Methodology to Implement an Amoeba Complex Object Server*

Wouter B. Teeuw Henk M. Blanken

Department of Computer Science, University of Twente
P.O. Box 217, NL–7500 AE Enschede, The Netherlands

Abstract

This paper describes the methodology we use in the design of a complex object server application for the Amoeba distributed operating system. We use the top-down design that was suggested by Parnas, in which a model is turned into an implementation by gradually adding details. We describe the abstraction levels that show up if going from a specification of the behaviour towards an implementation, and we show our methodology in which performance will be measured (instead of estimated) whereas the system has not yet been functionally implemented in its entirety.

1 Introduction

Within the Starfish project, several Dutch universities are cooperating in the design, implementation, and application of a transparent distributed computing system. The distributed operating system *Amoeba* [1] is used as a base to experiment with. The University of Twente studies the performance aspects of an extensible complex object server for Amoeba. The object server, which we call ACOS (Amoeba Complex Object Server) has to be seen as a database application for Amoeba.

Complex objects are used in so-called *non-standard* database application areas, such as geographic information systems, robotics, cartography, and CAD/CAM. Complex objects are data objects that are both highly structured, and large in size. These large clusters of structured data are a unit of manipulation. The structural aspects of complex objects can easily be captured in object-oriented data models [2]. But, except for rich data structuring capabilities, in non-standard database applications a high performance is generally required as well. Therefore, the physical design of a complex object server needs to be efficient enough to allow a fast retrieval and processing of the complex objects.

Rather than designing and implementing our entire system, discovering some performance bottlenecks, and solving these performance bottlenecks by either last minute changes, or making a new design, we aim at detecting performance bottlenecks as soon as possible. Therefore, we use the top-down design methodology as described by Parnas [3–5]. Parnas describes a design methodology in which the close relationship between designing computer systems and simulation is emphasized. The idea is that a simulation model is evolved to a real system by gradual addition of detail. The same idea is expressed by Randell [6], who introduces several levels of abstraction. Each design decision is handled at the appropriate level of abstraction. For both Parnas and Randell, the model is not only a true representation of the system to be designed, it *is* the system.

This paper describes our approach and is organized as follows. In Section 2 we outline a design methodology for computer systems in general. In Section 3 we focus on complex object servers, and we show the successive modelling steps for designing them. In Section 4 one of these steps, the physical database design, is again decomposed into a number of steps. Up to this section our paper is rather general. Issues that are specific for our situation will be marked between the keyword *ACOS* and a box □. Finally, in Section 5, we present the current status of our project.

2 Top-down design methodology

Parnas [3] studied a large number of student design teams, working on the same project, all completely designing a working system. A number of observations showed up. First, a bottom-up approach, designing small parts and trying to put them together into a total system, did not produce working systems. Second, for the successful groups, all work done before they hit upon the top-down approach could be classified as false starts. Finally, with the top-down approach, the work progressed well until it became time to test the unit composed of the components designed separately by various members of the design group. Either the components did not work together since everybody had its own idea of what each component should do, or, if the individual components met their specification, the combined system failed since there had been

*The investigations were partly supported by the Foundation for Computer Science in the Netherlands SION under project 612-317-025 nicknamed Starfish.

0-8186-2812-X/92 $03.00 © 1992 IEEE

no means of verifying that the initial structuring of the problem was correct and feasible.

A design methodology arose, based on three important points. First, the design should begin with a specification of the overall behaviour of the computer system. From the specification, one proceeds to a means of achieving it by either lowering the level of abstraction or functionally decomposing the components. Applying this technique recursively to each component brings us from the purely behavioral specification to the purely structural final design.

Second, to avoid a situation in which an oversight in an earlier stage is not detectable until all components have been designed and are being tested, simulation should be used. Also, because at times some components of the system will still be in a relatively abstract form, while others will have proceeded through one or more levels closer to reality of the implementation, the ability is needed to have several levels of abstraction resident and interacting within the simulation.

Third, since the similarity between the simulator and the operating computer system is so large, the simulator or model must become the system. The simulator and the model are so close that it is not meaningful to have a parallel development of the system and its model, with the model following the system as it develops. Rather, the simulation model should evolve into a real system by gradual addition of detail. In this way double effort is avoided.

We use this approach in ACOS, that is we top-down implement the system while still designing it. The *critical* parts of the system are implemented, while other parts, which have not been designed yet, are simulated or replaced by dummies (their design is probably nothing more than a description of inputs and outputs). The performance of this functionally partially implemented system can be measured and will guide further design. That is, modelling, either analytical or by simulation, and performance measurements will be integrated into the design of the system.

The described top-down strategy is not equal to a common used approach, in which the system is designed (on paper) in a top-down way, each time distributing the functionality over more detailed components, until the design is on a level of such great detail that it is almost an implementation, whereupon in a bottom-up way the functionally complete system can be implemented on the hardware. That is, smaller modules are implemented and tested, and will be composed into larger ones. With such an approach, there will be some kind of performance estimation during the design, and after implementation the performance will be measured. The performance evaluation becomes a design verification.

The drawback of this approach is that whether the functionally complete design of the system meets its performance requirements will not be discovered until the system has been built and used in its operating environment. At that stage system modifications can be extremely difficult or costly. Even if it is possible to measure the performance of smaller components at an early stage, the optimization of individual components will not automatically lead to optimization of the whole system. Not to mention the effects of an incorrect design. A slight oversight in the design, probably made on a high level of abstraction, will not be discovered earlier than after the implementation.

3 Complex object server modelling

3.1 Conceptual modelling

The first step in the design of a distributed complex object server is the *conceptual modelling*: making a conceptual description of the complex objects. For, in order not to confuse the question of behaviour with the question of how to construct the system, a precise definition of the behaviour of the server is needed. A problem must be defined *before* solving it. The behaviour can best be described by specifying the database data (objects and attributes) and their relationships, as well as the constraints, queries, and update transactions for these data and relationships.

Besides statements that *specify* the behaviour, there are also *design criteria* for the behaviour of the system, and *restraints* on the behaviour of the system. General design criteria are a minimization of the average query response time or a maximization of the system throughput. Statements that restrain the behaviour of the system are, e.g., maximum disk capacities or a maximum response time.

ACOS: We use the database specification language TM for conceptual modelling [7]. A design criterion is to minimize the average response time of the server. The behaviour of ACOS is restrained by the architecture and characteristics of Amoeba [1].

In TM, objects belong to classes that are arranged in an inheritance hierarchy. The objects of a class are characterized by attributes, which are either atomic, or structured in value. In the latter case, the attribute is typed as a list, a set (power type), a tuple (record type), or a variant. Attributes values are chosen from basic types, such as character, integer, etc. Among the basic types are all object types that appear in the database as well. In this way the object occurrences of the classes are related to each other. Constraints and methods can be defined on the level of object, class, and database extensions. □

3.2 Logical modelling

As a second step, the *logical modelling* is performed. The goal of logical complex object design is to map the conceptual design on an implementable data model. That is, the data is arranged in a non-redundant, integrable, and generally manageable way. It is a first step towards implementation. Often used data models are the record based models, such as the relational

```
TUPLE-OBJECT Person = {⟨
        identifier:     OID,
        name:           STRING,
        date_of_birth:  ⟨  day:     INT,
                           month:   INT,
                           year:    INT ⟩,
        partner:        REF(Person),
        children:       {REF(Person)}
⟩}
```

Figure 1: Declaration of a Person tuple-object.

model. Lately, object oriented- and logical data models get much attention. Part of each data model are the data manipulation tools, generally consisting of a query language and several update methods.

ACOS: We use the nested relational model for logical modelling [8]. A so-called *tuple-object* is constructed by applying the list, set, and tuple constructors an orthogonal way to basic values. The top level construct is always a tuple. Connected to each tuple-object instance is a unique system generated identifier that distinguishes the object from all others in the system. Among the basic types is a reference type as well, which makes tuple-objects the unit of sharing among a number of composed objects. Data is not shared among tuple-objects. Figure 1 shows an example tuple-object. □

3.3 Physical modelling

Finally, we have the *physical modelling*. In this step we go towards implementation. The goal is to implement the data in such a way that queries are efficiently supported. The physical modelling consists of several steps. Each step represents an abstraction level that hides the implementation details of the next levels of lower detail: the implementation details of lower levels are *transparent*. We start with the result of logical modelling as the highest level of transparency. If data processing, communication, and I/O would be infinitely fast, the physical design would be simple. But, devices have a limited speed and the data structures and software are often fairly complex, leading to a performance that is often lower than desirable or acceptable.

4 The physical design

4.1 Separation of programs and data

A first step is to make a clear distinction between what will be stored as data and what will be implemented as programs. Although in most cases the choice might be obvious, this decision remains dependent on system and workload aspects. For instance, it is most likely that an attribute *date of birth* will be stored as data and the derived one *age* as code. For, *date of birth* is immutable (will not be updated), and *age* can 'easily' be calculated from it. But, although the (imaginary) attribute *City* can be derived from the attribute *Zip-code*, *City* will be materialized since it will be 'often used' and its derivation from *Zip-code* is 'complex' or might even require an additional table. Such a motivation is based on physical aspects.

4.2 Fragmentation of the data

A next step in the physical design process is the fragmentation of the (nested) relations (or in general: of the sets of records). Fragmentation enhances performance by allowing the access to data units that are as small as possible for the tasks that need them. On the other hand, joins and unions will be necessary to reconstruct the relation from its fragments. Horizontal fragmentation is the partitioning of the tuples of a relation into subsets. Vertical fragmentation is the clustering of the attributes of a relation into groups. Three conditions must be met when defining fragments. First, all the data of the relation must be mapped into the fragments (completeness condition). Second, it must always be possible to reconstruct the relation from its fragments (reconstruction condition). Third, fragments have to be disjoint so that the replication of data can be controlled explicitly at the allocation level (disjointness condition).

ACOS: Within ACOS an identifier mechanism guarantees that tuple-objects can always be reassembled from their fragments. Therefore, the reconstruction condition is superfluous. We do not take care of the disjointness condition as well! For, complex objects are used in non-standard databases for which performance is often critical. Leaving out the disjointness condition enables us to introduce redundant storage of data, not only by simply storing identical copies of a single fragment, but also by replication of data within different clusterings. This might improve the performance of the system. □

4.3 Allocation of the fragments

The allocation problem may be described as given a workload and a certain cost function to be optimized, determine for each fragment the nodes on which this fragment will be physically materialized. Since a query may use many fragments and may be executed in parallel, fragments cannot be allocated independently and we have to allocate the operations on the fragments simultaneously. Notice that strong and complex relationships between the fragments may exist. The allocation problem consists of two basic sub-problems. The fragment replication problem determines the number of replicas of a fragment to be stored. Network placement determines the nodes on which these copies will be stored.

ACOS: With ACOS the goal of the allocation step is to minimize the overall response time for queries

and updates. We have a shared-nothing system with a local area network. For such systems the disk IO seems to be the bottleneck [9]. Therefore, we try to use the allocation to enlarge the effective I/O bandwidth by parallel disk I/Os. □

4.4 Mapping of fragments on storage models

A next step is mapping the fragments on storage models. The minimum unit of retrieval, and the clustering of these units into files is determined. Also, the secondary indices are chosen. The relations between the basic units, as well as their internal structure, have to be stored as well, either as meta data or within these units. We still have physical device independence. Using a different storage device does not alter the decisions taken during this step.

ACOS: The basic storage unit in ACOS is a (nested) tuple. The internal structure information will be stored within this tuple. Also, an identifier keeps inter-object relationships. For example, with a normalization of the hierarchically structured tuple-objects, each flat tuple will get a tuple identifier containing the identification of itself, its parent tuple, and its root tuple. Also, tuples belonging to the same tuple-object will be clustered. □

4.5 Storage on physical devices

Finally, the storage model has to be implemented on the physical storage devices. Concepts like physical blocks, tracks, and cylinders show up.

ACOS: In ACOS, we use the Amoeba disk server called bullet server. This server is based on caching and immutable files that are variable in length. □

5 Current status and conclusions

The current status of our project is as follows. We started with an extensive analytical performance evaluation (e.g., [10]). We derived formulas for disk I/O, network communication, and CPU time and compared several strategies for parallelism, query execution, and communication. Based on the results of this performance evaluation, we are currently designing and implementing a simplified version of the object server ACOS. The critical parts (according to our analytical performance evaluation) that get most attention are the storage of objects on disk, the start of all processes on all nodes, and the communication. For implementation, simulation, and evaluation test, we have a seven node system. Performance test results are not available yet, but are expected in the near future.

Most of the ideas presented in this paper are, by the standards of Computer Science, old. They are like the wheel, which is often reinvented because it is a good idea. But, the authors who reported on the students design experiment and proposed a design methodology [3] felt quite strongly that the described problems were not confined to students in courses, but were common in professional circles as well. Often, we get the impression that, after more than twenty years, still the same errors tend to be made.

A top-down implementation of a complex object server suggests an empirical approach. First we have an analysis of what could be the best design. That is, we define our hypothesis. Then by analytical performance evaluation, simulation, or experimental measurements we test our hypothesis. Depending on the results, the design model is improved or adapted, and the same cycle starts again. Each time the model is enhance like physical scientists and engineers make use of theories. That is, each time the model, which is a simplified description of reality, becomes inadequate we have to search for a better one.

References

[1] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse and H. van Staveren, "Amoeba — A Distributed Operating System for the 1990s," *Computer* **23** (5), pp. 44–53, May 1990.

[2] B. R. M. van den Akker and H. M. Blanken, "Geographic Data Modelling in TM," in *Advances in Data Management. Proceedings Third International Conference on Management of Data, Bombay, India, Dec. 12–14, 1991*, P. Sadanandan and T. M. Vijayaraman, Eds. New Dehli, India: McGraw-Hill, pp. 107–126, 1991.

[3] D. L. Parnas and J. A. Darringer, "SODAS and a methodology for system design," in *Proceedings AFIPS 1967 Fall Joint Computer Conference, Anaheim, CA, Nov. 14–16, 1967*. Washington, DC: Thompson Book, pp. 449–474, 1967.

[4] D. L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering (version 2)," McMaster University, Hamilton, Ontario, Canada, CRL-237, Sept. 1991.

[5] D. L. Parnas, "The Application of Modelling to System Development and Design," in *Proceedings International Computing Symposium, Bonn, Germany, May 21–22, 1970*, W. D. Itzfeldt, Ed. Frankfurt, Germany: German Chapter of the ACM, pp. 134–142, 1973.

[6] F. W. Zurcher and B. Randell, "Iterative Multi-Level Modelling — A Methodology for Computer System Design," in *Proceedings International Federation of Information Processing Societies 1968*. pp. 138–142, 1968.

[7] H. Balsters, R. A. de By and C. C. de Vreeze, "The TM Manual," Universiteit Twente, Enschede, The Netherlands, working document, Nov. 1991, (version 1.21).

[8] H. -J. Schek and M. H. Scholl, "The Relational Model with Relation-Valued Attributes," *Inf. Syst.* **11** (2), pp. 137–147, 1986.

[9] D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of Database Processing or a Passing Fad?," *ACM SIGMOD Record* **19** (4), pp. 104–112, 1990.

[10] W. B. Teeuw and H. M. Blanken, "Joining Distributed Complex Objects: Definition and Performance," *Data & Knowl. Eng.* **9** (1), Jan. 1993, (to appear).