Technical Report

UCAM-CL-TR-296
ISSN 1476-2986

Number 296

A persistent programming language for multimedia databases in the OPERA project

Z. Wu, K. Moody, J. Bacon

April 1993

ISSN 1476-2986

A Persistent Programming Language for Multimedia Databases in the OPERA project

Z Wu, K Moody and J Bacon

University of Cambridge Computer Laboratory, UK

1. Introduction

The advent of high bandwidth local area ATM networks has transformed the potential of distributed computing systems. At the Computer Laboratory we are moving towards a world in which multimedia displays are managed by editing, browsing and composing tools [Bates 93]. The recently completed Pandora project [Hopper 90] has given us some experience of multimedia applications, and an idea of their scope.

Current storage services are unable to meet the requirements of emerging application areas. There are both architectural and engineering reasons for this. A computing environment which supports multimedia (such as real-time video and audio) requires very large amounts of storage, the ability to express complex relationships between stored objects and synchronised delivery of media streams at guaranteed rates. A multi-service storage architecture (MSSA) has been designed for such applications [Bacon et al., 91].

The architecture is open for flexibility and extensibility. At the low level of a two-level hierarchy are physical servers which manage storage media of any type. At the high level there are services to manage video and audio as well as conventional files, together with a server for structured data, the *structured file custode* (SFC) [Thomson 90].

We have developed a persistent programming language PC++[Wu 93], an extension to C++, to help programmers developing multimedia applications to make use of the MSSA. To manage data in a distributed, real-time context an application language must meet a number of requirements.

- It must be easy to manipulate data structures stored across many special purpose servers in a distributed environment.
- It must provide applications with the ability to express complex relationships between objects.
- It must allow applications to handle very large objects.
- Clients should only need to be aware of naming mechanisms – existence control should be automatic.
- Persistent data must be protected against the many modes of failure that will occur in a distributed computing environment.
- Often, several data objects must be modified in a consistent fashion. A mechanism for atomic update (and also concurrency control) is essential.

In this paper we present the design of PC++ and show how its special features meet these requirements.

2. Overview and related work

PC++ extends C++ with persistent classes. A class in C++ defines a type, and its definition includes both the physical representation of any object of the class and the operations that may be performed on it. A persistent class in PC++ keeps all the features of class, and provides additional features which are important for multimedia databases. Instances of a persistent class are called persistent objects. They are allocated in persistent store and continue to exist after the program that created them has terminated.

Research in persistent programming languages has been carried out for some time, and the tradition can be identified in PS-Algol [Atkinson et al. 83], Galileo [Albano et al. 85], and E [Richardson and Carey 89]. The approach of PC++ is based on atomic data types: the project that has had the most influence is Argus at MIT [Weihl 84]. The Arjuna system at Newcastle [Shrivastava et al. 91] shares some features but has a number of significant differences. Persistent programming languages are closely related to object-oriented database systems. A number of these systems, such as GemStone [Copeland and Maier 84] and Vbase [Andrews and Harris 87], have appeared both in the literature and in the market-place.

PC++ is based in this tradition, but it is tailored specifically to multimedia applications. It is integrated with a storage architecture (MSSA) that was developed to support continuous media as well as conventional files. It uses the SFC for storing both data and metadata, thus integrating the persistent programming language environment with the storage architecture.

PC++ supports the names (SSIDs) of objects managed within the open architecture of MSSA as a special type. Persistent class declarations can include SSIDs, and the application programmer may therefore construct and manipulate object data structures (for example, to represent a multimedia presentation) whose components are stored across many special purpose servers within the MSSA.

The use of the SFC for data storage has proved very convenient. Since SFC objects are tree-structured data migration and shadow versions may be managed at subobject level, which allows applications to work with very large objects. Also, PC++ stores the metadata table that maps persistent object identifiers (OIDs) to storage service identifiers (SSIDs) as a structured object on the SFC. This ensures that the existence control mechanism provided by the SFC applies even when there are multiple shadow versions of a (sub)object associated with current transactions.

Persistent objects can be located anywhere in a distributed system, and they are accessed as if they were local to the application. The use of remote procedure calls (RPCs) is hidden by a remote object invocation (ROI) mechanism provided by the system. The ROI mechanism also masks many of the failures that may occur in a distributed environment.

Persistent objects have the properties of atomic data objects. A lower (physical) level guarantees the atomicity of method invocation. A higher (semantic) level enforces a serialisable execution that takes advantage of operation semantics. We believe that the use of (non-blocking) optimistic concurrency control may prove more suitable for flexible real-time applications than two-phase locking.

3. Data persistence

In PC++ an abstract data type is declared by the *persistent class* construct. Like the *class* construct in C++, persistent class declarations consist of two parts: a specification and a body. The *specification* represents the class "user interface"; it contains all the information necessary for the user of a class. The *body* consists of the bodies of methods declared in the class specification.

A member in the specification may be either a data item, called a *data member*, or a method, called a *member function.* The data members of a class describe the object state, and the member functions describe the operations applicable to the state. The data members and member functions of a class are together called the *properties* of the class. Generally, properties can be classified into *public, private* or *protected* according to their accessibility. Data members can only be defined as either private or protected to enforce information hiding and thus to enhance reliability.

The declaration of data members is the same as in C++. A set of *primitive types* and a set of *type generators* are provided to define data members. Data members can be of any provided *primitive type* or *structured type* generated by the generators provided. The primitive types supported are *long int, double, char, boolean, string, OID* and *SSID.* The structured types that may be generated include *record* and *sequence. OID* and *SSID* will be explained later. The others retain their C++meaning.

3.1 Storage service identifiers

PC++ is designed to support multimedia applications and is integrated with the MSSA. This storage architecture is open and adopts a layered approach. At the lowest level are high speed *byte-segment custodes* which provide performance guarantees through *sessions* and *tickets.* Above this level are custodes for different data storage types including *continuous media file custodes*, traditional *byte-stream file custodes* and *structured file custodes.* Multimedia applications therefore can store their heterogeneous data across these special purpose custodes in a distributed environment. A structured file custode (SFC) is lightweight compared with a full typed-object manager. It supports only bytes and storage service identifiers as primitive storage types but the constructors sequence, record and union are used to create objects of arbitrary depth.

Every object in the MSSA is named and protected by a storage service identifier (*ssid)* which is a capability labelled with a principal [Gong 89]. An *ssid* can be used to identify an object uniquely system-wide, and only the principal may use the *ssid* to access the object. PC++ supports the storage service identifier (SSID) as a special type. Persistent class declarations can include SSIDs, and the application programmer may therefore construct and manipulate objects which contain references to any items managed within the open architecture of MSSA. This allows a multimedia presentation object, for example, to refer to objects of various media, including video and audio. By making multimedia objects persistent, multimedia applications can enjoy all the benefits of object-oriented programming languages, including the ability to express complex relationships between objects.

Persistent objects are stored on the structured file custode (SFC) and every persistent object is named and protected, at the PC++ language level, by an *oid* (object identifier). An *oid* is a capability labelled with a principal and tagged with a type identifier which is used for type checking. *oid*s are described by the type OID.

3.2 Representation of persistent objects

A persistent class is implemented by two parts in PC++: a definition for data members and a definition for member functions. The former is represented by a persistent data type, called the *state type*. The latter is represented by a C++ class, called the *formal class*. The formal class defines the member functions of the persistent class, while the state type defines the data members of the persistent class.

A persistent object is composed of two parts: a *formal object* that is a C++ object and a *value object* that is an SFC object. The formal object represents the operation part of a persistent object, and the value object the data part. Value objects are stored on the SFC and can be shared by different users. Formal objects are transient; they are created by, and are local to, a user program and are destroyed automatically at the end of a program session.

A user program creates a persistent object by calling the *create* operation on a formal object which is declared in the program. The *create* operation creates the data part of the persistent object, the *value object,* and an *oid* is generated. A value object can be accessed by any program which has adequate rights.

To access an existing value object, a user program must declare a formal object which is of the same type, and then call the *invocation* operation to bind the value object to the formal object, thus making the persistent object active. After activating an object, the program can manipulate it by calling any operations defined on it.

3.3 Object naming

In any system that supports persistence, mechanisms are needed to support persistent object naming by users and by the system. Typically, strings are used for user-level naming and a scheme based on globally unique object identifiers is used at system level. Clearly, a mapping between the two naming schemes is required.

In PC++ the system-level name for a persistent object, the *oid*, can be used directly by programmers. Therefore, a user-level text-name is not strictly necessary for a persistent object. Also, if an object is referred to directly by its *oid*, the mapping between the user-level name and the system-level name is avoided, thus speeding up access. The reason that PC++ can exploit its system-level naming scheme at the user-level is that the *oid* of a persistent object is protected (a capability labelled with a principal) and only the principal may use the *oid* to access the object.

A system with only *oid*s as user-level names would be tedious to use, so PC++ supports optional text string, user-level names. However, many objects in a system are referenced directly only by other objects and this can be done through *oid*s.

3.4 Binding and type checking

An essential property of a language with persistence is that objects in the object store can be manipulated using the same expression syntax as volatile objects. In order to execute such an expression there must first exist a binding between symbols in the program and objects in the object store. A subject which has a description of the object to which it wishes to bind may perform a binding. The description is usually provided in the form of a type. Since the subject and object may be prepared separately, type-checking is needed to ensure that the types match.

A value object stored on the SFC is bound to a formal object at run-time as described in Section 3.2. To ensure data abstraction is not violated, it must be guaranteed that a value object can only be bound to a correct formal object. To achieve this, the identifier of the class of a persistent object is included in its *oid*. This can be checked against the class identifier of the formal object to which the value object is going to be bound.

3.5 Data migration

Because value objects reside on the SFC, a mechanism is needed to migrate the data in and out of user memory space during a program run. Data migration does not happen with object binding, but with expression evaluation. Although the granularity of binding is the object, data can be migrated in and out of user memory space at any granularity. By moving in only the components that are needed to evaluate an expression, memory space and time can be saved. This allows programs to use very large persistent objects.

3.6 Referential integrity

In a system which supports persistence, attention should be paid to preserving referential integrity [Morrison 90]. If an object is pointed to (shared by) two or more other objects, this sharing should be preserved when they are stored in the persistent store and when they are activated again.

Objects can be shared by being referenced from other objects. A formal object can share another formal object by pointing to it. In the object store, a value object can share another value object by containing its *oid*.

When an active object first makes a reference to another object through its *oid* the PC++ server checks whether that object has already been activated. If not, the PC++ server activates the object, then passes the new memory address to the user process. If the object is already active the PC++ server passes the existing memory address of the object to the user process.

3.7 Object existence control

Existence control in the MSSA is provided through an ageing mechanism [Wilson 92]. It is the responsibility of the clients (which are likely to be other services) to touch all their objects within a defined time after which the MSSA is at liberty to delete them. The SFC provides this "touching" service for its clients. It is therefore in a client's interest to preserve its SSIDs in SFC structured objects. The metadata tables that map persistent object identifiers to storage service SSIDs are stored in this way on the SFC. Clients of PC++ need only be aware of naming mechanisms, object existence control is automatic.

4. Data atomicity

Persistent classes have the properties of atomic data types, whose objects are responsible for their own serialisability and recoverability.

4.1 Concurrency control

PC++ uses an optimistic method, called the *dual-level validation* (DLV) method [Wu 93], to provide serialisability and recoverability for persistent objects. A lower (physical) level validation guarantees the atomicity of method invocation. A higher (semantic) level enforces a serialisable execution that takes advantage of operation semantics. The concurrency control strategy is optimistic, and has the following advantages:

- It allows applications to handle very large objects by supporting multi-granularity shadowing.
- Transaction recovery is simple to implement.
- It works well in distributed environments because its validation algorithm requires only weak preconditions.

4.2 Representing the semantics of operations

PC++ provides a small language for users to specify the operation semantics of persistent objects. In a specification an operation is represented not only by its name but also by its result and the object component that it acted on. Operation semantics can be represented at a high level, thus maximum concurrency is suppported, subject to exclusion requirements at the low level. The result of an operation is recorded as either *failed* or *succeeded*, since this distinction is usually sufficient to determine operation conflict.

4.3 Constructing atomic objects

By using type inheritance PC++ provides an implicit approach for implementing atomic objects while still permitting objects to support type-specific concurrency. The method is quite straightforward. A special type called *Scheduler* is constructed to provide the DLV method of concurrency control. User-defined types can inherit this underlying concurrency control facility by using type-inheritance. By allowing particular types to provide their own operation semantics, objects can provide type-specific concurrency. Therefore, to construct an atomic data type, a programmer need only define the object state and object operations in a sequential environment, plus the specification of the operation semantics of the type.

4.4 Constructing transactions

Transactions in PC++ are implemented as objects. A class, called *Transaction*, is available and programmers construct a transaction by declaring an instance of *Transaction* and enclosing computations by the primitive operations provided by *Transaction*.

A transaction is assumed to be a passive entity that controls the outcome of the operations it encloses. Each operation accesses atomic objects that provide local atomicity by using the DLV method. There are three primitive operations which may be used to declare and control a transaction: *begin-transaction, commit-transaction* and *abort-transaction.*

To construct a transaction programmers need to declare an instance of the *Transaction* class in a program and then enclose computations by the *begin-transaction* and *commit-transaction* operations. If, during the execution of the computation, it becomes necessary to abort the transaction then the *abort-transaction* operation is invoked.

5 Distributed data

Persistent objects can be located anywhere in the distributed system and can be accessed as if they were local to the application. The use of remote procedure calls (RPC) to perform actual accesses is hidden by a *remote object invocation* (ROI) mechanism provided by the system. The client/server model is used. Every resource or abstraction is represented by an object. A server provides service through a set of objects. Clients make use of a service by invoking operations defined on the objects. Several server or worker processes are created for and assigned to each object to handle its invocation requests. When a client makes an operation invocation, a process in the corresponding server object accepts the request and performs the operation on the client's behalf.

In order to provide distribution transparency, for each object resident on the server side, called the *server object*, a corresponding object is provided on the client side, called the *client object*. Each user program runs a copy of client objects and invokes operations on them to ask for service. A client object has the same interface as the server object, but has a different implementation. A client object is implemented in such a way that each of its operations actually is an RPC call to the server side. The call message contains the target object's name and the operation's parameters, among other things. On the server side, when a call message arrives, the server process extracts the object name and the procedure parameters. It then invokes the corresponding operation on the server object, and sends a reply message after execution. Since a client object has the same interface as its corresponding server object, there is no difference between invoking operations on client objects and on server objects.

The binding between a client object and its corresponding server object is done by the ROI mechanism. Before invoking any operation on a client object, the client program must call the *invoke* operation on it with the server name and the object name. The ROI mechanism intercepts this invocation, makes use of the RPC mechanism to locate the specified server object, issues an *invoke* operation on the object (this registers the client program with the object), and returns the logical address of the object. Then the client object is bound to the server object, thus the ROI and RPC mechanism can transfer any invocation on the client object to an appropriate invocation on the server object and possibly return a result.

A stub generator is implemented which operates on the original C++ class definitions to produce the definitions of client objects etc. needed by the ROI mechanism. Because the stub generator can operate directly on the C++ class definitions, programmers need only produce a single object description as if the system were not distributed, relying on the stub generator to produce the distributed version automatically.

6 Summary

We have described the design of PC++ and have shown how it provides special support for multimedia databases. Implementation details can be found in [Wu 93].

PC++ has already been used to reengineer a simple distributed application, namely to maintain the database for an active badge sytem that is used within the Laboratory. This is a low bandwidth application in which updates to the database are obtained from distributed collection points. The database can be interrogated from any terminal within the Laboratory. We are also working on planned (and unplanned, due to network failures) detached working . The optimistic approach used in PC++ provides an ideal model.

A platform for composing and displaying multimedia presentations is nearing completion in prototype form [Bates 93]. At that stage we shall have a complete quality of service architecture in place and will be in a position to experiment with supporting a wide range of multimedia applications.

Acknowledgements

To other members of the project, past and present: Noha Adly, Mohammad Afshar, Huang Feng, Richard Hayton, Scarlet Schwiderski, Robert Sultana, Sue Thomson, Tim Wilson. To Heather Brown for many helpful discussions.
To SERC for grant GR/H 13666 which supports the storage service aspects of the work. To Olivetti Research Laboratories, Cambridge, for network support and use of a Pandora box. To John Wilkes of HP Research Laboratories Palo Alto, CA, for support and continued interest.

References

[Atkinson et al. 83]
M. P. Atkinson and P. J. Bailey and K. J. Chisholm and P. W. Cockshot and R. Morrison, "An Approach to Persistent Programming", Computer Journal 26(4): 360--365, 1983

[Albano et al. 85]
A. Albano and L. Cardelli and R. Orsini, "Galilieo: A Strongly Typed, Interactive Conceptual Language", ACM Transactions on Database Systems, 10(2): 230--260, 1985

[Andrews and Harris 87]
T. Andrews and C. Harris, "Combining language and database advances in an object-oriented development environment", Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications, October 1987

[Bacon et al. 91]
J. M. Bacon, K. Moody, S.E. Thomson and T. D. Wilson, "A Multi Service Storage Architecture" ACM Operating Systems Review 25(4): 47-65, October 1991

[Bacon 93]
J. Bacon, "Concurrent Systems: An Integrated Approach to Operating Systems, Database and Distributed Systems", Addison--Wesley 1993

[Bates 93]
J. O. Bates, """Support for Real-time Interactive Presentation of Distributed Multimedia" Computer Laboratory PhD thesis in preparation 1993

[Copeland and Maier 84]
G. Copeland and D. Maier, "Making smalltalk a database system", Proceedings ACM-SIGMOD International Conference on Management of Data, 1984

[Gong 89]
L. Gong, "A Secure Identity-Based Capability System", Proceedings of the IEEE Symposium on Security and Privacy, 56--63, May 1989

[Herlihy 90]
M. Herlihy, "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types", ACM Transactions on Database Systems, 15(1): 96--124 March 1990

[Hopper 90]
Hopper A, "Pandora, an experimental system for multimedia applications" *ACM Operating Systems Review* 24(2), 19-34, April 1990

[Moody et al. 93]
Moody K, Bacon J M, Bates J O, Hayton R, Lo S L, Schwiderski S, Sultana R and Wu Z "OPERA: Storage, Programming and Display of Multimedia Objects" submitted to IEEE Lisboa 93 and Computer Laboratory TR 294

[Morrison and Atkinson 90]
R. Morrison and M. P. Atkinson, "Persistent Languages and Architectures", in Security and Persistence,9--28, eds. J. Rosenberg and J. L. Keedy, Springer-Verlag,1990

[Richardson and Carey 89]
J. E. Richardson and M. J. Carey, "Persistence in the E Language", Software-Practice and Experience, 19(12): 1115--1150 December 1989

[Shrivastava et al. 91]
S. K. Shrivastava and G. N. Dixon and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", IEEE Software, January 1991

[Thomson 90]
S. E. Thomson, "A Storage Service for Structured Data" Cambridge University Computer Laboratory, November 1990

[Weihl 84]
W. E. Weihl, "Specification and Implementation of Atomic Data Types", MIT Laboratory for Computer Science, March 1984 Tech. Rep. MIT/LCS/TR-314

[Wilson 92]
T. D. Wilson, "Increasing Performance of Storage Services", Cambridge University Computer Laboratory, 1992

[Wu 93]
Z. Wu, "A New Approach to Implementing Atomic Data Types", Cambridge University Computer Laboratory, thesis in preparation, 1993