Augmenting Object Persistency Paradigm for Faster Server Development

Erdal Kemikli and Nadia Erdogan

ITU Electric and Electronics Faculty, Department of Computer Engineering, Maslak, 80626, Istanbul, Turkey
erdalk@otokoc.com.tr, erdogan@cs.itu.edu.tr

Abstract. Server program development is a difficult task due to extra requirements on synchronization, communication , and availability. This paper defines an extensible and tailorable computing system model, EPS, which proposes a faster server program development cycle to programmers. The resulting system is suitable to be used as a base for an extensible system for different types of application domains. EPS is implemented by the following components; inter-object communication service (IOC), client object library (COL), active object library (AOL), naming and protection server (NPS), object server (OBS), and a preprocessor on Linux operating system.

1 Introduction

The concept of persistency suggests that data in a system should be able to persist (survive) for as long as that data is required [1]. As a result, persistent systems provide a uniform abstraction for data management. This uniform abstraction removes the task of converting data model for file to memory or vice versa, therefore, saving programmers from considerable amount of program development task. A research by designers of PS-Algol showed that %30 of program code consists of these load/save/convert operations [2].

An extensible system is one that can be changed dynamically to meet the needs of an application [3]. Most operating systems are forced to balance generality and specialization. While a general system is suitable for many types of applications with some effort, specialized systems run a specific type of application with fewer problems. Implementing a new server program as a way of extending an existing system has several requirements specific to server development; synchronization, communication, and availability.

This paper describes a persistent, extensible and tailorable computing system model, which utilizes the object persistency paradigm for easier server development. This model consists of two basic abstractions. A passive object contains only persistent data while an active object is persistent data and methods used to modify it, and acts as a server that extends the original system behavior. The resulting system is suitable to be used as a base for an extendible system with new server functionality.

The Extensible Persistent System (EPS) [4] supplies facilities that will help developers to extend the functionality of the system with better modeling based on passive/active object abstraction, and simplify programming task with persistency

T. Yakhno (Ed.): ADVIS 2002, LNCS 2457, pp. 327–335, 2002.

support and active object development facilities. Moreover, the resulting extended system will not need extra system administration tasks and complex configuration management. Therefore despite its revolutionary nature as a unified application program development and system model, with a new approach to system abstraction, EPS is easy to learn and adapt since it closely resembles the most common computing environment of today, namely UNIX.

The EPS model explained in this paper is composed of several components in different layers. On the highest level, a new programming language, EPS-C that is an extension of the existing ANSI C programming language is defined. Extending an already available language will flatten the learning curve of software developers. 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) is linked to the program. COL 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. Active object library (AOL), which supplies basic active object functions to developers, is linked to active objects during the link phase. 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 message exchange. This functionality is implemented as a run-time library and linked to every process with the need of communication.

2 Programming in EPS

Programming in EPS is quite similar to programming under UNIX 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 its support of an inter-object communication, which is easy to use yet powerful and flexible. The last difference is the support for active object development, which simplifies system programming and prevents programmer from making errors.

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. A typical EPS-C program starts with the declaration of persistent variables. One of the first few lines of executable code includes a call to EpsInit() function, which will initializes the persistent process environment by loading the persistent objects. After the intended operations are conducted, the program is exited by calling the EPSCommit() function, which synchronizes data and releases resources.

2.1 EPS-C Programming Language

The primary programming language is the C programming language, which is enhanced with some extensions. This extended syntax is called as EPS-C to distinguish it from the standard C programming language. This implementation allows application developers to use the C programming language alone if they prefer. The main enhancement to C language syntax is the addition of a reserved keyword, "$", to declare the persistent nature of variables and a number of standard EPS functions, accessible through a library.

The loading of persistent objects are handled either by explicit function calls or the necessary statements are inserted into program code by the EPS-C preprocessor. While the requirement for explicit function calls may look like an extra task to programmer, we should be aware of the fact that the programmer shall declare the binding between the persistent object and the database in any case. These two options let programmers decide on the loading time and allows for flexibility to design programs for different purposes and characteristics.

Persistent variable types can be any simple C variable type or structures declared as user defined types. Structures should be one level structures consisting of simple C types, and every type definition of persistent variable shall be stored in a separate file with the same name. As a part of EPS design philosophy, there are no specific types of files for the formats used. Instead, the already available type declaration capability is used for the type checking facility of passive persistent objects.

2.2 Active Object Development

Active objects are developed with the support of by an active object library (AOL). AOL implements a main server loop to receive and handle client requests. This library also includes standard commit and rollback functions. Active objects are part of EPS and, in a sense, they act as client objects with regard to their relationship with NPS and OBS.

Programmer shall follow four steps to complete the development process of a functional server, ready to run:

a) Write a function for each type of message server accepts. This function shall satisfy the demand of the client sending this specific message.
b) Fill the entries of a static function pointer array with the addresses of these functions:
   int (* f[MAXFUNCTION]) (char *, int) = {AobCommit, AobRollback, NULL, NULL, NULL, NULL, AddItem, GetItem};
   In this example, AddItem and GetItem functions are developed by the application programmer. The first six locations of the array are used by AobServer function, and shall be left unchanged.
c) Write a main function that initializes an EPS session (EpsInit) and then calls AobServer function (AobServer(f)):

```
…
EpsInit(&session,"warehouse",lCapF);
AobServer(f);
EpsClose(&session,COMMIT);
…
```

d) Compile and link program with active object library (epsaob.c) and client object library (epscob.c)

Now, we have an active object, actually a server object which responds to request messages from client objects by executing the corresponding server function.

3 EPS Design

EPS, built on UNIX, has a multi-tier architecture that is designed to extend naturally based on needs. The system is composed of three basic components; Naming and Protection Server (NPS), Object Server (OBS) and Client Object library (COL). An underlying messaging facility, inter-object communication 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).

For the communication between objects, a high level interface matching the general requirements of the system is provided. Inter-object communication is the bond among different objects in EPS. The primitives are designed to support object-to-object synchronous(Request, Receive) and asynchronous (Send) communication, and the choice is left to the programmer. Message structure is based on object communication paradigm, hiding the IPC details from programmer.

The design of inter-object communication subsystem is based on Unix IPC mechanisms. Primitives of this high level interface are used both in systems development and application development, so they are available to application programmers as well.

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. Remote parts of the services are requested from EPS servers and active objects through the inter-object communication subsystem.

COL creates and manages a client session, which is a term used to define the local management scope of EPS and also the storage area for loaded persistent objects. It is actually the primary memory part of the EPS persistent store. When the client object is loaded, a new session is initialized in the process address space. Loaded persistent objects are stored and handled in the client session without the intervention of programmer.

Session object is used to hold information on the current process such as its active object id, and loaded passive objects. A session object actually consists of two parallel linked lists. Parallel nodes of the session store the original and modified objects, thus allowing the system to realize commit and rollback requests.

Client Object Library also manages the local capability store, which is the storage area for previously obtained capabilities. When a request for an object is to be made, a client object first checks if it already has the capability for that object in its local capability store. If it is not available, then it requests this capability from NPS.

3.2 Naming and Protection Server

Naming and Protection Server (NPS) is implemented in the form of a server process. It runs as a background process and is responsible of the security and synchronization of object access. Each request for object access is received and handled by NPS (Figure 1).

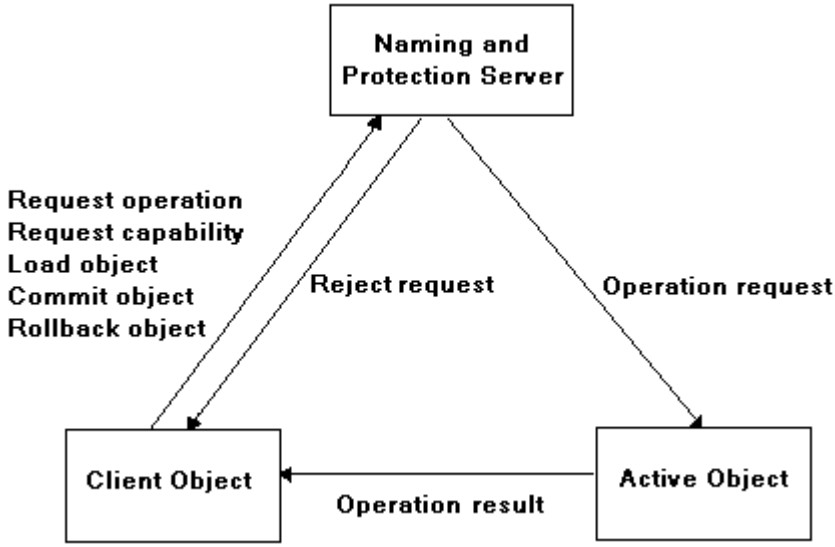

Fig. 1. Message Flow among NPS and Other Objects

A data structure, the InUse table, is used to keep the records of all currently used/loaded objects. There is only one copy of InUse table, so all object access control and synchronization is controlled with this information safely, without any further investigation by objects other than NPS. It also holds the entire capability information. Since the number of loaded objects cannot be anticipated, its size is dynamically adjustable.

Requests to NPS are sent and received via inter-object communication subsystem. Request messages for active object services are also handled by NPS, but message contents are defined in the active object and there is no special treatment for different active object requests, other than a control on access rights.

3.3 Access Synchronization

The target object name in every request for loading a static object or for an operation from an active object is first converted into a Persistent IDentification (PID) by matching the requested target object name with the existing capabilities, and then evaluated by NPS.

If a request is validated, then a record is inserted into the inUse table for the persistent object. The requested operation is forwarded to OBS if necessary. For instance, a Load Passive Object request is sent to OBS to be realized. In this case, the result of the load operation is sent to the client by OBS. A client process can access multiple persistent objects simultaneously. The access control and protection mechanisms will be used for the entire persistent object. A finer grained control, which would enable client processes to control a subset of an object such as a part of a tree structure, is not supported.

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 multiple processes, therefore access synchronization has to be explicitly managed. When a request for a passive object is validated, the access level of the inUse row is set. When a new request comes for the same object, the current access level and state transition rules are used to decide if it is possible to satisfy this request. If it is not accessible, than NPS sends a rejection message to requesting client object. Client object reports back when it is done with the requested object, so the access level is reset.

3.4 Object Server

Object Server (OBS) which can be considered as a part of the operating system kernel due to its functionality, is the persistent object store of EPS. Physically it is implemented as a separate module and runs as a background process. The Object Server handles the storage and retrieval of passive and active objects. While active objects consist of data and methods, a passive object contains data but does not have methods to modify it.

When a request for an object load operation is received, OBS first finds out if the target is a passive or an active object, then loads it as appropriate. Hence OBS ensures the availability of servers to clients, loading the active object if it is not already present. OBS accepts requests only from NPS.

Consistency of multiple objects in memory is maintained with a “shadow object” mechanism. The objects are kept in the memory in two forms; original and modified. When a “commit” statement is issued, the modified versions are copied into the original versions and all heap values with modified flag set are written to the file system. Therefore, it becomes possible to maintain consistency of interrelated local passive objects.

Service requested from an active object is another important issue. When a client object requests a method from an active object as part of its transaction, the system automatically sends rollback /commit requests to the active object. Active object library offers basic commit and rollback services, but issues like controlling

remodification of a modified but not committed object are expected to be handled by programmer.

Shadow files technique is used to reduce the risk of creating an inconsistent database. In shadowing, the modified object 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. Since it brings a certain performance penalty, this function is made optional and specified when starting Object Server (OBS).

Persistent objects are stored on the disk for long-term storage and are loaded into memory by OBS. A passive object loaded into the process memory is represented by its root object, which is the only variable accessible directly through the program instructions, as the root of a tree or the head pointer of a list structure. 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 includes a persistent root. Each persistent object group is kept in different file and includes a homogeneous structure inside its file. These groups are called as persistent databases. The type information of the persistent objects is kept inside the persistent root. While the simple types such as integer or character are defined directly in the persistent root header, complex user defined types are referenced in the header and are kept as type definition include files separately.

3.5 Active Object

Active Objects (AOB) are not part of the core EPS. 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 transparently loads active objects whenever they are needed. 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. Some methods are mandatory if a program is to be named as an active object:

- Rollback operation
- Commit operation
- Execute local method

In addition to these mandatory operations, an interface that accepts requests from other objects is a natural part of an active object. The standard interface of an active object is implemented by the active object library to ease programmer's task of server development. This library supplies messaging and synchronization service, and a framework for server processing cycle.

3.6 EPS Security Architecture

In EPS, security is based on capabilities. EPS capability structure includes three Ids used for Access rights limitation, persistent object id and name, rights given by this capability, and fields to determine if this capability is transferable or not. Each capability presented by the requesting process will be checked if its clientObjectId field matches the presenter, or it has the special "can be used by every object" value in the clientObjectId field and the capability field "transfer" is true.

The server always checks to see if target id is valid or not, so that an unauthorized copy of the capability cannot be used to gain access to the related resource.

Capability objects can be stored in two different locations; local capability stores for each client, and a common capability store managed by NPS. A process can always load the capabilities in its local capability store while it can only retrieve capabilities in public store if it has the required capability for them. Once the object is activated, that is loaded into memory, the capability is also loaded into the InUse table.

4 Programmer Productivity

Programmer productivity has been a subject of interest for many years and different methods such as correlating programmer productivity to lines of code and function point counts have been developed [5,6]. Productivity is estimated statistically by Capers Jones [7] based on function points and source line of code (SLOC)[8].

To investigate the advantage (if any) of program development in EPS over conventional Linux environment, a simple program for the following scenario is developed. Let's assume that a scientist collects some data everyday, for a number of days. He wants to save these data daily in a file so that at the end, he can quickly analyze data. The two programs, "tableproc.c" and "tableproeps.epsc" to satisfy this requirement are written in C programming language and EPS-C, respectively. Two programs do exactly the same task;

- load a table of data from secondary storage
- display table content
- prompt for new data
- save modified table to secondary storage

Table 1. Comparison of C and EPS-C Program Source

	C Language	EPS-C	Percentage of Difference
Source Line of Code	89	65	27%

The resulting numbers of source code lines are compared in table 1. EPS-C code is significantly (27%) shorter when compared to C code performing the same task. The percentage of difference is in line with the results of other EPS program comparisons and similar to observations of developers of PS-Algol [2]. The percentage difference of the number of source lines of code can be accepted as the measure of the advantage gained by using EPS.

Another interesting issue is the types of code sections and relationship of different types of sections to the level of gain. If this relationship can be modeled, than it will be possible to inspect program requirements and decide whether it is better to use EPS-C or not, considering run time and development productivity tradeoff.

Two factors influence the advantage gained in the persistent environment:

- Number of persistent objects (N)
- Complexity of persistent objects (C)

We can formulate the gain function as:

$$Gain = f(N, C) . \quad (1)$$

5 Conclusion

Productivity gain in formula 1 is based on standard client programs, and does not take into consideration the extra support of active object development. While server productivity gain shall be greater, we still need to conduct large number of experiments in order to have conclusions based statistical results.

We believe that a system with elaborate support for program development is an important step in society's computer usage by supporting more software functionality development in shorter time and with less effort. Our future research will provide more data on effects of persistency in server development task. Consequently, in the near future, EPS shall be packaged as an easily installable system with support documentation to facilitate an experimentation process.

References

1. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W. and Morrison, R., 1983. An Approach to Persistent Programming. The Computer Journal, 26 , 360–365.
2. Cockshot, W. P., Atkinson, M. P. and Chisholm, K. J., 1984. Persistent Object Management System. Software- Practice and Experience, 14, 49–71.
3. Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., Eggers, S., 1995. Extensibility, Safety and Performance in the SPIN Operating System, Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, USA, 1995, 267–284
4. Kemikli, E. and Erdogan, N., 1999. Extendible Persistent System., Proceedings of 3rd WSES/IEEE/IMCS International Conference on Circuits, Systems, Communications and Computers (CSCC'99), Athens, Greece, July 4–8.
5. Albrecht, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D., 1983. Software function, source lines of code, and development effort prediction: A software science validation, IEEE Transactions on Software Engineering, SE-9/6, 639–648.
6. Matson, J.E., Barrett, B.E., and Mellichamp J.M., 1994. Software Development Cost Estimation Using Function Points, IEEE Transactions on Software Engineering, 20/4, 275–287.
7. Jones, C., 1986. Programming Productivity, McGraw-Hill, New York.
8. Ragland, R., 1994. Function Point Counting Practices Manual Release 4.0, International Function Point Users Group (IFPUG), Atlanta.