A Shared, Persistent Object Store

Colin Low

Department of Computer Science,
Queen Mary College,
Mile End Rd, London E1 4NS.

ABSTRACT

Smalltalk-80 is presented as a useful testbed for prototyping applications involving shared, persistent objects, and a detailed design of a shared persistent object store is discussed. The store is a set of named containers for object state, and it provides low-cost atomic transactions using an optimistic synchronisation technique. The standard Smalltalk-80 virtual machine is modified to support a new object class, the **Transaction**, and an example of a Smalltalk program using nested sub-transactions is given. Immutability of object state is identified both as an important property of objects, and a basis for producing an efficient implementation within a distributed system environment.

Keywords and phrases: object-oriented programming, distributed systems, atomic transactions, Smalltalk-80, persistence, immutable objects.

Many die too late, and some die too soon. Still the doctrine sounds strange: "Die at the right time".

Nietzsche

1. Introduction

Programmers have been using one particular type of shared, persistent object for years: this object type has many guises, but it is normally called a **file**. Files are the donkeys of persistent programming; anything and everything that will fit is perched on the backs of these poor animals. The shared, persistent object is a natural generalisation of a concept of which a file is the most common example.

The terms are loosely defined as follows. An object is a container for a value of some kind. The object is associated with a set of operations for observing, and possibly changing that value. It is possible to create new objects with an initial value. It must be possible to reference a particular object in order to apply one of it's operations, and so objects are named. Objects may cease to exist; it is possible to have referential failure.

An object is persistent if it "dies at the right time"; persistence is an observation about the lifetime of an object, namely that it exists for precisely as long as intended, and then it disappears. There are various ways of deciding the right time, but generally speaking a programmer or user demands that an object disappears according to some predetermined criteria, and only then. It should not cease to exist as a

result of a processor crash, or a bad block on a disc drive, or some other predictable failure mode. An object can be persistent without being shared, and shared without being persistent; the problems are orthogonal.

The work described in this paper grew out of the author's interest in the programming of distributed applications. A large proportion of such applications involve objects which are shared and which are persistent. One approach is to use shared files held on public file servers such as the SUN NFS file servers [24] used at Queen Mary College. There are two problems however. The first is that files are the *only* persistent objects provided in the author's environment. This is a nuisance but it can be circumvented by using files to contain the representation of different types of object using ad-hoc conventions. The second problem is that the commonly available programming languages, such as C, Pascal, Modula 2 etc. do not provide the programmer with persistent objects. The data types that represent objects are held in volatile memory, and the state of an object disappears along with the process that created it. The programmer has almost no control over the lifetimes of objects created within the programming language environment. The normal solution is to devise a scheme for saving the state of an object in a file, and to read it back into volatile memory again at an appropriate time. A substantial amount of programming effort is required to map between volatile and non-volatile representations of an object's state. This is clumsy and a poor use of a programmer's time; the advantages of integrating persistence into programming languages have been discussed by a number of authors, for example Atkinson *et al* [2].

If sharing of objects is involved then the problems are compounded. A realistic solution to the problem of sharing is complex if several objects are involved, integrity constraints exist, and concurrency is to be maximised; reviews of work in this area have been carried out by Kohler [15], and Bernstein and Goodman [5], and we return to the problem later in this paper. It is unreasonable for every programmer to have to re-invent a solution to the dual problems of persistency and sharing, and one answer is to use a programming language that supports the use of shared, persistent objects as naturally and transparently as possible.

The choice of Smalltalk-80 [14] as the host language for experimenting with distributed applications was motivated primarily by the simplicity and expressive power of the language. It is an excellent testbed for trying out new ideas, and the language is embedded in an interactive programming environment that provides support for sophisticated user interface design. The addition of shared, persistent objects to Smalltalk makes it possible to build on the existing and extensive class libraries in the construction of working prototypes; it is possible to experiment with user interfaces to shared objects without the systems programming expertise that is usually needed for this type of work. Much of the language is defined at a level where it is easy to modify; for example, in the programming of distributed applications the ability to handle many kinds of exception can be important, and Smalltalk provides the building blocks for implementing exception handling.

There were other reasons for choosing Smalltalk. It is portable and runs on most of the workstations in the author's environment. The virtual machine that provides run-time support is described in detail [14], and an efficient high-level language implementation is available locally [21].

It is important that any modifications to the Smalltalk-80 system preserve the reactive quality of existing Smalltalk-80 implementations; this is a severe constraint that has guided much of the design.

The system described is tightly coupled to Smalltalk-80, but many of the techniques used have wide applicability. It can be regarded as an instance of the application of distributed system techniques to a particular problem; multiple Smalltalk-80 client virtual machines sharing a persistent object storage service.

Sections 3 of this paper discusses how Smalltalk-80 objects are represented in object memory. It shows how it is possible to remove part of the object memory to a remote server, the persistent object store how persistent objects are loaded into volatile object memory on demand. Section 4 describes the optimistic concurrency control algorithm. Section 5 outlines the importance of immutable objects for improving performance. Section 6 discusses new object classes and extensions to the Object protocol. Section 7 is a description of a Smalltalk program in Appendix 1. that shows the use of nested transactions for modifying a shared, persistent object.

2. The Environment

The environment assumed is a typical distributed system such as described in [12]: a number of client graphics workstations interconnected by a fast local area network. Shared services such as filing, naming, authentication etc. are provided by dedicated server systems, and a fast, lightweight remote procedure calling mechanism is used for client-server communication. Users wishing to use Smalltalk will run it on a client workstation; a number of dedicated servers are used to implement the shared persistent object store. This is shown in Figure 1. below. Each server has a significant amount of volatile semiconductor storage and a large amount of slower non-volatile storage (e.g. disc drives).

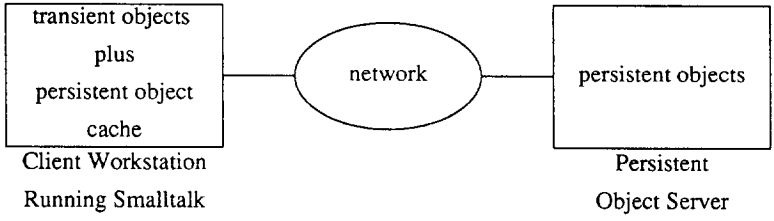

Figure 1: The Client-Server Relationship

There can be several client systems using persistent objects, and the objects themselves can be spread across several persistent object stores.

3. Object Memory

Smalltalk-80 objects must be represented in memory of some kind. Normally this is the volatile semiconductor memory in a workstation. This section describes the abstract relationships of objects and shows that it is possible to spilt the object space between different physical memories, some volatile, and some non-volatile. This gives rise to two types of object: transient objects whose death is unimportant, and persistent objects which satisfy guarantees about their lifetimes.

3.1. The Object Space

A Smalltalk-80 system contains a large number of objects. Objects reference oneanother. Even though all objects reference a class object, they can be split into two major categories: some objects *only* reference a class object and correspond to primitive data types, such as symbols or floating point numbers. The other category of objects directly reference one or more other objects in addition to the class reference. The first type of object I will refer to as *primitive*, while the second type will be referred to as *composite*.

A composite object *directly depends* on those objects it references (including its class). A primitive object directly depends only on its class. An object *indirectly depends* on any object that can be reached through the chain of direct dependencies. The direct dependency relationship between objects structures the object space into a directed graph. This is shown in Figure 2. below; the object at the root of the graph directly depends on every object to which it is directly connected with an arrow, and indirectly depends on every object beneath it.

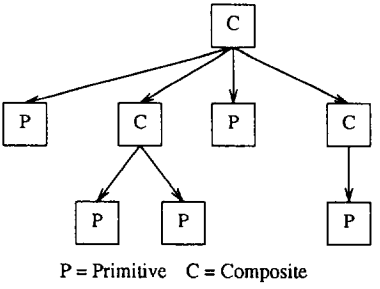

Figure 2: Object Dependencies

A chain of direct dependencies is terminated as follows:

- small integers directly depend on nothing.
- primitive objects directly depend only on their class.
- the **nil** object depends only on its class.
- the class **Object** (from which all classes are derived) depends on nothing.

Indirect dependency is important; an object of type **Class** directly depends on its method dictionary, and indirectly depends on method texts and bytecodes. Change a method and the behaviour of a class changes. Indirect dependency, and the consequent set of objects so related to a given object, reflects the way in which the programmer builds complex objects and behaviour out of simple components; it is an important concept when considering persistent objects.

An object is a container for a value; for example, a tuple of references to other objects, and in general that value may change in time as the operations associated with an object modify it. The object behaves like a finite state machine, beginning in an initial state, and moving through a succession of new states. The *total* state of an object is the collective state of every object on which it indirectly depends. Like indirect dependency, the total state of an object is important because objects are not normally used in isolation; they are meaningfully related by the programmer into complex structures.

Object names in Smalltalk-80 have only local significance within a given instance of the Smalltalk-80 virtual machine. They can be 16 or 32 bit quantities depending on the implementation.

3.2. The Smalltalk-80 Object Memory

The Smalltalk-80 object memory is the repository for all of the object state in a conventional Smalltalk system. It provides the following functionality:

- a set of named containers for object state.
- a set of operations on containers. The normal operations on containers are **create** a new container, **put** a value in a container, **get** a value from a container, and **delete** a container.

The actual interface differs in detail. There is no delete; object deletion is handled implicitly. The object state is divided into parts, with explicit operations for putting or getting individual components of the object state. Everything in Smalltalk is an object, so with minor exceptions (e.g. small integers) the representation of objects is simple and uniform. The visible components of object state are the **class**, the **size**, and the **instance variables**. The size refers to the number of instance variables. The instance variables can either be used as a tuple of object references, corresponding to a composite object, or as numeric values representing, for example, floating point numbers, long integers, strings etc, which correspond to a primitive object.

Object names, or references, can either be memory addresses, or more commonly, integer indices into an Object Table; these names have no significance outside of the Smalltalk-80 virtual machine. Smalltalk implementations provide persistent objects by saving and loading much of the state of the Object Memory, but because object names retain their local significance sharing is not possible. This procedure is manual; a snapshot of the object memory is written to disc and so the object memory can be reconstructed at a later time, but it is an all-or-nothing procedure. It is not possible to select one object and access it from another different virtual machine.

3.3. Persistent Objects

The first step in providing shareable, persistent objects is to provide objects with global, persistent names so that the same object can be referenced from anywhere at any time using the same name. However, not all objects need to be persistent (for example, an instance of the browser), and a large proportion of the objects in a running Smalltalk system could be transient even if persistence was available. Fast creation and manipulation of objects is too important to be compromised. A second consideration is that the number of global persistent objects that a user could reference is potentially enormous; persistent object names could be larger than ordinary object names, particularly if they are viewed as capabilities and sparsely embedded within a range of values. This is a strong argument in favour of having a two-level naming scheme. The existing Smalltalk Object Memory interface is retained, along with the existing object names, and the conversion between local and persistent object names is carried out transparently within the Object Memory. This approach is modular; a large proportion of the virtual machine remains unchanged.

If an arbitrary Smalltalk object is to be made persistent then every object on which it indirectly depends must also be persistent, otherwise there will be dangling references in the persistent store. This is a strong requirement; every object references a class, so this implies that every class referenced by a persistent object must also be global and persistent. In turn this implies that the class methods are persistent, as are all the literal symbols referenced by the methods.

There must be at least one object from which the persistency of every other object depends, and the solution chosen was to make the global directory **Smalltalk** a global persistent object. Any object reachable from **Smalltalk** is global to every virtual machine. Any object not reachable from **Smalltalk** is transient.

A two-level naming scheme, with the local Object Memory translating between local and non-local references is like the LOOM system [16]. The LOOM system was designed to overcome address space restrictions in workstations at the time, and provided an object virtual address space very much larger than that possible in a memory resident system. It was essentially an "object paging" system, with non memory resident objects being paged in whenever an "object fault" occured. The LOOM system did not separate out transient and persistent objects, nor was sharing of objects possible, but some of the algorithms used have been adopted, as described in 3.5.

3.4. The Persistent Object Store

The persistent object store, like the object memory, is a collection of named containers for object state. An object name is referred to as a *persistent object identifier* or PID. A PID can be decomposed into two parts: a partition of the total object space, and an object within a partition. A partition is synonymous with a server; persistent objects can be spread amongst several servers. The basic operations provided are:

CreatePSObject: ( ) → PID
Create a new persistent object in the null state and return its PID.

GetPSObject: PID → ObjectState
Get the current state of the specified object.

PutPSObject: PID × ObjectState
Assign a new state to the specified object.

SwapPid: PID × PID
Swaps the state associated with a PID so that the state associated with the first PID becomes the state associated with the second PID and vice-versa.

SetPSNil: PID
Store the PID of the object that represents the undefined object **nil**.

GetPSNil: ( ) → PID
Get the PID of the **nil** object.

SetPSRoot: PID
Store the PID of the root object.

GetPSRoot: ( ) → PID
Get the PID of the root object.

An object is created in a null state because of objects that refer to themselves; unless the PID of an object is known, the value of the object (which contains a reference to itself) cannot be assigned. **SwapPid** is necessary to support the primitive method **become**. One of the important uses of **become** is to grow collection classes. There are two special objects that must be individually accessible. **SetPSRoot** makes public the PID of the root object via **SetPSRoot**. **SetPSNil** and **GetPSNil** do the same for the **nil** object. Every persistent object is a component of the total state of the root object and is indirectly related to it. There is no difficulty in having several root objects, but in this implementation there is only one, the global directory **Smalltalk**.

The object state can be decomposed into components. These are

class: the PID of the object that is the class of this object.

hash: a unique value permanently associated with this object.

timestamp: the (pseudo)time of the last write to the object.

size: the number of components of a given type making up the instance variables (see **type**).

type: indicates whether the instance variables should be interpreted as object names (PID's), or as values of a simple type. (e.g. words, bytes, bits. See [14]).

immutable: a flag to indicate whether an object's instance variables are frozen (see below).

instanceVariables: there are **size** items of a type indicated by **type**.

All object references within the persistent object store are PID's.

3.5. Object Loading

Persistent objects are loaded into Smalltalk object memory on demand. The algorithm is very similar to, but simpler than, the algorithm used in LOOM [16]. The LOOM system was designed to overcome address space limitations, and some of its complexity is a result of trying to obtain satisfactory performance in the face of these limits. There are broad similarities to the persistent heap used with S-Algol [3] and the subsequent PS-Algol [8]. The following description assumes a familiarity with the implementation of the Smalltalk object memory described in [14].

When a persistent object is loaded into local object memory it must look like any other local object. Persistent objects only reference other persistent objects, so all the PID's that are part of the state of a persistent object must be transformed into local references. A reference to a persistent object (PID) is

resolved in three parts:

1. check to see if it is already loaded. A dictionary called **LoadedPIDs** is used to store an association between a PID and a local object pointer. If it is already loaded then the PID can be replaced by a local reference. If the object is not loaded then.....
2. a vacant object table slot is allocated; the index in the table is the local object reference or identifier. The persistent object now has a local reference that can be stored in the instance variables of other local objects. The association between the PID and the object identifier is added to **LoadedPIDs**. A flag is set in the object table entry to indicate that the object state has not been loaded. The state of the object is not loaded until.....
3. it receives a message. When an object receives a message its state is loaded. The **class** component, and the **instanceVariables** component of composite objects, contain PID's which must be transformed into local references by going back to step 1.

The partly loaded object in step 2 is identical to what in LOOM is called a *leaf*. Object table slots were at a premium in LOOM, and this was behind the decision to implement *lambdas*, unresolved PID's in local Object Memory, rather than turning every PID in an object into a leaf. Object table slots in 32 bit architectures are no longer at a premium, and *lambdas* were not implemented. The resulting algorithm is simple. Almost all the changes required are confined to the object memory. The object memory maps object pointers to object state; it must check that the state is loaded while doing this. This test must be made for every object access, but it is typically a couple of machine instructions to do this.

Adding support for persistent objects increases the overall size of object memory. The object memory keeps more administrative information about each persistent object than a non-persistent object, but careful design can minimise the increase.

An unexpected cost of persistence is the need to explicitly and permanently associate with every object a random value. All objects respond to the method **hash**, which is used, for example, in the class **Dictionary**. A **Dictionary** uses the hash value for a hash table lookup, and as it *could* be persistent, the hash values it uses must remain constant over its lifetime; this implies that objects, whether persistent or not, must return a hash key which is not calculated from transient information.

There are two separate garbage collectors. Persistent objects are garbage collected if they are inaccessible from the root object **Smalltalk**. At present this is an off-line activity. The local Object Memory maintains its own garbage collector and discards objects which are not currently referenced within the Object Memory; discarding a persistent object has no global effect. Persistent objects may be discarded from local memory at any time as part of LRU space recovery.

If the state of a persistent object is changed, then its new state should be reflected in the persistent object store at some point. Even when the object store is not shared this is not a trivial problem. The simplest method is to carry out an immediate write-through whenever a change to a persistent object is made. This is not usually what is wanted. If a message is sent to an object, it may send further messages to objects on which it is directly dependent, as may the objects which receive the messages. It is the change to the *total state* of the object which is of interest, not just the individual changes to each object. We want *all* of the

changes to be done or *none* of the changes to be done.

As an example, I may write a program to iterate over a persistent **Set** of numbers, adding 1 to each number in the set. I either want all of the numbers to be incremented, or none of them; if there is a crash I do not want a situation where I am unable to tell how many of the elements have been incremented.

It is reasonable to expect the total state of an object to change consistently to new states. This all-or-nothing property of changes in the face of failure and concurrency is familiar in database systems where it is referred to as *atomicity*. Atomicity is composed of two properties:

> **recoverability**: integrity constraints on data are preserved in the face of crashes and other unexpected events.
>
> **serialisability**: integrity constraints on data are preserved in the face of multiple, concurrent sets of changes, and the effect of concurrents sets of changes is the same as if they had been executed consecutively.

The requirement that the total state of an object changes consistently to a new total state means that even in the absense of sharing we need recoverability. The next section shows how both serialisability and recoverability are provided using atomic transactions.

4. Concurrent Access

The persistent object store is designed to be shared by many users. Sharing introduces the traditional problems of concurrency control encountered in shared databases. It is important to make some assumptions about the environment in which objects are shared, for without assumptions one is forced to make pessimistic design decisions.

1. The persistent object store is not intended to be used as an object-oriented database. It *is* a database, but lacks the sophistication and optimisation of special purpose database systems. An examination of what additional facilities might be needed has been made by Bloom and Zdonik [7].
2. It is possible to make the *optimistic* assumption that, because the number of objects is very large, the probability that any two sets of concurrent changes intersect, is small.

It is the author's belief that such a system is useful. There are many applications in a distributed system in which sharing is required, but not at the level of commercial database systems. At QMC I use various shared databases for such things as telephone numbers, reports, student applications, mail and electronic bulletin boards, purchases, authentication and authorisation, and so on. I also share documents, programs and program sources. This sharing is implemented using UNIX files and simple file locking. It works because the optimistic assumption holds. It is unreasonable to implement every instance of sharing as if it was a no-holds-barred state-of-the-art transaction processing system. The system described makes no compromises; it provides serialisability and recoverability, but uses the optimistic assumption to provide them cheaply and efficiently.

Each Smalltalk object memory can function as a large cache of persistent objects. Reading the state of an object is cheap, because it is local, and it was decided to use a concurrency control algorithm that takes advantage of this cache. Bernstein and Goodman [5] have analysed a large number of algorithms and

concluded that all are variations on two basic techniques: 2-phase locking (2PL), and time-stamp ordering (TSO). 2PL with time-out based deadlock prevention, such as used in Violet [13], is essentially a pessimistic technique. It is valuable in an environment where conflicts between transactions are common. Under such circumstances it is better to use an algorithm that enforces serialisation through waiting rather than restarting. A number of TSO schemes were considered and two which seem most appropriate are optimistic algorithms proposed by Thomas [25], and Kung and Robinson [17]. The two algorithms use the same principles, but differ in detail; Thomas's algorithm associates a timestamp with each item of data, while Kung and Robinson do not, at the expense of a more complex validating procedure. Both algorithms can be used, but Thomas's variation was chosen because it allows each Smalltalk virtual machine to use its cache of persistent objects very efficiently.

The solution adopted is a conventional one in which the programmer brackets access to persistent objects inside a **Transaction**. Transactions are atomic, and may span several persistent object stores. Transactions may also be nested inside oneanother. A transaction may involve reading and writing many persistent objects; this is done locally using the persistent object cache in the client workstation, and assuming the objects are present, does not require any external communication. A transaction adds very little processing overhead to the interactive cost of reading and writing persistent objects; the cost is incurred at commit time, when the read and write set of the transaction is *validated* by the persistent object stores involved. The outcome of validation is that the transaction is accepted at every store, and committed, or it is rejected at one or more, and aborted; the coordination of multi-server commits or aborts is carried out by an new external service called the **Transaction Manager** (TM).

Each persistent object is associated with a timestamp which is updated when a new value is assigned to the object. If an object is loaded into an object memory, it carries with it the timestamp current at the time it was loaded. A transaction records the sequence of persistent object references carried out in local object memory; for each reference it records the object, the operation (read or write) and the timestamp. This information is used by the validating algorithm in the persistent object store to decide whether to accept or reject a transaction. Each transaction consists of at least three, possibly four phases:

1. the read phase. This can last an indefinite period. A transaction which has entered its read phase is invisible. No external communication between the Smalltalk virtual machine and any other external agent are necessary during the read phase, unless of course, an object fault occurs and it is necessary to load a persistent object from a store. Any changes made to persistent objects are local and hence invisible.
2. the validation phase. The transaction becomes visible at the end of the read phase, when the Smalltalk virtual machine requests a new transaction number from the TM. The virtual machine then communicates with every object store involved, passing it the transaction number, the set of names (PID's) of objects read from that store, the set of objects written to in that store, and for each updated object, a new value. Each component of the transaction is separately validated by the relevant object store, and is either accepted or rejected. Each component of the transaction enters the *pending* stage. The Smalltalk virtual machine then sends to the TM the transaction number, and a list of the object stores involved in this transaction, and awaits the total result. The TM is

responsible for handling the usual 2-phase commitment protocol (see, for example Ceri and Pelagatti [11]). The TM communicates with each object store asking whether the component of the transaction handled in that store was accepted or rejected. If all are accepted, then the transaction as a whole is committed, if any are rejected then the transaction as a whole is aborted. The TM logs the result in stable store, and sets about informing every object store of the result. If the result is commit, then the component at each object store enters the write phase, otherwise the pending write is deleted.

3. the write phase. During the pending phase the new values for each object in the write set were written to secondary storage, *but the objects themselves were not changed.* For a period of time both the original object, and a new "shadow object" exist. When the object store is informed by the TM that it can complete the transaction each object is swapped with the "shadow object" that has been created, and when all the swaps have been carried out, the write phase is marked as complete. The write phase is irrevocable and non-interruptable. Only one transaction can be in the write phase in any object store. If a crash occurs, only one write phase could have been affected, and it is restarted at the beginning, over and over, until it is done. If a transaction is aborted, the object store is informed eventually and the shadow objects are garbage collected in the normal way.
4. the pre-write phase. This phase does not occur in every case. If several transactions are pending, and their write sets intersect, then they must enter the write phase in the order defined by their transaction numbers, otherwise inconsistencies will result. In this case the write phase of a transaction will be delayed, although it is guaranteed to be done eventually. A transaction in the pre-write phase has been committed, but it cannot enter the write phase until all earlier pending transactions have been resolved. If the write set of a transaction is a subset of the read set, then the transaction will never enter the pre-write phase, because the read set is known not to intersect the write set of any pending transaction.

It is important that the 2-phase commit mechanism runs separately from client workstations; 2-phase commit is an essential part of the algorithm for ensuring that updates preserve integrity constraints, and for this reason an independent service, the TM, is used to carry it out. It could have been built into the persistent object store, but it has nothing to do with object storage as such, and separating it out produces a clean division of responsibilities with the object stores in a subordinate relationship to the TM.

The validation of a transaction works as follows:

1. if the timestamp on each object in the read set is still current, the the transaction is accepted.
2. if the read set intersects the write set of any pending transaction then the transaction can either be rejected or the decision can be deferred. The transaction will be rejected if the transaction number of the pending transaction is more recent than that of the transaction being validated, otherwise the decision is deferred.

The purpose behind deferring a decision is the hope that a pending transaction which conflicts with the transaction being validated might be aborted, in which case there will no longer be a conflict. A transaction is split into components, one in each object store involved, and it is possible that a component of transaction T1 might be deferring on the outcome of transaction T2 in one object store, and the other way

around in a second object store. The result is a deadlock, and the way to avoid deadlock is to use the ordering on transaction numbers. For this reason an older transaction is not permitted to defer on the result of a younger transaction.

Despite the optimism of the concurrency control mechanism described above, the processing and communication overheads are not neglible. It is not as bad as it looks; only mutable persistent objects require concurrency control, and as will be described below, large numbers of persistent objects can be immutable. A major advantage of this method is that almost no overhead is incurred by the Smalltalk virtual machine until a transaction attempts to commit, and then all the overheads are incurred in one go. This is different from 2PL for example, where locks are acquired progressively. This means that it is possible to produce the same kind of reactive Smalltalk program as before; if a user is interacting with a persistent data-base, there are no unpredictable gaps in the interaction where the program fails to respond because slow concurrency control measures are being applied.

One problem, and a major one, is the problem of restarts. A transaction will be rejected and restarted if its read set has been invalidated, and the probability increases as the read set increases in both size and age. The bigger the read set, the more likely a conflict will occur. The older the read set, the more likely that a conflict has already occurred. If the size of a read set is a problem, then the algorithm of Kung and Robinson offers a finer grain of concurrency, so reducing the chance of conflict. The problem of age can be solved by always reloading an object prior to reading it , but this is pessimistic. If the network supports broadcasting, or better still, multicasting, then each object store can broadcast a list of PID's that have been invalidated by the last transaction. Each Smalltalk virtual machine will examine the list, and delete (turn into a *leaf*) any of the objects that are currently resident. If the object is referenced again it will be reloaded in its most modern version.

A second method of controlling restarts is to use sub-transactions. The room-booking program in Appendix 1. uses this method. When a sub-transaction commits and is accepted, nothing actually changes in the persistent store, because the sub-transaction is dependent on the overall outcome of its enclosing transaction; it is guaranteed to commit or abort if necessary, but it remains a shadow pending the outcome of the 2 phase commit. In this state it behaves like a lock on all the objects it has changed; no more recent transaction can invalidate it. The behaviour is not precisely the same as 2PL, as subsequent transactions are not serialized through waiting, but it is possible to obtain some of the advantages of 2PL without sacrificing any of the advantages of the optimistic technique.

5. Immutable Objects

The term "immutable object" as used by, for example Liskov and Guttag [18], refers to objects whose state cannot change, because there are no operations which "mutate" that state. New objects are created in an initial state, which remains fixed for all time. It is useful to relax this idea of immutability to include objects which were mutable up to a certain time and thereafter have a fixed state. The crucial difference is that the second type of object *does* have mutating operations, which suddenly become invalid. A common example of this can be found in containers which have a "read-only" switch which can be toggled. The configuration RAM in my music synthesiser has such a switch; UNIX files have such a switch. When

the switch is set to read-only, attempting to write causes an error.

It is possible to distinguish three types of immutability:

1. objects which have no mutating operations, and are immutable by declaration e.g. integers.
2. objects which become "read-only", but whose state at the time it is frozen may contain references to objects which are mutable.
3. objects which become "read-only", and whose state at the time it is frozen contains only references to other immutable objects of type 3 or type 1.; e.g. the *total state* of the object is immutable.

Type 1. will be called *intrinsic immutability*, type 2. will be called *read-only*, and type 3. will be called *recursive read-only*. Instrinsic immutability is a property which can be declared at the language level; Emerald [6] provides the ability to declare immutable objects of this type. The other two types of immutability are dynamic; a mutable object becomes immutable, and in this system, remains immutable.

Immutable objects are the keystone to building an efficient, shared, persistent object store. Immutable objects do not have to participate in concurrency control. Once loaded into local object memory they can be read at no further cost. There are enormous benefits in trying to make as many objects as possible immutable. In particular, it is proposed that Smalltalk classes can be made recursive read-only.

One of the problems of sharing classes is that if someone modifies a shared class, everyone else will be affected by the modification. It is not in the common interest to permit important system-wide classes to be mutable. The rule adopted is that every shared class is made recursive read-only. When a Smalltalk program is compiled, the compiler binds textual references to classes into object references by using the global dictionary **Smalltalk**. The behaviour of a class can be modified by binding a new class to the class name in the dictionary, rather than by directly modifying the class object. The procedure for modifying a class is

- copy the current immutable shared class.
- modify the copy.
- make the copy recursive read-only.
- bind the copy to the name of the class in **Smalltalk**.

New instances of the class will conform to the new behaviour. As instances of classes refer to their class via an object reference, old instances will still refer to the original class, which is now inaccessible but will remain in the persistent object store for as long as there is an instance to reference it. This behaviour is exactly what is needed: an object always refers to the immutable class object of which it is an instance, and this relationship exists until the last object of that class is removed. Classes can be modified, but not at the expense of existing instances. The installation of modified shared classes still needs to be controlled, but change control is necessary in any shared environment, and is beyond the scope of this paper.

A Smalltalk programmer usually develops new classes; these are frequently changed during development and debugging, and it would be counter-productive to slow down program development by insisting that all classes are shared, persistent and immutable. It is possible to have mutable private classes; to do this requires a minor modification to the Smalltalk compiler so that it resolves global names by searching a

local name dictionary in addition to **Smalltalk**.

Smalltalk in its current practice does not favour the use of immutable objects; the ability to change anything at any time is one of its attractions for rapid prototyping. The functionality of the persistent object store does not depend on having immutable objects, but its performance does. Aside from the performance issues, immutability is an important concept that is deeply connected with persistence and sharing. An object is a container for a value; values are intrinsically immutable. By distinguishing between immutable and mutable objects we are really distinguishing between containers and the values they contain. Sometimes we wish to share a value, at other times a container. It is important that a language can distinguish between these two very different ideas.

The introduction of immutable objects into the Smalltalk system is not a trivial change and requires changes to be made to many of the system classes, such as the **Browser**.

6. New Classes and Modifications to the Object Protocol

This section introduces a new class and changes to the Smalltalk **Object** protocol. Additions to the object protocol are:

persist - the receiver and every object indirectly dependent on it will be made persistent. Note that unless the object is inserted into the tree of persistent objects it will not be accessible, and hence garbage collected.

reload - the receiver and every object on which it indirectly depends is reloaded into the cache in its most recent version, on demand. The current version in the cache is invalidated.

shallowReadOnly - returns a copy of the receiver that is read-only. The object is copied in the same way as **shallowCopy**.

deepReadOnly - returns a copy of the receiver that is recursive read-only. The object is copied in the same way as **deepCopy**.

The new object class is **Transaction**. A **Transaction** is a primitive object type that acts as an anchor for the read and write sets that must be accumulated as part of the concurrency control algorithm. Sending the message **commit** to a transaction object results in the read and write sets being sent to the appropriate object stores; the result of committing a transaction is either **True** or **False**, depending on whether it completed or aborted. Creating a transaction modifies the global state of the virtual machine on a per-process basis; an internal boolean flag **IsTransaction** is set to true, and the transaction object id. is added to a push-down stack of active transactions. Any reads or writes on the state of a mutable persistent object are logged by adding an entry to the transaction at the top of the stack. This provides nested sub-transactions, and permits a single large transaction to be split into several parts. The ability to handle nested sub-transactions does add to the complexity of the implementation, but as the TM service already has the ability to manage a transaction split into components, the addition of components which are themselves transactions is not a significant change.

The protocol for **Transaction** is:

new - returns a new transaction.

restart - discards any previous state associated with a transaction and begins again. This is necessary to support nested sub-transactions. Restarting the same sub-transaction until it succeeds is not the same as creating one new transaction after another until one succeeds.

commit - commit a transaction; returns **True** or **False**. The meaning of true varies; if the transaction is a sub-transaction then true means that the effects of the transaction will be visible if the parent commits, but until then the changes are pending, and are guaranteed to be able to commit if necessary. If a transaction is at the top-level then **True** means that changes to the persistent object store are visible to everyone.

abort - abort the transaction.

7. An Example

The following example is taken from a real problem at QMC: we have a seminar room which is used for meetings, seminars, tutorials etc. The seminar room is booked by obtaining a particular notebook, looking up the date, and booking the time slot on the appropriate day. This concurrent access problem has been solved in Smalltalk[1] in a very simple (and unsophisticated) way, to illustrate how transactions and persistence can be used. The booking sheet is an array of 12 entries, one for each month. Each month is a dictionary, with a possible association between a day number and a booking page. A dictionary is used because the room is booked on only a few days each month. Each booking page consists of nine slots corresponding to the hours between 9.00 and 17.00 inclusive. The relationship of these objects is illustrated in Figure 3. below.

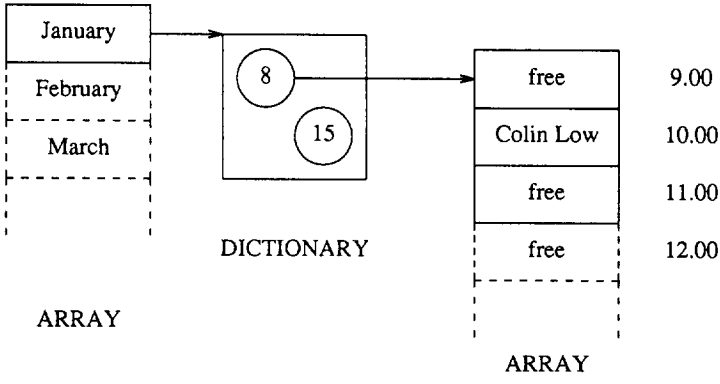

Figure 3: A Persistent Booking Sheet

[1] Actually Little Smalltalk version 2 [9]. The advantage is that the example is compact and completely self contained, including the user interaction. The additions to the protocol for **Object** and the **Transaction** class are not part of standard Little Smalltalk.

The user follows a simple interactive dialogue to pick the month, the day, and then the hour in the day. If the slot is free, the user inserts his or her name, and may then attempt to book more slots. The full text of the program can be found in Appendix 1.

The booking sheet is initialised as an array of 12 dictionaries, made persistent, and then made **shallowReadOnly** because the array will never be changed after initialisation. It is placed in a global dictionary, which can be assumed to be the root for all persistent objects. It is found in this example by using the class **Finder** to map a pathname onto a persistent object.

The **Booker** class uses a main enclosing transaction t1 and a number of subtransactions represented by t2 and t3. It is possible to book several slots as individual sub-transactions, and then to commit or abort all the bookings in one go. t2 is an example of a transaction that restarts automatically and is invisible to the user; if there is no booking page for a particular day in the month, one is inserted, but this only has to be done once. If there is a concurrency conflict and the transaction fails, it is guaranteed to succeed next time, because it will not be necessary to insert a page; it will see the new insertion. It is guaranteed to see the new insertion because the dictionary is reloaded. t3 is restarted until either the user gives up or books a slot. Thus both t2 and t3 can be completed, and so the enclosing transaction t1 can be completed. Subtransactions are an important method for avoiding lengthy restarts.

8. Ownership

The example above works because the **Booker** class is trusted to be civilised. It would be simple to write another version of **Booker** that rips out pages, discards previous bookings, and so provides preferential access to the owner of the modified class. There are many places in a persistent object store where different classes of ownership and protection may be needed to prevent the malicious manipulation of object state; such facilities are normal in filing systems.

Providing a protection scheme poses many practical difficulties, and is a significant problem in its own right. This design for persistent object storage has no protection, a serious weakness that requires further work.

9. Status

A prototype of the persistent object store has been integrated and tested with Little Smalltalk version 2, but it does not contain the concurrency control algorithm described. The object store is currently being completely re-implemented to run under UNIX 4.2 BSD and will have the functionality described in this paper. Modifications are being made to the Smalltalk-80 virtual machine implementation available at QMC [21].

10. Related Work

The integration of languages and persistence overlaps conventional database technology, and the two directions of research are converging rapidly. The following selection of related work does not attempt to be comprehensive, and concentrates on a few closely related projects.

The work described in this paper was influenced by the PISA (Persistent Information Space Architecture) project [4] at the Universities of Glasgow and St. Andrews. As has already been mentioned, the work is based around the language PS-Algol, which uses a shared, persistent heap. The persistent object space is split into databases which provide the unit of granularity for lock based concurrency control.

The Telesophy system described by Caplinger [10] provides an object-oriented framework for the management of *Information Units* (IU). An IU can be any kind of information; IU's can be stored, viewed and modified within a distributed system environment. Concurrency control is based on timestamping.

The GemStone database system [19] combines an object-oriented programming language, OPAL, and persistent objects. The system as described appears to be centralised, with remote user agents to provide a user interface. Concurrency control is achieved with an optimistic technique, and a hybrid approach incorporating locking has been proposed [23].

The VBASE system [1] is based on a pair of languages, TDL and COP, which together constitute a strongly typed object-oriented programming language something like CLU [18]. Database features such as data clustering, inverse relationships and triggers are provided, as are synchronisation and recovery, although the latter are not described.

Jasmine [26] is an object-oriented system for manipulating persistent objects that describe the structure and versions of software. This system is interesting because the objects are immutable and can be replicated at low cost, a factor of great importance in a distributed application.

Merrow and Laursen [20] describe a system called Coral3 which extends Smalltalk to include shared persistent objects. An object class **SharedObjectHolder** is used to encapsulate a representation of the state of an arbitrary object. Sharing is coordinated by locking but transactions are not supported. Behaviour is not shared, that is, classes do not seem to be persistent, leading to possible inconsistencies between different uses of the same object.

The Amoeba File System [22] supports a tree-structured hierarchy of files, and uses file versions and both optimistic concurrency control and locking to provide atomic transactions. In so far as a persistent object store is normally built on top of a filing system, the Amoeba Filing System solves almost all the problems at an underlying level; a persistent object store needs very little extra functionality. It is clear that the difference between a filing system and an object store is largely one of nomenclature, and it will be interesting to see whether a general filing system can match the performance of a persistent store optimised to match the characteristics of a particular object-oriented programming environment.

11. Conclusion

A number of issues have arisen in this work that are of general interest. The first is the identification of the *total state* of an object, which in Smalltalk includes all of the classes from which the object inherits behaviour. The need to store behaviour as well as object state is paramount; there is no actual distinction, and it is a credit to Smalltalk that it supports such a fundamental idea so well.

The second issue is whether the behaviour of an object should be mutable or immutable; Smalltalk permits behaviour to be modified dynamically, but in a shared environment this is a recipe for chaos. The system described adopts the convention that shared behaviour is immutable. The binding between a name, for example, the name of the class **Set**, and the behaviour associated with the class, is mutable; the behaviour itself is immutable and will persist for as long as it is referenced.

The third issue is how the use of immutable objects can be extended as far as possible; the resources wasted in maintaining many immutable objects can be solved by the decreasing cost of hardware, but the difficulty in providing efficient implementations of atomic transactions and the related problem of replication in a distributed system are unlikely to be solved so easily.

Lastly, the uniformity of Smalltalk's object model and the way in which a powerful programming notation can be developed out of simple elements make it useful for low cost experimentation in an area which normally requires a high investment in languages, systems and expertise. It should be an invaluable tool for developing prototypes of highly interactive applications that use shared persistent information.

12. Acknowledgements

Thanks to Eliot Miranda for helping with Smalltalk-80 esoterica and to Takis Anastassopoulos for conversations about concurrency. Thanks also to the referees for their comment and advice.

13. References

[1] Andrews, T., & Harris, C., *Combining Language and Database Advances in an Object-Oriented Development Environment,* ACM OOPSLA Proceedings 1987.

[2] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J., & Morrison, R., *An Approach to Persistent Programming,* Computer Journal **26**(4) 360-365 (1983).

[3] Atkinson M.P., Chisholm K., Cockshott, P., & Marshall R., *Algorithms for a Persistent Heap,* Software Practise and Experience, **13**(3), 259-271 (1983).

[4] Atkinson M.P., Morrison, R., & Pratten, G.D., *A Persistent Information Space Architecture,* Persistent Programming Research Report 21, Persistent Programming Research Group, Dept. of Computing Science, University of Glasgow.

[5] Bernstein, P.A. & Goodman, N., *Concurrency Control in Distributed Database Systems,* Computing Surveys, **13**(2), 185-219 (1981).

[6] Black, A., Hutchinson, N., Jul, E., & Levy, H., *Object Structure in the Emerald System* ACM OOPSLA Proceedings (1986).

[7] Bloom, T., & Zdonik, S.B., *Issues in the Design of Object-Oriented Database Programming Languages,* ACM OOPSLA Proceedings (1987).

[8] Brown, A.J., & Cockshott, W.P., *The CPOMS Persistent Object Management System,* Persistent Programming Research Report 13, Persistent Programming Research Group, Dept. of Computing Science, University of Glasgow.

[9] Budd, T., *A Little Smalltalk,* Addison-Wesley, 1987.

[10] Caplinger, M., *An Information System Based on Distributed Objects,* ACM OOPSLA 1987 Proceedings.

[11] Ceri, S., & Pelagatti, G., *Distributed Databases* McGraw-Hill, 1985.

[12] Coulouris, G., & Dollimore, J., *Distributed Systems: Concepts and Principles,* (to be published) Addison-Wesley 1988.

[13] Gifford, D., *Violet: an Experimental Decentralised System,* Operating Systems Review, **13**(5) 1979.

[14] Goldberg, A., & Robson, D., *Smalltalk-80, The Language and its Implementation,* Addison-Wesley, 1983

[15] Kohler W.H., *A Survey of Techniques for Synchronisation and Recovery in Decentralised Computer Systems,* Computing Surveys, **13**(2), 149-183(1981).

[16] Krasner, G., *Smalltalk-80, Bits of History, Words of Advice,* Addison-Wesley 1983.

[17] Kung, H.T. & Robinson, J.T., *On Optimistic Methods for Concurrency Control,* ACM-TODS, **6**(2), 1981.

[18] Liskov B., & Guttag, J., *Abstraction and Specification in Program Development,* MIT Press, 1986.

[19] Maier, D., Stein, J., Otis, A., & Purdy, A., *Development of an Object-Oriented DBMS,* ACM OOPSLA Proceedings 1986.

[20] Merrow T., & Laursen J., *A Pragmatic System for Shared Persistent Objects,* ACM OOPSLA Proceedings 1987.

[21] Miranda, E., *Brouhaha: A Portable Smalltalk Implementation,* ACM OOPSLA 1987 Proceedings.

[22] Mullender, S.J., & Tanenbaum, A.S., *A Distributed File Server based on Optimistic Concurrency Control,* Proc. of the 10th. Symp. on Operating Systems, ACM. N.Y. P51-62.

[23] Penny, D.J., & Stein, J., *Class Modification in the GemStone Object-Oriented DBMS,* ACM OOPSLA 1987 Proceedings.

[24] SUN Microsystems Inc. *The Network Services Guide,*

[25] Thomas, R.W., *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases,* ACM-TODS, **4**(2), 1979.

[26] Wiebe D., *A Distributed Repository for Immutable Persistent Objects,* ACM OOPSLA Proceedings 1986.

Appendix 1

```
*  Room Booking Class
Declare Booker Object
Class Booker
        bookRoom  |aBookingSheet month day more t1|
           aBookingSheet <- (Finder new) lookup: 'a pathname'.
           more <- true.
           t1 <- Transaction new.
           [more] whileTrue:
              [month<- self selectMonth: aBookingSheet.
               day <-  self selectDay: month.
               self selectSlot: day.
               (slot isNil)
                  ifFalse:
                     [day at: slot put: self createEntry].
               more <- self continue
              ]
           'you may now commit any bookings you have made' print.
           (self continue) ifTrue: [t1 commit] ifFalse: [t1 abort]
|
        selectMonth: aBookingSheet
           ^aBookingSheet at: (self prompt: 'select month [1..12]: ') asInteger
|
        selectDay: aMonth    |day t2|
           day <- ((self prompt: 'select day [1..31]: ' ) asInteger).
           t2 <- Transaction new.
           [true] whileTrue:
              [ (aMonth includesKey: day)
                   ifFalse: [aMonth at: day put: (Array new: 9)].
                (t2 commit)
                   ifTrue: [^(aMonth at: day)]
                   ifFalse:[t2 restart. aMonth reload]
              ]
|
        selectSlot: aDay     |hour more t3|
           "iterate until a free slot is chosen or user gives up"
           t3 <- Transaction new.
           more <- true.
           [more] whileTrue:
           [  t3 restart.
              self showBookings: aDay.
              hour <- (self prompt: 'select slot beginning hour [9..17]: ') asInteger.
              hour <- (hour - 8).        "convert hour to array index"
              (aDay at: hour) isNil
                 ifTrue:  [ aDay at: hour put: self createEntry.
                            (t3 commit)
                               ifTrue:
                                  ['done' print. ^nil]
                               ifFalse:
                                  ['just booked - try again' print]
                          ]
                 ifFalse: ['this slot is already booked' print].
              aDay reload.
              more <- self continue.
           ].
           t3 abort
|
        showBookings: aDay
           (1 to: 9) do:
              [:slot| ((slot + 8) printString , ' ') printNoReturn.
               ((aDay at: slot) isNil)
                  ifTrue:  ['free' print]
                  ifFalse: [(aDay at:slot) print]
              ]
|
        createEntry
           ^self prompt: 'enter your name: '
|
        continue
           ('y' = (self prompt: 'continue y/n: '))
              ifTrue:  [ ^true]
              ifFalse: [ ^false]
|
        prompt: promptString   |aString|
```

```
            [ promptString printNoReturn.
              aString <- smalltalk getString. aString size = 0] whileTrue: [ nil ].
              ^aString
]
Declare BookingSheet Array
Class BookingSheet
        initialise
           (1 to: 12) do: [:month| self at: month put: Dictionary new].
           self persist; shallowReadOnly
]
Declare Finder Object
Class Finder
        "turns a pathname into a persistent object - here it is a stub"
        lookup: aPath
           ^globalNames at:#bookingSheet
]
Declare SetUp Object
Class SetUp
        new
           globalNames at:#bookingSheet put: (BookingSheet new:12) initialise
]
```