money;
            report! : Report)
```

BlockNames returns a (finite) set of names of blocks owned by the client.

Since a client may own many blocks, it may not be practical in a single operation to return all of their names:

$$allnames = blocks^{-1} \circ owner^{-1}\ (\!|\{nickname\}|\!)$$

The operation BlockNames therefore returns only a part of allnames, and repeated calls of it may be necessary to construct allnemes as the union of the parts returned.

To construct allnames, the client first calls BlockNames with a special key StartKey:

```
BlockNames |   key? = StartKey
```

He then continues to call BlockNames repeatedly, supplying as the new key (key?) in each case the key (key!) returned by the previous call. The following is an example of the i^{th} call:

```
BlockNames |   key?     = key_i
               key!     = key_i+1
               nameset! = nameset_i
```

Finally, the special key EndKey will be returned to indicate that no more calls need be made.

```
BlockNames |   key! = EndKey
```

At that point, providing allnames has remained constant (*i.e.* no submits *etc.* have occurred for this client),

$$\text{allnames} = \bigcup_i \text{nameset}_i$$

Definition

BlockNames

ΔSS

key? : Key
count? : $\mathbb{N}$

key! : Key
nameset! : $\mathbb{F}$ Name

allnames : $\mathbb{F}$ Name

$SS' = SS$

$allnames = blocks^{-1} \fatsemi owner^{-1} (\!|\{nickname\}|\!)$

$key? \subseteq key!$
$nameset! = (key! - key?) \cap allnames$

$\#nameset! \leq count?$

count? limits the size of the set returned.

A key is itself to be regarded as a set of names; *i.e.* the set of keys is the (finite) set of all such sets of names

$$Key \mathrel{\widehat=} \mathbb{F}\,\mathbb{F}\,Name$$

Each key value, passed from one call to the next, includes all the names that have been returned (and possibly some that have not, but never will be).

The special keys are

```
StartKey, EndKey: Key

StartKey ≙ {}
EndKey   ≙ Name
```

Reports

```
Success
ServiceError
```

BLOCKCOUNT

Abstract

```
BlockCount (client?: User;
            count! : N;
            cost!  : Money;
            report!: Report)
```

An estimate (upper bound) is returned for the number of blocks currently owned by client?.

Definition

BlockCount

ΔSS

count! : $\mathbb{N}$

allnames: $\mathbb{F}$ Name

$SS' = SS$

$allnames = blocks^{-1} \circ owner^{-1} (\!|\{nickname\}|\!)$

$count! \geq \#allnames$

The count returned is an upper bound (rather than an exact value) because it may include blocks which have been scavenged since the last initialisation of the service.

This operation may be unavailable if it was not enabled at initialisation of the service.

Reports

Success
ServiceError

CountNotAvailable

SCAVENGE

Abstract

Scavenge (name?: Name)

The block stored under name? is removed from the service; only expired blocks may be scavenged.

Scavenge may be invoked by the service at any time; it can never be invoked by clients.

Definition

Scavenge

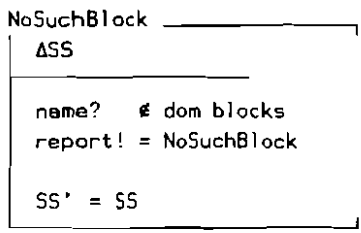
```
ΔSS
Block
name?  : Name
─────────────────────
Block = blocks name?

expires < now?

blocks' = blocks \ {name?}
```

5.5 Error reports

The report! parameter of each operation indicates either that the operation succeeded or suggests why it failed. In most cases, failure leaves the service unchanged.

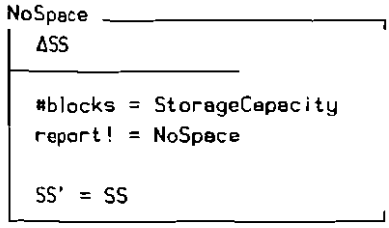
An operation can return only the report values listed in the Reports section of its definition. If it returns the value Success, it must satisfy its defining schema. If it returns any other value, it must satisfy instead the appropriate schema below.

NoSuchBlock



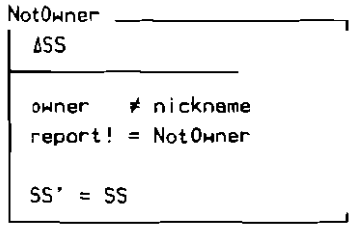
This report is given if there is no block stored under name?; note that this may be because the block has been scavenged.

NoSpace



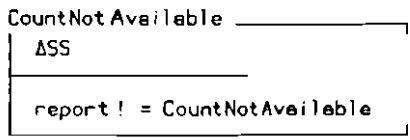
A new block cannot be submitted when the service's storage capacity is exhausted.

NotOwner



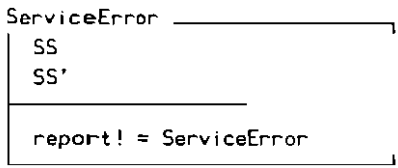
Operations which can remove or change a block must be performed by the block's owner only (excepting Scavenge).

CountNotAvaliable



If the BlockCount operation was not enabled at service initialisation, this may be the result of an attempt to invoke it.

ServiceError



ServiceError indicates an unexpected failure which is probably not the client's fault.

These are typical causes:

Network (hardware or protocol) failure
Service hardware fault (e.g. disk error)
Service software error

5.6 UCSD Pascal Interface

```
UNIT SI;

INTERFACE {UNIT SI  10-Aug-84}

{Block Storage Service  -  UCSD-Pascal Interface}

  USES {$U SVTYPES.CODE} SV_Types;


CONST
  SI_DataSize  = 528;  {data bytes per block}
  SI_NameLimit =  64;  {max returnable names per call of BlockNames}

TYPE
  SI_Name    = SV_16HEX;
    {block name}

  SI_Data    = PACKED ARRAY [1..SI_DataSize] OF SV_Byte;
    {block data}

  SI_Key     = PACKED ARRAY [1..4] OF SV_Byte;
    {key used to chain BlockNames calls}

  SI_NameSet = RECORD
                 count : INTEGER;
                 names : ARRAY [1..SI_NameLimit] OF SI_Name
               END;
    {set i.e. names[1]..names[count] returned by BlockNames}

  SI_Report = (SI_Success,
               SI_ServiceError,
               SI_NotOwner,
               SI_NoBlocksLeft,
               SI_NoSuchBlock,
               SI_CountNotAvailable);
```

```
VAR
  SI_NullName : SI_Name;  {neme of no block}

  SI_StartKey : SI_Key;   {given to first call of BlockNames}
  SI_EndKey   : SI_Key;   {returned by final cell of BlockNames}


PROCEDURE SI_Submit  (    InClient   : SV_User;
                          InLifetime : SV_Intervel;
                      VAR InData     : SI_Data;
                      VAR OutName    : SI_Name;
                      VAR OutCost    : SV_Money;
                      VAR OutReport  : SI_Report);
  {store given data in a block, returning its name}

PROCEDURE SI_Read    (    InClient   : SV_User;
                          InName     : SI_Name;
                      VAR OutData    : SI_Data;
                      VAR OutCost    : SV_Money;
                      VAR OutReport  : SI_Report);
  {read data of named block}

PROCEDURE SI_Destroy (    InClient   : SV_User;
                          InName     : SI_Name;
                      VAR OutCost    : SV_Money;
                      VAR OutReport  : SI_Report);
  {destroy named block}

PROCEDURE SI_Replace (    InClient   : SV_User;
                          InName     : SI_Name;
                      VAR InData     : SI_Data;
                      VAR OutName    : SI_Name;
                      VAR OutCost    : SV_Money;
                      VAR OutReport  : SI_Report);
  {gives effect of destroy then submit}
```

```
PROCEDURE SI_Stetus  (    InClient    : SV_User;
                          InName      : SI_Name;
                      VAR OutOwner    : SV_Nickname;
                      VAR OutCreated  : SV_Time;
                      VAR OutExpires  : SV_Time;
                      VAR OutCost     : SV_Money;
                      VAR OutReport   : SI_Report);
  {return attributes of named block}

PROCEDURE SI_Extend  (    InClient    : SV_User;
                          InName      : SI_Name;
                          InLifetime  : SV_Interval;
                      VAR OutCost     : SV_Money;
                      VAR OutReport   : SI_Report);
  {change the lifetime of the named block}

PROCEDURE SI_BlockNames (    InClient  : SV_User;
                             InKey     : SI_Key;
                             InCount   : INTEGER;
                         VAR OutKey    : SI_Key;
                         VAR OutNames  : SI_NameSet;
                         VAR OutCost   : SV_Money;
                         VAR OutReport : SI_Report);
  {starting from given key, return up to given number of names
   belonging to this client, plus new key to obtain further names}

PROCEDURE SI_BlockCount (    InClient  : SV_User;
                         VAR OutCount  : SV_10INT;
                         VAR OutCost   : SV_Money;
                         VAR OutReport : SI_Report);
  {if available, return the number of blocks owned by this client}
```

5.7 Modula-2 Interface

```
DEFINITION MODULE SI;  (* Roger Gimson  22-Feb-84 *)

(* Block Storage Service  -  Modula-2 Interface *)

FROM SVTypes IMPORT User, Nickname, Time, Interval, Money;

IMPORT Long;

EXPORT QUALIFIED DataSize, NameLimit,
                 Name, Data, Key, NameSet, Report,
                 NullName, StartKey, EndKey,
                 Submit, Reed, Destroy, Replace, Status, Extend,
                 BlockNames, BlockCount;

CONST
  DataSize  = 528;  (* data bytes per block *)
  NameLimit =  64;  (* max returnable names per call of BlockNames *)

TYPE
  Name      = ARRAY [1..4] OF CARDINAL;       (* block name *)
  Data      = ARRAY [0..DataSize-1] OF CHAR;  (* block data *)

  Key       = ARRAY [1..2] OF CARDINAL;
    (* for chaining BlockNames calls *)
  NameSet   = RECORD
                count: CARDINAL;
                names: ARRAY [1..NameLimit] OF Name;
              END;
    (* set i.e. names[1]..names[count] returned by BlockNames *)

  Report    = (Success,
               ServiceError,
               NotOwner,
               NoBlocksLeft,
               NoSuchBlock,
               CountNotAvailable);
```

```
VAR
  NullName : Name;  (* name of no block *)
  StartKey : Key;   (* given to first call of BlockNames *)
  EndKey   : Key;   (* returned by final call of BlockNames *)


PROCEDURE Submit  (    InClient   : User;
                   VAR InLifetime : Interval;
                   VAR InData     : Data;
                   VAR OutName    : Name;
                   VAR OutCost    : Money;
                   VAR OutReport  : Report);
  (* store given data in a block, returning its name *)

PROCEDURE Read    (    InClient   : User;
                   VAR InName     : Name;
                   VAR OutData    : Data;
                   VAR OutCost    : Money;
                   VAR OutReport  : Report);
  (* read data of named block *)

PROCEDURE Destroy (    InClient   : User;
                   VAR InName     : Name;
                   VAR OutCost    : Money;
                   VAR OutReport  : Report);
  (* destroy named block *)

PROCEDURE Replace (    InClient   : User;
                   VAR InName     : Name;
                   VAR InData     : Data;
                   VAR OutName    : Name;
                   VAR OutCost    : Money;
                   VAR OutReport  : Report);
  (* give effect of destroy then submit *)
```

```
PROCEDURE Status  (    InClient    : User;
                   VAR InName      : Name;
                   VAR OutOwner    : Nickname;
                   VAR OutCreated  : Time;
                   VAR OutExpires  : Time;
                   VAR OutCost     : Money;
                   VAR OutReport   : Report);
  (* return attributes of named block *)

PROCEDURE Extend  (    InClient    : User;
                   VAR InName      : Name;
                   VAR InLifetime  : Interval;
                   VAR OutCost     : Money;
                   VAR OutReport   : Report);
  (* change the lifetime of the named block *)

PROCEDURE BlockNames (     InClient   : User;
                           InKey      : Key;
                           InCount    : CARDINAL;
                       VAR OutKey     : Key;
                       VAR OutNames   : NameSet;
                       VAR OutCost    : Money;
                       VAR OutReport  : Report);
  (* starting from the given key, return up to given number of names
     belonging to this client, plus new key to obtain further names *)

PROCEDURE BlockCount (     InClient   : User;
                       VAR OutCount   : Long.Integer;
                       VAR OutCost    : Money;
                       VAR OutReport  : Report);
  (* if available, return the number of blocks owned by this client *)

END SI.
```