IEEE MELECON 2002, May 7-9, 2002, Cairo , EGYPT.

# An Extendible Persistent System For Programmers

**Erdal Kemikli and Nadia Erdogan**

Electrical and Electronics Faculty
Computer Engineering Department
Istanbul Technical University
Ayazaga, Istanbul, 80626, Turkey
Tel. +90 (212) 229 9555 Fax. +90 (212) 229 9549
Email erdalk@otokoc.com.tr

**Abstract**

A different approach to define a system, persistent object abstraction offers programmers new opportunities. Persistent systems are the next logical step in the higher-level abstraction of electronic information systems. This paper summarizes a research prototype; Extendible Persistent System (EPS) and explains the programmer facilities on this system.

**Keywords** Persistent system, operating system, programming

## 1. INTRODUCTION

In a conventional system, programming languages provide very good support for transient data. Data with longer life spans can be supported by a conventional file system or by a Database Management System (DBMS). The concept of persistency [1] on the other hand suggests that data in a system should be able to persist (survive) for as long as that data is required. As a result, persistent systems provide a uniform abstraction for data management, and save programmers from considerable amount of program development task. In many cases, support for persistency is provided at the programming language level such as PS-Algol [2] and X programming language [3]. The persistency support at the programming language level has two drawbacks; operating systems may not provide necessary support for the implementation [4], and efforts are duplicated for every new persistent language implementation. These reasons motivate the implementation of persistency at the system level [5].

The primary goal of this paper is to explain a persistent, extensible and tailorable computing system model, which attacks the programmer productivity issue from the technical side. The resulting Extendible Persistent System (EPS) is suitable to be used as a base for an extendible system with the required server functionality.

## 2. EPS DESIGN PHILOSOPHY

EPS will supplies facilities that will ease developers to extend the functionality of a system with better modeling and simpler programming. Moreover, the resulting extended system simplifies system administration tasks and configuration management.

EPS is built on five principles:

- Despite of its revolutionary nature, EPS is designed to be easy to learn and adapt since it inherits and resembles to common computing environments of today.
- A uniform system, program and data abstraction is developed that is easy to understand and extend.
- Programmer productivity is addressed on the design, and freedom of choice and programmer control is aimed .
- A modular and user level services approach is chosen for the system design to enable programmers to extend the system easily.

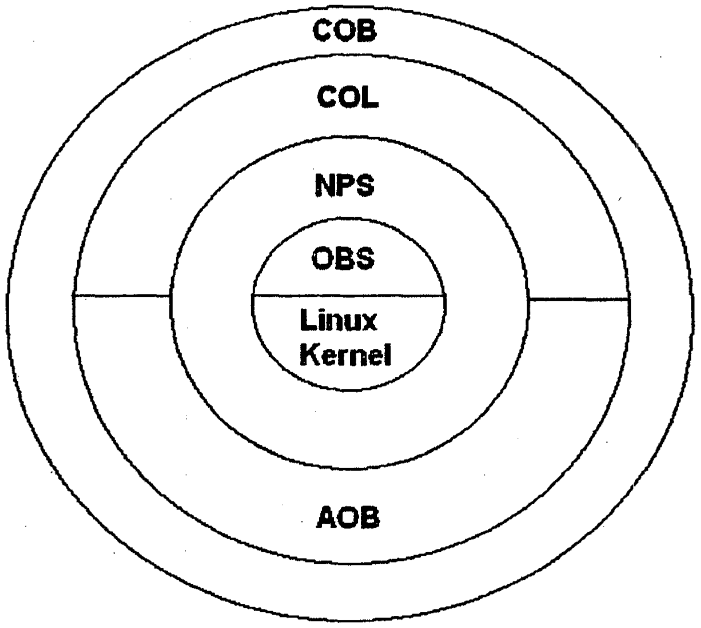

Figure 2.1 EPS System Layers

## 3. EPS COMPONENTS

EPS multi-tier architecture (figure 2.1) is designed to extend naturally based on the needs. The EPS model is composed of several components in different layers. On the highest level, a new programming language, EPS-C which is an extension of the existing ANSI C programming language is defined. The programs written in EPS-C are compiled with the help of an EPS-C preprocessor. During the linking phase, a runtime library (Client Object Library- COL) which is responsible of conducting process level operations and hiding the system complexity from the users by supplying a well defined interface composed of system primitives is linked to the program. Naming and Protection Server (NPS) is a server process running just above the operating system kernel. It resolves persistent object names into persistent ids, protects objects against unauthorized access, and manages the synchronization of access to the objects. Object server (OBS) is another server component of the system responsible of the movement of objects between long and short-term storage devices. The last system component is the inter-object communication (IOC), which supports the whole model through a high level object communication interface based on the well-known IPC paradigm of UNIX. Communication primitives are designed for both synchronous and asynchronous communication. This functionality is implemented as a run-time library and linked to every process with the need of communication.

The basic system is composed of three basic components; Naming and Protection Server (NPS), Object Server (OBS), Client Object Library (COL), and Active Object Library (AOL). An underlying messaging facility, inter-object communication (IOC) interconnects these three modules. While these modules constitute the base system, further user needs can be satisfied by extending the system via active objects (AOB).

### 3.1 Client Object Library

Client Object Library (COL), which is used by every EPS user program, and active object has facilities to hide the underlying complexity of the system. It is linked to every program. Some of the system primitives are implemented fully or partially in the COL. The services of COL also include the local (in-process) implementation of synchronization and address translation facilities, which are transparent to the application programmer. The remote parts of the services are requested from other EPS servers and active objects through the inter-object communication subsystem.

Theoretically, COL is the part of EPS, which needs to be ported to be used for the support of different programming languages. Persistent objects loaded into the primary memory by OBS, and transferred to local memory via inter-object communication subsystem needs one more crucial operation to be useful; conversion of persistent pointer addresses into local memory addresses. While data section of persistent variables does not need any specific post-load operation, pointers in the variables are converted into local pointer addresses. This conversion is conducted by COL and based upon the persistent object type. Once replaced, the pointer values are valid to be used in the process, and need to be converted back into persistent pointers only when they are written back to the disk saving the system from conversion overhead for every access to the persistent object.

## 3.2 Naming and Protection Server

Naming and Protection Server (NPS) is implemented in the form of a server process. NPS is responsible of the security and synchronization of object access. Each request for object access is received and handled by NPS. Persistent object names consist of two parts: creator object name and persistent object name. This two level naming scheme helps to distinguish the persistent objects created by a particular active object, and reduces the probability of collision in persistent object names. Long life span and very wide scope of persistent objects increase the probability of using the same name for different objects.

Every request for loading a static object or an operation from an active object is first converted to Persistent Identification (PID) by matching the requested object name with the existing capabilities, and then evaluated by NPS. If a request is validated then a record is inserted to the inUse table for the persistent object. The result is sent to the client. A client process can access multiple persistent databases simultaneously.

Synchronization has a different context for passive (data only) and active (server) objects. An active object can answer only one request at a time, so the synchronization problem is naturally solved. A passive object can be shared among processes, therefore synchronization issue has to be explicitly managed. Client object tells back when it is done with the requested object, so the access level is reset. This approach has one major drawback, deadlocks. When two objects request objects already accessed by each other, this scheme will cause a deadlock. Instead of deadlock resolution, we chose to implement an extra feature to prevent deadlocks. When client accesses a new passive object, it tells the expected completion time of the request, and it is recorded and used for deadlock prevention.

## 3.3 Object Server

Object Server (OBS) is the persistent object store of EPS. Object stores [6, 7], which are storage services for persistent data, are common component in persistent environments. Physically it is implemented as a separate module and linked to the kernel. The Object Server handles active and passive objects. While active objects consist of data and methods, a passive object contains data but does not have methods to modify it. Since OBS accepts requests only from NPS, it needs not to worry about the synchronization and security issues.

Shadow files technique is used to reduce the risk of creating an inconsistent database. In shadowing, the modified database file is written back to the disk into a different file first. After the file write is complete it is copied into the original file.

Persistent objects are stored on the disk for long-term storage, and loaded into the memory by OBS. A passive object loaded into the process memory is represented by its root object. Root object is the one variable accessible directly from the program instructions. Other related objects are linked to the root object via pointers. Client processes will get the physical address of the root persistent object, and then reach the other objects using pointers.

Each persistent object group is kept in different files and demonstrates a homogeneous structure inside its file. These groups will be called as persistent databases. The type information of the persistent objects will be kept inside the persistent root. While simple object types are defined directly in the persistent root header, complex, user defined types are referenced in the header and will be kept as type definition include files separately.

## 3.4 Active Object

Active Objects (AOB) are not part of the core EPS. However we expect to see a considerable number of AOB to be developed in the future, which will benefit from advantages of EPS. In short, active objects are persistent server processes which themselves can act as client objects to other active objects and EPS.

An important aspect of active objects is their support by the EPS. EPS loads active objects whenever they are needed, transparently. The access control for active objects is handled by NPS, similar to passive objects. Active objects also use the COL library and the inter-object messaging system. An interface, which accepts requests from other objects, is a natural part of an active object.

## 4. PROGRAMMING IN EPS

Programming in EPS is quite similar to programming in a UNIX system with some exceptions. The first difference is the support for persistent data. Another important aspect is the transparent loading of server processes (active objects) as required. The third major difference is the support of an inter-object communication, which is easy to use yet powerful and flexible.

The primary programming language is the C programming language with some extensions. This extended syntax is called as EPS-C to distinguish it from the standard C programming language. Main enhancement to C language is the addition of reserved words to declare the persistent nature of variables and some standard EPS functions as a library. This implementation will allow application developers to use the C programming language alone if they prefer. An EPS-C program is compiled by EPS-C preprocessor and the C compiler. Object code is linked to COL in addition to other required libraries.

Every EPS-C program shall start with the call of EpsInit function. This function will first load the program table. Program table is a predefined type of persistent object used by every EPS program. This persistent object is used to store state data of a program. It can also be used to store small-grained simple type data. EpsInit function also loads the persistent objects if this option was chosen at compile time.

### 4.1 Persistent Variable Management

Since we do not assume all variables to be persistent in EPS, some extra notation was necessary for the declaration of persistent variables. We chose to use a single character "$" at the start of a line of code to declare the persistency property of a data object as seen in the example below.

```
$int      testVar;
```

Persistent variable types can be any simple C variable type or structures defined as types. As a part of EPS design philosophy there is no specific types of files or formats used. Instead, already available type declaration capability is used for the type checking facility of passive persistent objects.

For a complex data structure such as a linked list or linked tree, declaration of the root pointer variable as a persistent object is sufficient. In the loading or saving phases all the objects reachable by the persistent root will be processed.

### 4.2 Compilation and Linking

EPS-C is implemented through a preprocessor used together with a standard C language compiler and a COB library that will be linked to every EPS program. Preprocessor parses the program code and while replacing the persistent declarations with normal C variable declarations, the persistent variables are inserted into the program table. After EPS-C preprocessor phase, the C compiler runs to process the program code. In the linking phase, a standard COB library is linked to every EPS-C program. This library not only includes the application programming interface functions but also contains functions to handle persistent variables.

The EPS-C preprocessor accepts an option to control the loading time of persistent data. By default, all persistent objects are loaded automatically at the start up phase of a program. On the other hand, a compiler option disables the automatic persistent object loading and lets the programmer take control on the object loading time at run time through explicit function calls.

## 5. CONCLUSION

EPS is one of the efforts in the scientific community for the design and development of a computer system with a different philosophy. Main concerns in EPS design are, the software crisis and solution of this problem through an innovative technology. We believe that, persistency and object paradigms, when used together ease the burden on system architects and programmers in development of complex information systems.

Currently explained design is implemented as a prototype system on Linux. The next phase of the research will include experimental studies for the determination of the effectiveness of the system in reducing system complexity and improving programmer productivity.

## References

[1] Atkinson, M. P. et al, (1983), "An Approach to Persistent Programming", The Computer Journal, Vol. 26, No. 4, pp.360-365.

[2] Atkinson, M.P., Chisholm, J., and Cockshott, W.P., (1982), "PS-algol:and algol with a persistent heap", ACM Sigplan Notices, Vol. 17, No 7, pp. 24-31.

[3] Sajeev A. S. M., and Hurst, A.J., (1992), "Programming Persistence in X", IEEE Computer, Vol. 25, No. 9 , pp. 57-66.

[4] Dearle, A., Rosenberg, F.A., Henskens, J., Vaughan, F., and Maciunas, K.J., (1992), "An Examination of Operating System Support for Persistent Object Systems", Proceedings of the 25th Hawaii International Conference on System Sciences, Vol. 1. Editors V. Milutinovic and B. D. Shiver, IEEE Computer Society Press, Hawaii, USA, pp. 779-789.

[5] Kemikli, E., Erdogan, N., (1997), "Persistent Operating Systems", *Proceedings of the 12th International Symposium on Computer and Information Sciences*, Antalya, Turkey, pp.76-84.

[6] Shekita, E., and Zwilling, M., (1990), "Cricket: A Mapped, Persistent Object Store", Proceedings of the 4th International Workshop on Persistent Object Systems Design, Implementation and Use, pp. 89-102.

[7] Vaughan, F., and Dearle, A., (1992), "Supporting large persistent stores using conventional hardware", Proceedings of the 5th International Workshop on Persistent Object Systems, San Miniato, Italy, pp. 185-192.