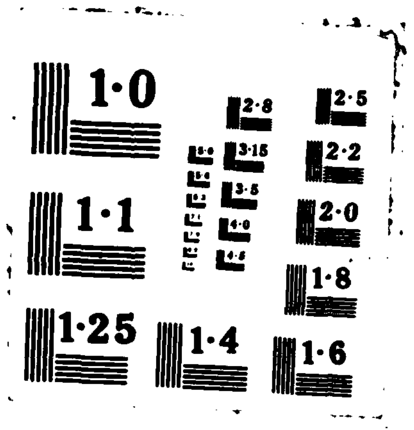
AD-A188 924 EVOLVING PERSISTENT OBJECTS IN A DISTRIBUTED ENVIRONMENT(U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST J NESTOR DEC 87 1/1

UNCLASSIFIED CMU/SEI-87-TR-46 ESD-TR-87-209 F/G 12/5 NL


1·0
2·8
2·5
3·15
2·2
3·6
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

Technical Report
CMU/SEI-87-TR-46
ESD-TR-87-209

2

Carnegie-Mellon University
Software Engineering Institute

DTIC FILE COPY

AD-A188 924

Evolving Persistent Objects in a Distributed Environment

John Nestor

December 1987

88 2 01 115

Technical Report
CMU/SEI-87-TR-46
ESD-TR-87-209
December 1987

Evolving Persistent Objects in a Distributed Environment

John Nestor

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]
Karl Shingler
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

Evolving Persistent Objects in a Distributed Environment

John R. Nestor

Software Engineering Institute, Carnegie Mellon University

November 29, 1987

Abstract

In distributed systems, it is useful to classify persistent objects as either immutable or mutable. The contents of an immutable object cannot be changed while the contents of a mutable object can. In a distributed system, multiple copies of an immutable object can exist at different places and can be used freely without the need for any special synchronization. Mutable objects, however, require synchronization. When an object is about to be changed, all current users need to be notified. When two users both try to change the same object, only one should be permitted to succeed. This kind of synchronization requires that for each mutable object there be a single point in the network that controls use of that object. If a network becomes temporarily partitioned into two isolated subnetworks, only one will have control over each mutable object.

This paper considers a class of objects called incrementally mutable objects that are intermediate between mutable and immutable objects. Intuitively the only permitted modifications to an incrementally mutable object are those that add new information to the object while preserving existing information. Changes to incrementally mutable objects do not require central synchronization. When a network becomes partitioned, the same incrementally mutable object can be safely modified in each subnetwork. A mutable object can be modeled by a set of immutable objects that represent each value of the object over time and an incrementally mutable object that relates each immutable object to its successor. Multiple successors are permitted to represent parallel changes. Keywords: computer programs; computer applications

This work was sponsored by the Department of Defense. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the Department of Defense, or the U.S. Government.

1 Introduction

The demand for software is steadily increasing both in the number of systems being built and in the complexity of these systems. Unfortunately, demand for software has been consistently growing faster than our ability to produce it. Software is often delivered late, costs much more than originally predicted, and fails to satisfactorily do the job for which it was built. The combination of these problems has been characterized as the software crisis and has led to increasing emphasis on the field of software engineering whose goals are directed at solving these problems.

Programming environments have become a focal point for much of the work directed toward improving the practice of software engineering. Environments are increasingly being based on multiple distributed machines connected with both local and wide area networks. The management of persistent data is a central issue for environments. Environments are being called on to be the repository of all information, both technical and managerial, created throughout the lifecycle of a software system from requirements though deployment and continued enhancement. Most current programming environments support persistent data by using a file system with one or more ad hoc databases. Many future environments will be based on object-oriented databases that combine methods used in traditional databases with the programming language concepts of objects and abstraction [1]. These new environments will differ in several important ways from traditional database systems [2,3].

This paper focuses on one important aspect of future programming environments: how to manage evolution of data in a distributed system. First, evolution, existing methods for providing it, and the weaknesses of these methods are considered. Second, some motivations for a new approach are discussed. Third, a new approach, incremental mutability, that eliminates these weaknesses is introduced. Finally, two examples of incremental mutability are presented.

2 Evolution

The information in a software environment evolves over time. Not only are new objects added, but existing objects must also evolve.

The simplest mechanism for evolution is to permit all objects to be changed. However, changing an object presents special problems when multiple users are involved, particularly when the environment is on a distributed system. When two users are changing the same object then the changes of one may overwrite the changes of the other or worse the resulting object may contain some combination of the partial changes of each user. Many environments provide a locking mechanism so that only one user can change an object at any time. This approach has two problems. First, while one user is changing an object, all others who need to change that object are locked out and must wait until the object is unlocked before they can proceed with their work. Since objects can be locked for long times, a loss of productivity may occur. Second, the locking mechanism requires that control of an object reside at a single point within a distributed network. This can be seen by considering two users at different points in the network both trying to change the same object. Suppose that the network is partitioned by breaking all paths between the two users. Now, at most one user will be able to change the object, because control of the object resides in the part of the network where changes to the object can be made. The place where control of an object resides will be called the *control point* for that object. A control point for an object resides at some single point within a distributed system and limits access to operations of the object. Although control points on a single centralized machine have proved useful and effective, they create problems in distributed systems that are discussed in the next section.

If historical information is to be preserved, evolution cannot be done by simply changing the existing object. Instead, a version scheme is needed that records a sequence of objects, each of which represents the state of some changeable object at a different point in time. Instead of changing an object, a new version of the object is created in which all changes have been made. Once created, a version cannot be changed. This ensures that historical information is preserved. Versions need not be totally ordered. When alternatives occur, such as when a bug is fixed in an old release while work continues on the next release, the sequence can fork. When alternatives come together, separate sequences can join. Abstractly, a directed acyclic version graph is formed. Not all points in the version graph are equally important; in practice, users impose additional structure at one or more levels of granularity and do not preserve versions below some minimum level of granularity. The finest granularity corresponds to every edit. A coarse granularity corresponds to major release points. Intermediate granularities are frequently defined to aid the management of a development project. Two common methods for supporting versions are discussed below: naming schemes and version control programs.

Source versions are often handled by conventions for naming directories and files. One primitive approach divides the world into three groups of directories: `old`, `current`, and `new`. Most users will use objects in `current`. Versions under development and experimental versions reside in `new`. When a `new` object is stable, the `current` version of that object is moved to `old` and the `new` version is moved to `current`. This approach has two disadvantages. First, only three versions of an object are kept. In practice, many more than three are needed. Second, the `current` state of the system is constantly changing. The behavior of any uses of `current` objects can change in unexpected ways without any notification. The user is never sure that what worked yesterday will work the same today.

A more general approach is to use names `V1`, `V2`, `V3`, and so forth. Although this approach can be extended to represent version graphs with forks and joins, naming can get complex. Name qualification can occur either at the directory or the file level. For example, if "/" separates directory names and "." can appear as a character within file names, two versions of the object, `foo.x`, could be represented either by directory names such as

```
/foo_project/source1/foo.x
/foo_project/source2/foo.x
```

or by file names such as

```
/foo_project/source/foo.x.1
/foo_project/source/foo.x.2
```

When directory names are used and one object in that directory is changed, then all other objects in the old directory must be copied unchanged into the new directory resulting in multiple identical files that represent a single logical object. [1] When file names are used, the consistency relationships between versions of different objects are no longer explicit and must be separately maintained. Another disadvantage of naming schemes is that command scripts need to be aware of the version naming conventions. For example, a command script that references a `V1` object will have to be edited before it can be used for a `V2` object. [2]

Several programs have been developed for version control including the Unix SCCS tool, a similar but improved Unix tool RCS [4], and most recently the Apollo DSEE system [5]. These tools support version histories of individual objects including those whose version graphs have both forks and joins. All versions of an object are stored in a single physical file using delta encoding to save space. In SCCS and RCS, before any particular version of an object can be used, a copy of it must be explicitly extracted from the physical file. After the copy is changed it must be explicitly copied back as a new version into the physical file. This approach not only requires the user to do these extra explicit operations, but also places the burden on the user of maintaining the logical relationship between extracted copies and the master version. In DSEE, the user specifies a configuration that lists specific versions of each object that the user wants to see. At this point transparent access to those specific versions is provided. Since logically no copy occurs, consistency is automatically maintained.

When a single version has multiple successors, all these tools designate some single version as the *primary successor*. Starting with the first version of the object and following the path via primary successors will end at a version that is designated as the *current version*. Users can request either a specific named version or alternatively can request the current version. Requesting the current version, however, has exactly the same problem of unexpected changes as the old-current-new naming scheme. Since there is only one primary successor, only one user can create it. [3] Control over which user can create the primary successor therefore must rest with a control point with all the resulting disadvantages discussed below.

[1] Many systems provide *links* that can be used to avoid this copying, but only at some cost in structural complexity.

[2] The edit can in some systems be avoided by passing the version as a string parameter that is then inserted into the right place in the file name.

[3] In SCCS, it is even worse since only one user can be changing *any* version of a object at the same time. This completely inhibits parallel development.

3 Motivation

In the next section, a new approach to evolution called incremental mutability is proposed. Incremental mutability is particularly well suited to distributed environments. Three aspects of distribution motivate this approach: the problems of centralized control points, the advantages of immutable objects, and the ways that groups of people manually synchronize their work.

As discussed above, a control point for an object is some single location in the network that centralizes control of the operations that can be done simultaneously on that object. Two examples of control points were discussed in the previous section: the lock that prevents multiple users from simultaneously changing the same object and the control that determines which user can create the primary successor of some single version. The use of control points in distributed systems has several problems:

- **Increased network traffic.** When the user of an object and the control point for the object are at different points in a network, messages must be sent between the user and the control point for each user operation. Consider as an example a conventional tree structured file system, such as the Sun Network File System [6], supported across a distributed system. Any user file creation or deletion requires interaction with the control point for the directory in which that file resides. Such operations can occur at a very high rate [7].
- **Increased user delays.** The round trip time for messages communicating with the control point can result in annoyingly slow response to user commands. This is particularly a problem when low speed links are involved such as modems over telephone lines or when the network is so large and complex that the path between the user and control point involves many intermediate machines.
- **Lock out.** When none of the network paths between the user of an object and the control point for that object are working, the user is completely locked out until some path again becomes available.

In one way or another all the previously discussed evolution schemes required a control point. A goal of the approach proposed below is to eliminate the need for control points.

A second motivation is the advantages of immutable objects in a distributed system. An immutable object is simply one whose value cannot be changed. Immutable objects are the obvious way to capture history. Most version management schemes treat previous versions as immutable. In a distributed system, identical copies of each immutable object can exist at different places within the system. This approach can ideally be regarded as an implementation strategy where there is a single abstract object with a replicated implementation. Any of the copies of the object can be used by itself without reference to the other copies or a centralized control point. Network traffic can be reduced by placing copies of immutable objects where they are likely to be frequently accessed. By having a copy close at hand, no network delays will occur during use. Finally, if a network should become partitioned by hardware failure, then users in each part can use the same immutable object providing each has a copy. As long as all objects are immutable, no control points are needed. However, if work is to progress, at least some change must take place. One approach would be to structure a system as a mix of both mutable and immutable objects. Although such an approach is a significant improvement over a system in which all objects can be changed, control points are still needed. A second goal of the approach proposed below is to gain the advantages of immutability while still permitting change so that work can progress.

To understand how to minimize the need for control points, it is instructive to consider how multiple users working on the same system interact when using a programming environment that provides no synchronization for object modification. Here, the users are forced to invent manual methods for synchronization. Other than failures that occur when someone forgets the state of a manually set lock, such methods work well. An important distinguishing characteristic of these manual methods is the frequency of the synchronization operations. While automated approaches often operate with a frequency of many synchronization operations per second, manual methods may have a frequency of only a few operations per day. While automated systems often do some kind of synchronization at every change, manual synchronization occurs only at relatively rare events such as when a software system is released. By implementing analogues of these manual methods, control point interactions can be decreased. A third goal of the approach discussed below is to support automated methods that synchronize only at major events much like informal manual methods.

4 Incremental Mutability

Incremental mutability is a new approach to evolution that eliminates control points, offers many of the advantages of immutability, and more closely models informal user interactions. In this approach, a class of objects intermediate between mutable and immutable objects is introduced. For a mutable object any change is permitted, while for an immutable object no changes are allowed. For an incrementally mutable object, IMO, the only permitted modifications are those that add new information to the object while preserving existing information.

Formally, we assume each IMO has a type and that each type defines a fixed set of permitted operations. There are three kinds of operations: *create* operations initially create an IMO, *use* operations return information from an object, *change* operations change the value of the object. For specification purposes, use and change operations take the IMO as their initial parameter. Create and change operations return the IMO as a result. For change operations the initial parameter is the value of the IMO before the change and the result is the value of that same IMO after the change. Each of these kinds of operations can optionally have additional parameters.

Formally, IMO's have two defining properties:

- **Monotonicity.** Invocations of a use operation are classified as either stable or unstable. For specification purposes, a predicate *stable* can be applied to any use operation invocation and return true if the result of that invocation will never change. Monotonicity requires that all uses of an object that are stable remain stable and produce the same result after the object is changed. An object is monotonic if:

> For any value X, use operation U, and change operation C of the object, and lists of values $v1$ and $v2$
>
> if $stable(U(X, v1))$
> then $stable(U(C(X, v2), v1)) \wedge (U(X, v1) = U(C(X, v2), v1))$

- **Commutativity.** Invocations of a change operation are classified as either legal or illegal. For specification purposes, a predicate *legal* can be applied to any change operation invocation. When a change operation invocation is illegal then an error condition occurs and the object is not changed. A sequence of changes is legal if all changes in the sequence are legal. Commutativity requires that for any two arbitrary legal sequences of changes that could be made to an object, then applying all of the first followed by all of the second will be legal and produce the same result as applying all of the second followed by all of the first. An object is commutative if:

> For any value of the object X and change operations $C_1 \ldots C_n$, where each C_i can be any of the change operations of the object and lists of values $v_1 \ldots v_n$
>
> Let Q_1 be $\lambda X.C_1(X, v_1)$
> ...
> Let Q_n be $\lambda X.C_n(X, v_n)$
>
> Let $S1$ be $Q_1 \circ Q_2 \circ \ldots \circ Q_m$
> Let $S2$ be $Q_{m+1} \circ Q_{m+2} \circ \ldots \circ Q_n$
> Let $S12$ be $S1 \circ S2$
> Let $S21$ be $S2 \circ S1$
>
> if $legal(S1(X)) \wedge legal(S2(X))$ then
> $legal(S12(X)) \wedge legal(S21(X)) \wedge$
> $S12(X) = S21(X)$

IMO's can be used to support re-creation in programming environments. Recreation is the ability to go back to an old version of a software product and repeat all the steps that were involved in manufacturing it [8]. Manufacturing takes primitives such as source modules and produces products such as executable programs by performing a set of manufacturing steps. All inputs to each manufacturing step, including the program used to perform the step, either must be a primitive or the result of some previous step. The partially ordered set of manufacturing steps is captured by a derivation graph. Re-creation of a product is possible if all primitive objects and the object that holds the derivation graph are either immutable or incrementally mutable and if all operations on those objects used during manufacturing are stable.

IMO's have an attractive implementation in a distributed network. First, like immutable objects, multiple copies of an IMO can be placed at different points within the network. When an IMO is changed, the local copy of that object is changed *immediately and messages requesting the change are sent to all remote copies.* The local user need not wait for those messages to arrive before proceeding to further use and modify the IMO. This implementation allows progress to be made even in the presence of long network delays and failures of network links. If a network becomes partitioned then any user who has access to any copy of a given IMO can use and change it. Any messages being sent to a place to which all network links are currently down are queued until some connection is again available. If an IMO is not changed again until all messages are received and processed, all copies of an IMO will converge to the same value.

The operations that change an IMO can be considered to be events. On a network basis, events are only partially ordered. At any given place within the network, events will be seen as totally ordered in a way that is compatible with the partial order. The total order seen at different places will, in general, be different. Another way of viewing the partial order of events is to consider time to be relativistic [9]. In relativistic time, there is no system-wide absolute clock. Each machine within the network is assumed to have its own clock that progresses at its own rate. Control points can be thought of as a way of establishing a system wide total ordering of events. IMO's permit events to occur in different orders on different nodes, thus *eliminating the need for control points.*

5 Example 1: Set

In this section a simple example of an IMO type is presented, the set type. Set objects can be used in a programming environment for several purposes:

- **Bug Report Set.** Here each element is a string that describes some bug found in a particular source module.
- **Distribution List.** Here the set elements represent people who have been sent a copy of some document.
- **Property Set.** Here the elements are references to other objects. A property set object contains references to immutable objects that all satisfy some specific property. [4]

IMO sets are formally described below by specifying a type for internal state and a set of operations. For each operation, algebraic rules give the semantics of the operation. Rules are also included to specify when use operations are *stable* and when change operations are *legal.*

[4]The multidirectory sets discussed in [10] could be implemented using property sets.

- **Type.**
  IMO sets can have elements of any type T.

  set of T

- **Create Operation: create_set_of_T.**
  The create operation creates an empty set.
  Form:

  create_set_of_T() $\Rightarrow$ set of T

  Rules:

  create_set_of_T() = $\emptyset$

- **Use Operation:*is_member.***
  This operation tests for set membership. Since members cannot be removed, is_member is stable when its result is true. Since members can later be added, is_member is unstable when its result is false.
  Form:

  is_member(s:set of T,e:T) $\Rightarrow$ boolean

  Rules:

  is_member(s,e) = e $\in$ s
  stable(is_member(s,e)) = is_member(s,e)

- **Change Operations: insert.**
  This operation adds a new member to the set if it was not already present.
  Form:

  insert(s:set of T,e:T) $\Rightarrow$ set of T

  Rules:

  insert(s,e) = s $\cup$ { e }
  legal(insert(s,e)) = true

6 Example 2: Version Graphs

Mutable objects can be modeled by a set of immutable objects that represent each value of the object over time and an IMO relation object that relates each immutable object to its successor. This IMO relation is, in effect, an encoding of a version graph. Multiple successors can be used to represent parallel changes. Multiple predecessors can be used to represent merged development paths.

The type and operations for version IMO's are given below.

- **Type.** Version graphs are encoded by specifying the initial version and the links between versions. Specific versions are represented by separate immutable objects. Each of these objects will have a unique identifier, UID, that distinguishes it from all other objects. The UID's are then used in version graph objects to serve as object references.

vgraph=record{initial:UID,next:set of record{old:UID,new:UID}}

- **Create Operation: create_vgraph.**
  This operation creates a vgraph object initialized to have only a single initial version.
  Form:

  create_vgraph(init:UID) $\Rightarrow$ vgraph

  Rules:

  create_vgraph(init).initial = init
  create_vgraph(init).next = $\emptyset$

- **Use Operation: initial.**
  This operation returns the initial version.
  Form:

  initial(d:vgraph) $\Rightarrow$ UID

  Rules:

  initial(d) = d.initial
  stable(initial(d)) = true

- **Use Operation: in_vgraph.**
  This operation returns true if some specified version is in a vgraph. Like the set is_member operation, it is stable when its result is true.
  Form:

  in_vgraph(d:vgraph,x:UID) $\Rightarrow$ boolean

  Rules:

  in_vgraph(d,x)) = (x = d.initial) $\vee$ ($\exists$ y, <y,x> $\in$ d.next)
  stable(in_vgraph(d,x)) = in_vgraph(d,x)

- **Use Operation: predecessors.**
  This operation returns all the predecessors of a given version. The add operation defined below guarantees that all predecessors of a version are specified at the time the version is entered into the vgraph and that no predecessors can later be added.
  Form:

  predecessors(d:vgraph,x:UID) $\Rightarrow$ set of UID

  Rules:

  predecessors(d,x) = {y | <y,x> $\in$ d.next}
  stable(predecessors(d,x)) = in_vgraph(d,x)

- **Use Operation: successors.**
  This operation returns all the successors of a given version. Since successors can always be later added, this operation is unstable.
  Form:

  successors(d:vgraph,x:UID) $\Rightarrow$ set of UID

  Rules:

  successors(d,x) = {y | <x,y> $\in$ d.next}
  stable(successors(d,x)) = false

- **Change Operation: add.**
  This operation adds a new version to a vgraph. All predecessors are specified and must already be in the vgraph. A value for the new version is passed to the add operation which returns the UID of a new object with that value. The add operation must have two results, the new vgraph and the UID of the new object. This is achieved by returning a record with two components, one for each result.
  Form:

  add(d:vgraph,old:set of UID,v:value) $\Rightarrow$ record{d:vgraph,new:UID}

  Rules:

  add(d,old,v) = <<d.initial,d.next $\cup$ {<o,new> | o $\in$ old }>,new>
  where new is the UID of a new object created by add
  whose value is v
  legal(add(d,old,v)) = ($\forall$ o $\in$ old, in_vgraph(d,o)) $\wedge$ old $\neq \emptyset$

7 Conclusions

Current methods for providing evolution of data in a programming environment were shown to have disadvantages when the environment runs on a distributed system. A new approach, incremental mutability, provides evolution but has none of these disadvantages.

8 Acknowlegements

I would like to thank Ellen Borison, Susan Dunkel, Peter Feiler, Robert Firth, Mary Shaw, Don Stone, and Bill Wood for comments on earlier drafts of this paper.

9 References

1. Klaus Dittrich and Umeshwar Dayal, International Workshop on Object-Oriented Database Systems, IEEE Computer Society Press, September 1986.

2. John R. Nestor, "Toward a Persistent Object Base", *International Workshop on Programming Environments*, Lecture Notes in Computer Science, Number 244, Springer-Verlag, 1986.

3. Philip A. Bernstein, "Database System Support for Software Engineering, An Extended Abstract", *9th International Conference on Software Engineering*, IEEE Computer Society Press, 1987.

4. Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the 6th International Conference on Software Engineering*, IEEE Computer Society Press, 1982.

5. David B. Leblang, Robert P. Chase, Jr., and Gordon D. McLean, Jr., "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts", *Proceedings of the 1st International Conference on Computer Workstations*, IEEE, November 1985.

6. R. Sandberg, "The Design and Implementation of the Sun Network File System", *Proceedings Usenix*, June 1985.

7. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2BSD File System", Technical Report UCB/CSD 85/230, University of California, Berkeley, April 1985.

8. Ellen Borison, "A Model of Software Manufacture", *International Workshop on Programming Environments*, Lecture Notes in Computer Science, Number 244, Springer-Verlag, 1986.

9. Leslie Lamport, "Times, Clocks, and the Ordering of Events in a Distributed Environment", *Communications of the ACM*, Volume 10, Number 2, March 1984.

10. John R. Nestor, "Views for Evolution in Programming Environments", To appear in book of papers by participants at the 1986 International Workshop on Object-Oriented Database Systems.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-87-TR-46	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-87-209
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE / 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE / 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A

11. TITLE (Include Security Classification)
EVOLVING PERSISTENT OBJECTS IN A DISTRIBUTED ENVIRONMENT

12. PERSONAL AUTHOR(S)
JOHN NESTOR

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ______ TO ______	DECEMBER 87	14

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	INCREMENTALLY MUTABLE OBJECT, DISTRIBUTED SYSTEMS, SYNCHRONIZATION, PERSISTENT OBJECTS, NETWORK PARTITIONING

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

IN DISTRIBUTED SYSTEMS, IT IS USEFUL TO CLASSIFY PERSISTENT OBJECTS AS EITHER IMMUTABLE OR MUTABLE. THE CONTENTS OF AN IMMUTABLE OBJECT CANNOT BE CHANGED WHILE THE CONTENTS OF A MUTABLE OBJECT CAN EXIST AT DIFFERNT PLACES AND CAN BE USED FREELY WITHOUT THE NEED FOR ANY SPECIAL SYNCHRONIZATION. MUTABLE OBJECTS, HOWEVER, REQUIRE SYNCHRONIZATION. WHEN AN OBJECT IS ABUOT TO BE CHANGED, ALL CURRENT USERS NEED TO BE NOTIFIED. WHEN TWO USERS BOTH TRY TO CHANGE THE SAME OBJECT, ONLY ONE SHOULD BE PERMITTED TO SUCCEED. THIS KIND OF SYNCHRONIZATION REQUIRES THAT FOR EACH MUTABLE OBJECT THRE BE A SINGLE POINT IN THE NETWORK THAT CONTROLS USE OF THAT OBJECT. IF A NETWORK BECOMES TEMPORARILY PARTICIONED INTO TWO ISOLATED SUBNETWORKS, ONLY ONE WILL HAVE CONTROL OVER EACH MUTABLE OBJECT.

THIS PAPER CONSIDERS A CLASS OF OBJECTS CALLED INCREMENTALLY MUTABLE OBJECTS THAT ARE INTERMEDIATE BETWEEN MUTABLE AN DIMMUTABLE OBJECTS. INTUITIVELY, THE ONLY PERMITTED MODIFICATIONS TO AN INCREMENTALLY MUTABLE OBJECT ARE THOSE THAT ADD NEW

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 / 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

BLOCK 19 CONTINUED

INFORMATION TO THE OBJECT WHILE PRESERVING EXISTING INFORMATION. CHANGES TO INCREMENTALLY MUTABLE OBJECTS DO NOT REQUIRE CENTRAL SYNCHRONIZATION. WHEN A NETWORK BECOMES PARTITIONED, THE SAME INCREMENTALLY MUTABLE OBJECT CAN BE SAFELY MODIFIED IN EACH SUBNETWORK. A MUTABLE OBJECT CAN BE MODELED BY A SET OF IMMUTABLE OBJECTS THAT REPRESENT EACH VALUE OF THE OBJECT OVER TIME AND AN INCREMENTALLY MUTABLE OBJECT THAT RELATES EACH IMMUTABLE OBJECT TO ITS SUCCESSOR. MULTIPLE SUCCESSORS ARE PERMITTED TO REPRESENT PARALLEL CHANGES.