Design and Implementation of the Amoeba Complex Object Server ACOS*

Frank Sauer *Wouter B. Teeuw* *Henk M. Blanken*

University of Twente, Enschede, The Netherlands

Abstract

In this paper we describe the initial design and implementation of a database application for the Amoeba distributed operating system: the complex object server ACOS. We use the top-down design methodology that was suggested by Parnas, in which a model is turned into an implementation by gradually adding details. Therefore, not all components of the system need to be implemented on the same level of detail at the same time. Those components that are considered to be a bottleneck will be fully implemented, whereas other components will still be simulated or replaced by dummies. We consider two main bottlenecks: starting all processes that together execute the parallel database query and the I/O of the data stored on the several disks in the system.

Keywords: Amoeba distributed operating system, complex objects, design methodology, parallelism, performance aspects, physical database design, shared-nothing databases.

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* [MRTv90,TRSS89] 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 form a unit of manipulation. The structural aspects of complex objects can easily be captured in object-oriented data models [AkBl91]. 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. Examples of complex object systems are DASDBS [Paul88] and AIM-P [PiDa89]. These systems are centralized however.

In this paper we describe the design and implementation of ACOS. Rather than designing and implementing the 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 as well as Randell [PaDa67,PaMa91,ZuRa68]. The idea is

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

that a simulation model is evolved to a real system by gradual addition of detail. The model is not only a true representation of the system to be designed, it *is* the system.

Two performance bottlenecks get special attention in this paper. First, the client-server based Amoeba system a query tree will be mapped on many processes in order to be executed. Experiences with the PRISMA database system [ABFG92] have shown that creating many processes may be a bottleneck. Second, for the shared-nothing database systems we aim at, disk I/O is a bottleneck due to the fact that during the past years CPU performance has improved from about 1 mips in 1983 to well over twenty mips in 1990, whereas disk drives have improved their response time and throughput by only about a factor two in the same period [DeGr90].

The remainder of this paper is organized as follows. In Section 2 we describe the problems and goals in the design of a complex object server for Amoeba. In Section 3 we outline our methodology for the design and implementation of ACOS. In Section 4 we give some information on the Amoeba OS, the architecture of the system on which ACOS is currently being implemented. In Section 5 we describe the design of ACOS itself, whereupon the design and implementation of the two earlier mentioned bottlenecks is the subject of the next sections: Section 6 and Section 7. Finally, in Section 8, we present the current status of our project, as well as some preliminary conclusions.

2 A complex object server: problems and goals

We take a nested or NF^2 tuple [ScSc86] with object references as an example of a complex object. A complex object is a tuple (a record) with both atomic-valued attributes, and relation-valued attributes. The latter ones are sets of tuples, with each tuple containing atomic- and relation-valued attributes again. So, the complex objects are hierarchically structured. An object identifier (OID) distinguishes the object from all other objects in the system. The OID may be used by the objects to refer to each other. In this way, relationships between objects are established.

The problem is how to fragment these objects, and how to distribute the fragments over the system nodes in order to achieve the, for non-standard database systems required, high performance. For example, the entire object may be stored directly into a single storage unit, like in the DASDBS system [Paul88], or the hierarchically structured objects may first be normalized (i.e., vertically fragmented), whereupon the fragments will be stored separately. Also, related objects or fragments can be stored on the same node or not, objects or parts of objects may be replicated, and so on. In short, we need an optimal physical design for the complex object server.

The way in which we want to attack these physical design problems is as follows. Since the performance of a complex object server is critical, it is our intention to experiment with the system performance while still designing and implementing the system. Therefore we use a design approach as described in the next section, which enables us to *measure* the performance even though the system has not yet been fully implemented. Based on analytical performance evaluations, we will simulate/implement several complex object fragmentation, replication, and distribution strategies, and measure the performance of the (partially implemented) system in order to guide further design.

3 Top-down design methodology

A commonly used design approach is that a 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. Then, in a bottom-up way the

functionally complete system will 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 point in time — when system modifications can be extremely difficult or costly — it may turn out that components designed separately by various members of the design group do not work together, since everybody has its own idea of what each component should do. Even if the individual components meet their specification, the combined system may fail since there has been no means of verifying that the initial structuring of the problem was correct and feasible. That is, a slight oversight in the design, probably made on a high level of abstraction, will not be discovered before the implementation is completed. Moreover, even if the overall design is correct, we still can not estimate the performance since the optimization of individual components will not automatically lead to optimization of the whole system.

In order to prevent such problems, Parnas [PaDa67] outlined a design methodology 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 design 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 still are 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 system 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 ACOSand 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 measurements on this partially implemented system will guide further design. That is, modelling, either analytical or by simulation, and performance measurements will be integrated into the design of the system.

4 The Amoeba architecture

The Amoeba distributed operating-system [MRTv90,TRSS89] is an object based system using the Client-Server approach. Client processes use *Remote Procedure Calls* (RPC) to request operations on objects, which are managed by server processes. Servers communicate with the outside world by creating a *port* and listen to that port with a call to `getrequest`. The server publishes the port, so clients can send their requests with a call to `trans` to that port. Finally the server sends the result of the RPC with `putreply` back to the client.

Objects are both identified and protected by *capabilities*. A capability is a placeholder for the port of the object's server, the object number, the access rights, and a cryptographic protection. Because of an cryptographic protection mechanism, capabilities can be managed outside the

kernel, by the user processes. Many of the services in Amoeba are offered by processes in user space, in the form of servers. The kernel offers memory management, multiple threaded processes, and handles interprocess communication. All other services, usually found in kernels, are in user space. An example of this is the directory name service. The directory-name server in Amoeba (called SOAP) is a server process in user space, translating names to capabilities. Note that each object in Amoeba is identified by a capability, and has no name. To make life easier for us humans however, the SOAP server offers a service that allows us to use readable (path)names for capabilities.

The Amoeba hardware usually consists of workstations, specialized servers (e.g. a file server), a processor pool and of course a network. Replication can be used to make the system fault tolerant. Our current configuration consists of one specialized file server (an Intel 80486 machine, with 64 MBytes of memory and a disk of 1.2 GBytes) and 6 workstations (Intel 80386 machines with 16 MBytes of memory and a 120 MBytes disk). The processors of the workstations are used to provide the computing power. We do not have a processor pool.

5 The design of ACOS

This section gives a general outline of the ACOS design, with a constant focus on the fact that we are building an experimentation platform for doing performance measurements and with the methodology described in section 3 in mind. Figure 1 gives an overview of the design in a dataflow diagram notation. Each bubble in this figure represents one type of server, which can be instantiated many times if necessary. Examples of multiply instantiated servers are the fragment server (one per disk) and the dataflow processors, which build up a graph of interconnected processes to execute a query. What follows is a short description of the several servers the system consists of.

Query Optimizer The query optimizer (QO) optimizes a query with respect to the fragmentation and allocation of the data, as known from the *Data-Dictionary Server*.

Query Scheduler The query scheduler (QS) is responsible for distributing the optimized query over the processors in the system. It starts and controls a possibly large number of *Dataflow Processes*. See section 6 for more details.

DataFlow Processor The dataflow processor (DFP) is a multi-threaded process performing a single database operation on a number of input streams, and sending the result to a number of output streams. See Section 6 for more details on DFPs and multi-threading.

Fragment Server The fragment server (FS) offers a disk service that permits the user of the disk (the DFPs) to identify fragments (parts of Complex Objects) by their OID. It hides the block service of the Amoeba virtual disk server. Each node of the system has one Fragment Server. See also Section 7.

Virtual Disk Server The Amoeba virtual disk server offers a block service with virtual block size, thus hiding the physical aspects of dealing with disks. Each disk is managed by its own virtual disk server.

Log Server An important aspect of an experimentation platform is gathering the results of measurements. The log server serves just this purpose. All other parts of ACOS send their results (timing, number of disk accesses, etc.) to the log server, which collects and stores them. The log information can be used later on to fine-tune the performance, e.g. by redistributing the data.

Figure 1: Flows of control information and data on a top-level description

Data-Dictionary Server The data-dictionary server manages a data-dictionary containing information concerning the structure, size, replication and distribution of complex objects.

6 A potential bottleneck: starting many processes

The query scheduler is responsible for the distribution of the optimized query over the nodes in the system. Its input is a Directed Acyclic Graph (DAG) that has been produced by the query optimizer and represents the query. For each atomic part of the query there is a vertex in the DAG. Atomic means that *one* operation is done on the incoming data streams from the children vertices. For each vertex in the DAG, the QS allocates a so-called dataflow processor (DFP) on a node of the system. Also, the QS connects the DFP's according to the query-DAG.

The algorithm to traverse a DAG and for each vertex starting a DFP is given in Figure 2. This algorithm — written in pseudo-code — recursively traverses the query-DAG, visiting each vertex exactly once. Note that the DFP processes have to be started in a bottom-up order, because each process must be able to find the capabilities (ports) of the processes they want to send requests to. We intend to investigate several strategies with regard to the initialization and communication activities of the QS and the DFPs [TeBl93a].

```
procedure  executeQuery(POINTER TO Vertex root)
begin
    comment is this a leaf-vertex?
    if (root→input = nil)
    then comment input from fragment server
         argv := setArgs(operation,nil,root→argv)
         executeProcess(DFP, root→node, QS, argv)
         mark(root)
    else  comment intermediate vertex, start children first
         child := root→input
         while (child != nil)
         do
             if (!marked(child) )
             then comment recursively execute offspring
                  executeQuery(child→v)
             else  comment nothing
             fi
             child := child→next
         od
         argv := setArgs(operation, root→input, root→argv)
         executeProcess(DFP, root→node, QS, argv)
         mark(root)
    fi
end  executeQuery
```

Figure 2: Algorithm to traverse the query-DAG and distribute the DFP's

6.1 Simulating queries

As already stated before, we want to be able to do performance measurements in an early phase of the project, possibly during the design, to have a short feedback between design and performance of possible implementations. To achieve this, it is necessary to simulate large parts of the system and implement only those parts that are of interest to the performance measurements. A large part of the final system will be concerned with the translation and optimization of user given queries. To measure the performance of the bottom layers of the system, it is not necessary to implement those parts. Therefore, we built a small library of C routines that can be used to manually construct an (optimized) query-DAG, which can be hard-coded (linked) into the QS. These routines build vertices, connect them and register the root vertex of each query with the QS, so the query can be executed once the QS is started. The data structures used to describe a DAG are listed in Figure 3.

Using a `nil`-pointer as input in a vertex, forces this vertex to connect to the fragment server of the node contained in `node`. The following calls are offered to create and connect vertices:

```
Vertex *newVertex(int node, Operations op, ... <operation dependent args> ...);
```

`newVertex` creates a new vertex, performing the operation contained in `op`. The related DFP will be executed on the node contained in `node`. The parameters of the database operation are given in the variable number of arguments following the operations opcode. Now consider the example query depicted in Figure 4. Assuming the disks are maintained by two different

```
typedef struct _VertexList
{ struct _Vertex     *v          // pointer to head of vertex list
  struct _VertexList *next       // for list construction
} VertexList

typedef struct _Vertex
{ VertexList         *input,     // pointer to list of input (child) vertices
  Operations         operation;  // operation to be performed
  char               *argv[]     // pointer to array of string arguments
  int                node;       // node number to be executed on
} Vertex;
```

Figure 3: The type-definitions for our query-DAG representation

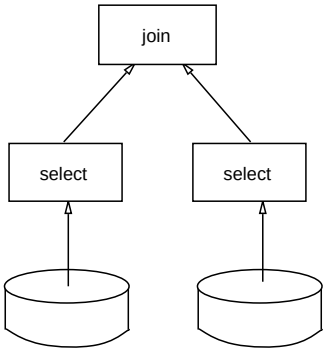

Figure 4: Example of a database query

nodes — say node 2 and 3 —, and node 1 is performing the join, this query is described by the following calls to `newVertex`:

```
s1 = newVertex(2,DFP_SELECT, ...);
s2 = newVertex(3,DFP_SELECT, ...);
j1 = newVertex(1,DFP_JOIN, ...);
makeChilds(2,j1,s1,s2);
```

The `makeChilds` function is used to connect the vertices s1 and s2 to j1 by filling the input list of j1 with pointers to s1 and s2. This function can be used to connect an arbitrary number of vertices, for it is defined with a variable number of parameters:

```
makeChilds(int nrOfInputs,Vertex *parent, ...);
```

A pointer to the root vertex of the query-DAG will be placed in a global list of queries, which are waiting to be selected by the QS for execution.

The arguments of the database operations are not important in this context. Even better, we don't have to perform any real database operations in order to get performance figures on the two potential bottlenecks we are concerned with. It suffices to simulate them. By simulating database operations we do not need to be concerned with the structure of the incoming tuples and the semantics of the database operations. Therefore, the DFPs can be implemented as fairly straightforward filters, reading input streams, filtering some data, and sending the resulting

operation	*arguments newVertex*	*description*
SELECT	`nrOfTuples`, `tupleSize`, `percentage`, `nrOfPageAccesses`	The DFP performing a SELECT starts by receiving `nrOfTuples` tuples of size `tupleSize`. It forwards a certain `percentage` of the received tuples to its output-streams. The last argument is used only by leaf vertices and states how many page accesses are needed to read a fragment from disk.
PROJECT	`nrOfTuples`, `tupleSize`, `percentage`, `nrOfPageAccesses`	In this case, the percentage states how much data of one tuple is sent through. The result tuples are therefore smaller than the received tuples.
JOIN	`nrOfTuplesLeft`, `nrOfTuplesRight`, `tsizeLeft`, `tsizeRight`, `tsizeResult`, `nrOfResultTuples`	The arguments state the number of tuples coming from the left, the right, the size of the tuples coming from the left, the right and the size of the result tuples, and finally the number of tuples to be forwarded to the output.
SPLIT	`nrOfProbs`, ...	`nrOfProbs` means number of probabilities. This number matches the number of output streams. The probabilities themselves are given in a variable number of arguments following `nrOfProbs`. The probabilities are used to implement a randomizer for choosing an output for each incoming tuple. Note that the probabilities have to add up to one.

Table 1: Examples of how to simulate the database operations

data to their outputs. Table 1 describes how the basic database operations are simulated. Note that this table does not contain all possible operations, it contains the operations that will be considered in our first experiments.

6.2 Implementation aspects of the dataflow processor

In Figure 5 we find the outline of the DFP in a dataflow diagram. An important aspect of the DFP is that is behaves both as a client and a server. A DFP is a server for the QS and for other DFPs that ask the DFP to do something. On the other hand, the DFP itself will request data from other DFP's or fragment servers and can therefore be called a client as well. To implement this behaviour and to guarantee a maximum throughput of data, the DFPs are implemented as multi-threaded servers. Each DFP has one *main* thread and several input/output threads. Each input and each output is managed by its own thread and the main (body) thread synchronizes with the input and output threads to see whether it is in the position to perform the database operation associated with the DFP.

After the creation of the DFP by the QS, it starts by creating a server port, which is sent to the QS for publication. It requests from the QS the whereabouts of its children DFPs by sending the input list contained in `argv` to the QS. The QS can translate this request into a list of capabilities of the children DFPs, because they published their capabilities before (due to the

Figure 5: The Dataflow Processor (DFP)

bottom-up traversal of the query-DAG).

After this connection phase, the DFP starts all its input and output threads and waits for a request to deliver some data. When this request arrives at one of its output threads (which perform the `getrequest`), the request is processed and transformed into requests to the children DFPs (the input threads will 'forward' the request to the children DFPs by means of a `trans`-call). When the children DFPs sent the replies (call to `putreply`), the data is processed and replied to the parent DFPs.

7 A potential bottleneck: the disk I/O

On the bottom of the query-DAGs we find the vertices requesting data from the fragment servers. The fragment servers manage the stored fragments of the complex objects on disk. A fragment is requested from the fragment server using the OID of the required tuple. This tuple may be scattered all over the system, so by requesting the tuple at every fragment server, all parts (fragments) are found and the tuple can be reconstructed.

In our system, an OID consists of three parts, and forms a <root, parent, self> tuple. Parts of the OID may be wildcarded in requests to the fragment server, so all nested parts of one tuple can be retrieved in one request.

Since the fragment server is built directly on top of the Amoeba virtual disk server, it is interesting to see what the interface of this virtual disk server looks like. the interface of the virtual disk server mainly consists of two calls:

```
disk_write(*diskcapability, L2BLOCKSIZE, start, nrofblocks, *buffer)
disk_read (*diskcapability, L2BLOCKSIZE, start, nrofblocks, *buffer)
```

The `diskcapability` defines the diskserver to communicate with. `L2BLOCKSIZE` means the ^{2}log of the actual block size, so a block size of 512 is defined with a `L2BLOCKSIZE` of 9. `start` defines

Figure 6: The Fragment Server (FS)

the first block to read/write, `nrofblocks` the number of blocks and finally, `buffer` is a pointer to the buffer containing the data to write (or a pointer to a buffer to read the data into).

Since this interface constitutes more or less raw disk I/O, the fragment server has to keep its own administration of free disk blocks, which blocks belong to which fragments, etc. In Figure 6 an overview of the fragment server is given.

The main task of the fragment server is to map requests containing an OID to a series of requests to the virtual disk server containing block numbers. Therefore, the fragment server has to keep a table containing <OID, block> entries. This table is kept on disk in the so-called *directory blocks*. The directory blocks form a linked list of blocks, starting at block zero of each disk. A directory block is defined by:

```
typedef struct _dirBlock
{ long  nextBlock;           // pointer to next block in the list
  long  firstFree;           // first free block on the disk
  struct _dirBlock *nextMem; // the list pointer in memory
  long  nrOfEntries;         // the number of directory entries in this block
  DIRENTRY  dir[DIRSIZE];    // array of directory entries
} DIRBLOCK;
```

`firstFree` is a pointer to the first free block on the disk and only has a meaning in block zero. The free blocks are also chained in a linked list using a nextBlock pointer at the beginning of the disk block.

The directory blocks are read in main memory at the start of the fragment server. The OIDs are put in a hash table to allow a fast translation from OID to blocks. The block number defined for an OID gives the first block belonging to the fragment. The rest of the data is found by traversing the nextBlock pointers found in each data block.

The structure of the data blocks is not defined in the fragment server. It uses the nextblock pointer to connect them, but the rest of the data has no semantics. We could have chosen for nested directories (per relation, per object, per tuple, etc.) but the chosen implementation with one directory level offers the most flexibility for experiments, because the data contained in the data blocks can always be interpreted as a second level directory by a higher layer of the system. One always has a choice between putting an OID in the main directory (fast access!) or in a nested directory (not defined yet). Another reason for making the fragment server as straightforward and dumb as possible is to allow a flexibility in storage models. Fragments belonging to objects stored with the *Direct Storage Model* will most certainly be structurally different from fragments stored in a *Normalized Storage Model* model [TeBl93b]. The fragment server doesn't care.

Because our prototype DFPs do not perform any real database operations, but merely consume bytes, the fragment servers can generate random data of the requested size. No real data needs to be stored in our first implementation.

8 Current status and conclusions

In this paper we described issues related to the design and implementation of the Amoeba Complex Object Server ACOS. We use a design methodology in which the system is implemented in a top-down way. First the critical parts are implemented, whereas other components are simulated. Which parts are critical and which parts are not has been decided based on a study of the literature, experiences from other projects, and analytical performance evaluations. 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. The design of the related ACOS components has been thoroughly described in this paper.

The current status of our project is as follows. We started with an extensive analytical performance evaluation (e.g., [TeBl93b]). 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. With the 'simulation-approach' taken in our design, it will be possible to do measurements on all relevant parameters in an experimental framework. This framework can be gradually transformed into a real system, using the results from the measurements to make design and/or implementation decisions. For implementation, simulation, and evaluation tests, we have a seven node system. Performance test results are not yet available, but are expected in the near future.

References

[AkBl91] B. R. M. van den Akker & H. M. Blanken, "Geographic Data Modelling in TM," in *Advances in Data Management. Proceedings Third International Conference on Management of Data, Bombay, India, December 12–14, 1991*, P. Sadanandan & T. M. Vijayaraman, eds., McGraw-Hill, New Dehli, India, 1991, pp. 107–126.

[ABFG92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten & A. N. Wilschut, "PRISMA/DB: A Parallel, Main-Memory Relational DBMS," *IEEE Transactions on Knowledge and Data Engineering* **4** (6), December 1992.

[DeGr90] D. J. DeWitt & J. Gray, "Parallel Database Systems: The Future of Database Processing or a Passing Fad?," *SIGMOD RECORD* **19** (4), 1990, pp. 104–112.

[MRTv90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse & H. van Staveren, "Amoeba — A Distributed Operating System for the 1990s," *Computer* **23** (5), May 1990, pp. 44–53.

[PaDa67] D. L. Parnas & J. A. Darringer, "SODAS and a methodology for system design," in *Proceedings AFIPS 1967 Fall Joint Computer Conference, Anaheim, CA, November 14–16, 1967*, Thompson Book, Washington, DC, 1967, pp. 449–474.

[PaMa91] D. L. Parnas & J. Madey, "Functional Documentation for Computer Systems Engineering (version 2)," McMaster University, CRL-237, Hamilton, Ontario, Canada, September 1991.

[Paul88] H-B. Paul, "DAS Datenbank-Kernsystem für Standard- und Nicht-Standard-Anwendungen – Architectur, Implementierung, Anwendungen –," Technischen Hochschule Darmstadt, Darmstadt, Germany, November 1988, (in German).

[PiDa89] P. Pistor & P. Dadam, "The Advanced Information Management Prototype," in *Nested Relations and Complex Objects in Databases*, S. Abiteboul, P. C. Fischer & H. -J. Schek, eds., Lecture Notes in Computer Science #361, Springer-Verlag, Berlin, 1989, pp. 3–26.

[ScSc86] H. -J. Schek & M. H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems* **11** (2), 1986, pp. 137–147.

[TRSS89] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen & G. van Rossum, "Experiences with the Amoeba Distributed Operating System," Vrije Universiteit, IR-194, Amsterdam, The Netherlands, July 1989.

[TeBl93a] W. B. Teeuw & H. M. Blanken, "Control versus Data Flow in Parallel Database Machines," *IEEE Transactions on Parallel and Distributed Systems* (1993), (to appear).

[TeBl93b] W. B. Teeuw & H. M. Blanken, "Joining Distributed Complex Objects: Definition and Performance," *Data & Knowledge Engineering* **9** (1), January 1993.

[ZuRa68] F. W. Zurcher & B. Randell, "Iterative Multi-Level Modelling — A Methodology for Computer System Design," in *Proceedings International Federation of Information Processing Societies 1968*, 1968, pp. 138–142.