Client / Server Distribution in a Structure-Oriented Database Management System

Roland Baumann
Department of Computer Science III
Aachen University of Technology, Germany
roland@i3.informatik.rwth-aachen.de

Abstract

Client / Server distribution in (software) design environments can be achieved by distributing an underlying repository database management system. Several alternative approaches for such a distribution exist, which differ in the layer of the architecture at which client / server communication takes place. We discuss two approaches — query-server and page-server — we have studied in the context of the graph database management system GRAS which is used in the IPSEN project. We focus this discussion on the impact the two approaches have on DBMS architecture and functionality as well as on performance with respect to the communication overhead induced by them. The work which has been done to evaluate and implement these approaches in GRAS is presented and problems of both alternatives and the solution we have found are outlined.

1 Introduction

Software systems for application areas like CAD or software engineering handle complex data structures to model the design objects they deal with. To effectively support the work of a design team, these software systems have to store the design documents persistently [2, 10]. For this purpose they often use database management systems as repositories. In contrast to more traditional applications, e.g. of business administration where data has a rather flat structure and relational database management systems are very well suited, design applications model structurally rich data. Hence, it seems natural for such applications to use a DBMS with a data model that allows to handle highly structured data in a straightforward manner. Such DBMSs are called structure-oriented DBMSs. Object-oriented DBMSs [1] are an example of structure-oriented DBMSs.

In general, more than one developer works on a project. So the project database has to be available at the same time to different people working at different workstations. While this is uncritical as long as only one designer at a time accesses a document stored in the repository, it becomes a problem if this restrictive policy cannot be ensured. As an example, consider the case where project administration information is also maintained with the help of the underlying DBMS. Developers have to be able to access the administration information concurrently. So a design environment has to deal with concurrency and distribution. Choosing the underlying DBMS as the part of the environment to tackle these problems is a very natural and successful approach. It leads to a logically still centralized view of the repository and at the same time eases enforcement of data consistency and makes the design data highly available.

But even if distribution and concurrency control of a design environment should be handled by its repository DBMS, several alternatives remain how this can be realized. We investigate two of these alternatives in this paper, namely distribution with a query-server and with a page-server DBMS. We study them using the DBMS GRAS as an example system. GRAS was build to efficiently store design documents in form of directed, attributed graphs.

To be able to evaluate different client / server architectures, several aspects must be taken into account. First of all, the distribution scenario, i.e. the way the system is used by multiple users and tools, must be made clear. This highlights the requirements imposed on the system regarding concurrency control and transaction support. Next, the introduction of an inter-process communication layer also influences the complexity of the overall system and can even affect the functionality of other parts of the DBMS. Last but not least, communication costs of the distribution cut should be as low as possible, because inter-process communication is still a highly time consuming factor. Communication costs of a client / server distribution depend on the number and size of the messages sent between client and server. We have studied all three aspects of client / server distribution mentioned above for the query-server and the page-server implementation of GRAS and present our results in this paper.

1.1 Related Work

Research on client / server distribution and performance aspects in database management systems for engineering environments was already described by several authors from the software engineering as well as the database community. The work presented in this paper differs in several respects from previously published work. DeWitt et al. [9] study object-server, page-server, and file-server architectures for object-oriented database systems. They perform their measurements using a benchmark especially developed for this task. This benchmark runs on three prototype systems based on the WiSS [7] which were implemented as test beds for the measurements. In this paper, we study the communication profile of existing applications using the DBMS GRAS, which is the normal DBMS for these applications. In this way, we hope to measure the exact access profiles our tools impose on the database. Like [9], we focus only on the client / server architectures of the systems, while keeping the rest of the DBMS constant.

This is in contrast to the work of Chu [8], who considered four different commercial DBMSs to evaluate their use for electronic CAD applications, including relational and object-oriented DBMSs. The experiments were performed with the VLSI layout editing system Magic. To ease the integration of Magic with the different DBMSs, only load / store operations of layout cells were implemented and measured, so that access to data was coarse-grained from the DBMS point of view. Chu also discusses object-server and page-server architectures for object-oriented DBMSs and the impact of fine-grained locking in a page-server system. In contrast, the tools we use in our experiments access their persistent data in a fine-grained fashion. Also, we are not concerned with the locking granularity of our page-server, here. Rather, we focus on the impact query-server and page-server architectures have on functional aspects of GRAS and their communication costs.

From an application point of view, the work of Emmerich [11], is the one most closely related to ours. He also investigates fine-grained access to data in software engineering environments. To evaluate the performance of different DBMSs, including relational and object-oriented systems as well as an early version of GRAS without client / server architecture, he uses a sophisticated benchmark which closely resembles the access profile of tools in an integrated software engineering environments. The work is concerned with the overall performance of the systems, however, and does not specifically study the impact of client / server distribution.

1.2 Structure of the Paper

The rest of this paper is structured as follows. In section 2 we introduce the software engineering environment IPSEN and its underlying DBMS GRAS. This provides a basic understanding of our approach to fine-grained integration and shows how IPSEN tools access their data. We will also discuss the requirements on client / server distribution in an overall integrated IPSEN environment. Following this, in section 3 we discuss the query-server implementation of GRAS as a first solution to distribute the functionality of the system. We outline the advantages and disadvantages of this approach regarding the GRAS functionality and also present our experimental measurements of the (expected) communication profile of IPSEN tools for query- and page-server implementation. Then section 4 gives an overview of how the page-server distribution of GRAS is designed and implemented and again, what impact this has on the functionality of the system. We conclude the paper with a summary in section 5.

2 IPSEN and GRAS

The incremental, interactive, and integrated project support environment IPSEN [23] was developed in several research prototypes at the Department of Computer Science III at the Aachen University of Technology. The goal of the IPSEN research activities is to develop languages, methodologies, and tools for efficient and flexible software engineering environments. The project uses an a priori approach to data integration, so that all tools rely on a common data model for their documents. To allow for fine-grained relations between and within documents, all IPSEN documents are stored as graphs, most often as abstract syntax graphs. Abstract syntax graphs are based on abstract syntax trees, representing the context-free structure of a document. Nodes in this tree are called increments, and each increment represents a non-terminal symbol of the grammar defining the document structure. The syntax tree is augmented with context-sensitive and semantical edges between increments, so that we end up with a directed, node and edge labeled graph.

All IPSEN tools mark one increment of an abstract syntax graph as the current increment. They navigate through a document by moving a cursor from the current increment to another increment which in turn becomes the current increment. Changes are always performed on the current increment. A change can affect the increment itself or the whole subtree below it, modifying the structure of the graph.

To ease development of new tools, the IPSEN approach postulates the use of a framework for its tools. The tool functionality is specified with graph rewriting rules. This can be done with PROGRES (PROgramming with GRAph REwriting Systems), a language and environment to edit and execute graph grammar specifications [26, 30]. Code in Modula-2 / Modula-3 or C can be generated from these specifications and embedded into the IPSEN framework. To measure the impact on communication costs the two studied client / server approaches have, we used the PROGRES environment (which itself is built using the IPSEN framework) and prototypes built from generated code of PROGRES specifications as example applications.

All IPSEN tools use the structure-oriented DBMS GRAS as their repository. GRAS (for GRAph Storage) [18] was especially designed to meet the requirements of IPSEN for a repository which is able to store complex fine-grained data. According to IPSEN's view of documents as abstract syntax graphs, GRAS' data model is the directed, attributed graph with typed nodes and edges. Since the two client / server implementations we present in this paper use different versions of the core GRAS system, we will briefly review the history of GRAS next.

Development of GRAS started several years ago [4]. The system was designed as a program library enabling every application that was linked with this library to store its data as persistent graphs. The first step towards a client / server architecture was done in [29] by implementing an RPC interface on top of the library. Experiences with this first distribution identified significant problems regarding both, functionality and performance (cf. section 3). This lead to many improvements which were mainly contributed by [25]. The improved query-server system is currently used by most of the tools of the IPSEN project. We will refer to this version of GRAS as GRAS2 in the following.

At the time of the first attempt to distribute GRAS as a client / server system, a different project was launched aiming to reimplement GRAS completely in the object-oriented programming language Modula-3. The main concern of this project is to clean up the code and at the same time test new features without interfering too much with the IPSEN tools still in use. The page-server we describe in section 4 is realized in the new GRAS system, which we will refer to as GRAS3. This page-server variant of GRAS is already used by two IPSEN-like tools. The first is an analysis and development tool which stores software design documents like architecture and interaction diagrams as graphs [19, 21]. The second, currently being developed, aims at a fine-grained reader-writer environment for electronic books [22, 13].

The next sub-sections will give details on the requirements of a client / server architecture for GRAS. After outlining the multi-user scenario we have in mind in 2.1, we will present the basics of GRAS' functionality and architecture in 2.2 and 2.3, respectively.

2.1 Client / Server Scenario

As mentioned in the previous section, we need to clarify the client / server scenario for IPSEN tools before we can evaluate the two approaches to distribution in GRAS. We do this here by highlighting the differences between client / server systems for business administration on the one side and an integrated design environment like IPSEN on the other.

Like many other aspects of database system support for the two application domains, the client / server scenarios aimed at by them are very different, too. First of all, design teams work in a network of powerful workstations with no clear dominating server machine. This is in contrast to the still mainframe oriented style of client / server distribution for business administration systems. This makes it possible to burden the client machines with more tasks otherwise performed by the mainframe, since the sum of the computational power of all client machines surpasses the power of a single workstation server.

Furthermore, the access of client applications to the database in design environments is quite different from that in business administration, too. Usually, designers will work on their documents within their private workspaces for quite a long time before sharing a first version with other designers. And even then, this sharing rather means shared reading without interference of a writer, because further development of the document will result in a new version being created. In this way other developers are still able to work with the previous released version without direct interference of a writer. Shared access to documents will only be granted to a limited number of clients, say the members of a design team. This is in contrast to, e.g. banking applications, with potentially hundreds of clients each potentially accessing the same database concurrently for money transfers.

On the other hand, there are hot spots in design environments, too. Project administration is one example. In an overall integrated design environment, tools of each project member need to access the status information of the project to derive the agenda of the project member and to report about the status of ongoing work, so that the agendas of other team members can be updated (e.g. implementation of a module can only start when its interface is fixed).

To conclude, in the scenario we are aiming at, the common case is a single user with a private workspace reading data from a shared repository. When data is shared, there should be few conflicts due to concurrent writes. An exception is project administration, where we have a higher potential for data contention but also shorter access times for the typical operations on this data.

2.2 Functionality of GRAS

The programming interface of GRAS allows to create and delete nodes, create and delete edges, initialize and modify attributes of nodes in form of byte arrays of arbitrary length, and to maintain indexes on attributes. A node in a GRAS graph can be characterized as a pair *(node id, node type)*, an edge as triple *(source node id, edge type, sink node id)*, and an attribute as *(node id, attribute name, attribute value)*. GRAS supports almost all kinds of queries on these tuples, e.g. we could query for all outgoing edges of a node *(source node id, ?, ?)*.[1] Queries can have single values, sets, or relations as results, e.g. the previous query delivers a relation of pairs *(edge type, sink node id)*.

Beside these basic operations, GRAS offers a mechanism for automatic attribute evaluation via callbacks to a client program. Consider as an example a list of nodes linked with edges of type 'next' as shown in fig. 1. Each node has an attribute 'pos' giving the position of the node in the list. The value *pos* of this attribute for a node n can be computed as $pos(n) = pos(n. \leftarrow next-) + 1 \mid 1$, which should be read as 'the position of node n is either one higher than the position of its predecessor (incoming edge 'next') or one, if there is no predecessor of n and hence n is the head of

[1]Not supported are queries of the form *(?, edge type, ?)*, *(?, attribute name, ?)*, and *(?, ?, attribute value)*, as well as queries leaving all components unspecified.

the list'. This would be implemented by first declaring a static dependency relationship between the position attribute of a node in the list and the position attribute of the node connected to it with an incoming 'next' edge on type level. This static dependency tells GRAS when to mark dependent attributes as invalid. We could, e.g., insert a new node in the middle of the list of fig. 1 in which case all position attributes of the nodes to the right of the new node needed recomputation and would be marked invalid by GRAS. To

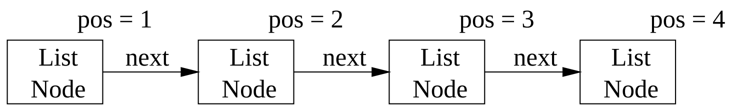

Figure 1: Linked list with position attributes.

enable GRAS to automatically re-evaluate invalid attributes, the application program then has to install a callback procedure which will be called every time GRAS tries to read an invalid position attribute. Note that this might result in a recursion in our case, because to compute a position attribute, the callback-procedure has to read the position attribute of the predecessor node, and this as well can be invalid, so that GRAS starts another callback.[2]

Callbacks are also employed by another feature of GRAS. A client can activate a demon that will notify it via callbacks when a specified event occurs. Such events can be all graph changing commands, but also undo-redo (cf. sub-section 2.3), and transaction-start / end. This gives GRAS some active behavior using an event-trigger-action mechanism like the one described in [20].

2.3 Undistributed Architecture of GRAS

Though the implementations of GRAS2 and GRAS3 differ in many respects, the coarse architecture of the core systems is the same for both. GRAS has a layered architecture as shown in fig. 2. The sub-systems have the following functionality:

PageStorage resides directly above the network-file-system. It provides a view on files as a sequence of memory pages. Pages are cached internally so that accesses to disk are reduced.

[2]Cycles in the evaluation process are detected and lead to an error message.

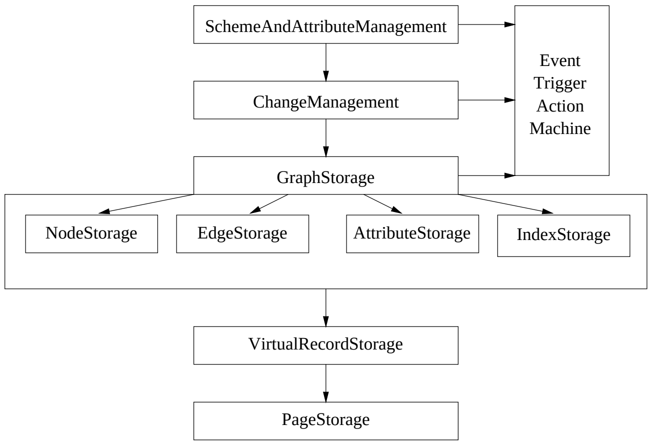

Figure 2: The architecture of GRAS without client / server distribution.

VirtualRecordStorage installs an efficient access structure on page-files. Pages store sequences of database records. GRAS uses dynamic hashing to compute the page a record is stored on and static hashing for the record location on a page.

EventTriggerActionMachine provides active behavior to the higher layers of GRAS and the GRAS clients. It uses callbacks to inform clients about occurring events. Events are detected by the three layers discussed next.

GraphStorage is the first layer of the GRAS architecture implementing an abstract data type for graphs. This layer already provides navigational access to a directed graph. Attributes are untyped, i.e. byte-arrays. It also detects events like creation / deletion of a node and notifies the event-trigger-action-machine about them.

ChangeManagement handles recovery situations of all kinds. It provides linear undo / redo, nested transactions, and crash recovery. It also supports versioning, but does not implement a fixed versioning concept.

SchemeAndAttributeManagement adds a type concept to the abstract graph data type. Nodes are instances of node types / classes and node types form an inheritance hierarchy. Node types define typed attributes for their instances. Attributes can be meta (class wide), intrinsic (instance), or derived (value depends on other attributes of the same or other nodes). Edge types determine the classes of source and target nodes of edges. Edge types also define source and target cardinalities to restrict the number of incoming or outgoing edges of the respective type at a node.

For the following discussion on distribution aspects, it is important to know that GRAS stores different parts of a graph in different storages. As can be seen in fig. 2 the sub-system GraphStorage contains the four sub-systems NodeStorage, EdgeStorage, AttributeStorage, and IndexStorage. These subsystems implement persistence for nodes, edges, long attributes, and indexed attributes, respectively. This design allows to keep the structural information of a graph (nodes and edges) together on relatively few pages, while unstructured data are stored separately. The goal of this separation is to speed up navigation, because in this way many more nodes and edges fit on one page than would if attributes, nodes and edges were stored together on the same storage pages. On the other hand, if attributes have to be read, the design is not a drawback, because the total amount of space for node and attribute information is almost the same, regardless whether they are separated or not, so that performance is mainly restricted by main-memory and cache size. To make sure navigation is efficient, GRAS tries to cluster neighboring nodes on the same page. Whether or not two nodes should be treated as neighbors can be determined by the application. This separation of structural information from unstructured contents distinguishes GRAS from most object-oriented DBMS, which usually store all information of an object in one continuous block of memory.

3 Distribution

The first thing to do when implementing a client / server architecture for a DBMS, is to select the architecture layer on which inter-process communication should take place. In [9], three alternatives are investigated: object-server, page-server, and file-server. These are all data-shipping approaches: they all ship data maintained by the server to the clients, so that these can work on it for some time autonomously. An additional alternative that we have studied is a query-server. This kind of server does not ship data but modifies it according to client-requests and answers client-queries. Query in our context means a navigational access to the graph databases, just like the programming interface of GRAS offers.

Though query-server architectures have been blamed to be unsuitable for navigational access to data ([9], [8], [11]), they do have their advantages. First of all, we have to note that the criticism mainly concerns query-servers with a high level query language such as SQL which does not support the required traversals well, so that many joins might be necessary to compute a result. This is different from our approach, because the queries offered by the GRAS server are the same as for the undistributed system. Moreover, we think an architecture like the one proposed for GRAS2 is a natural choice for distribution in heterogeneous environments. Because only the server needs to access the physical representation of data on storage pages, the clients may well operate on completely different operating systems and platforms. What is required though, is a machine independent representation of data sent as parameters and results to queries. This however is offered by most RPC system, e.g. the XDR encoding for the Sun RPC system [27]. The same arguments make the query-server the favorite candidate for use in a CORBA [14] environment.

The Triton object manager, which is used as repository in the Arcadia project [17] is another example for a query-server system. In [15], accessing the server from applications written in different programming languages serves as a main argument for the query-server architecture. This basically supports our argument for favoring query-servers in heterogeneous environments.

In the following sub-sections we will introduce the query-server implementation of GRAS (GRAS2) and discuss its advantages and disadvantages regarding the functionality and architecture of the system. We will then review the other alternatives for client / server distribution (object-server and file-server) and argue why we think they are infeasible for GRAS. Finally, subsection 3.3 presents the experiments we conducted to evaluate the communication costs query-server and page-server architectures impose on GRAS.

3.1 Query-Server Realization of GRAS

The architecture of the GRAS query-server implementation differs only in the top-level layer from its undistributed counterpart. As the new top-most layer, we added an RPC interface to the system, so that any calls from a database client to the server are realized by remote procedure calls. This architecture, which was also presented in [18], is shown in figure 3 in a compact form.

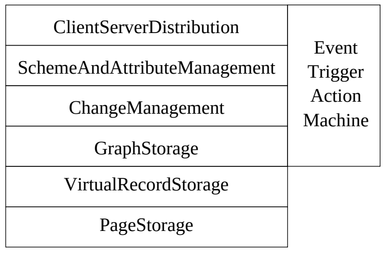

Figure 3: GRAS architecture with query-server distribution.

With this distribution style, all data is still accessed and changed in a centralized fashion, because all data is maintained by the server process exclusively. Clients can read and modify graphs by sending corresponding requests to the server process. Here, these requests are exactly the same interface calls as in the undistributed case. This centralization simplifies integrity maintenance and concurrency control to a great extent when many clients work on the same data, because only the server process performs data accesses.

On the other hand, when many clients concurrently access data, the server needs to answer many requests at once. This imposes a heavy load on the server process, because it has to perform all database functionality for all its clients. Figure 4 shows a simple scenario with only one server and two client applications accessing it. In general, GRAS2 support a multi-client / multi-server scenario, so that not one single server maintains all data. Still, server performance may severely limit the performance and scalability of the overall system with this kind of distribution.

A quite different problem arises due to the special interface of GRAS. As mentioned in section 2, GRAS uses callbacks to evaluate derived attributes and also for its event-trigger mechanism. These callbacks access code in the client application and hence have to be converted into remote procedure calls in this architecture. This has two consequences:

1. Client processes must in principle always be ready to receive callbacks from the server, even during time intervals when they are not involved in communication with the server.

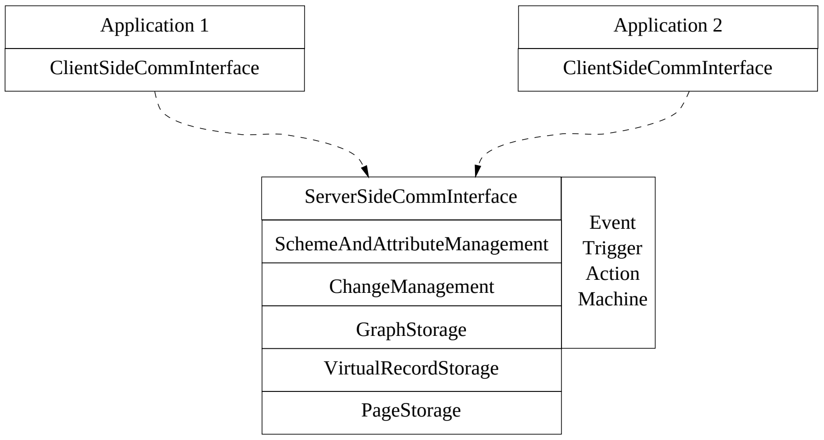

Figure 4: Simple scenario with one query-server and two clients.

2. When evaluating derived attributes, the common case is to use values of other attributes to compute the new value (see the example of the 'pos' attribute above). Hence, the client which computes the new derived value has to query the server for the values of other, defining, attributes. If these defining attributes are also derived, their value in turn might need recomputation, which leads to a callback to the client and so on. This means, queries from client to server and callbacks from server to client can be nested, as depicted in figure 5.

These two problems inhibit the use of one of the classical RPC mechanisms, which commonly do not support callbacks. Therefore [25] introduced a new communication infrastructure that is capable to manage such nested callbacks and also enables clients to respond to server requests when necessary.

Triton [15] has to deal with similar problems. In contrast to GRAS, Triton stores C++ objects as entities. And whereas the code for attribute evaluation in GRAS is executed by the client processes, the code for invocated methods in Triton is executed by the Triton server. While this supports the use of Triton in heterogeneous environments, it also has severe disadvantages. First of all, it further increases the load on the server machine, lowering the overall performance. Second, the code does not belong to the database management system but to the application using the DBMS. Hence, it might not be as reliable as the code for the server. In case of a programming error,

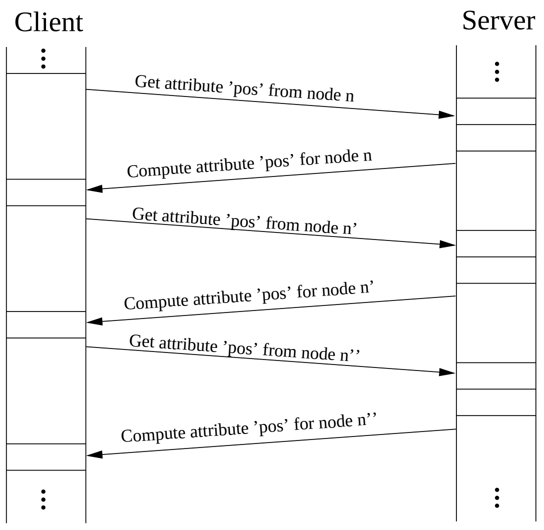
Client
Server
Get attribute 'pos' from node n
Compute attribute 'pos' for node n
Get attribute 'pos' from node n'
Compute attribute 'pos' for node n'
Get attribute 'pos' from node n''
Compute attribute 'pos' for node n''

Figure 5: Call stacks of a client and a server process with nested callbacks.

however, the server process might crash and in this way not only the client invoking the faulty method but all clients accessing the server would suffer damage, maybe even data loss. A solution to this problem would be to use an interpreted language. The code could then either be executed in a controlling environment at the server side or shipped to the client for execution.

The problem of asynchronous callbacks due to action execution of the event trigger machine, however, seems to be unavoidable. As long as active rules are only used to enforce or check consistency constraints, the solutions outlined in the previous paragraph were also applicable. But in integrated environments like IPSEN, tools use this mechanism also to synchronize themselves with other tools. Therefore, a communication mechanism that is able to deal with this kind of callbacks is necessary anyway, even, as we will see later, in the page-server system.

Besides the fact, that a query-server may well become a performance bottle neck due to the many client-requests it has to answer, the query-server solution also lacks performance because of the huge number of remote procedure calls between client and server (consider e.g. the number of RPCs necessary to perform a complete traversal of an abstract syntax tree). Since RPCs involve inter-process communication, they are several orders of magnitude slower than local procedure calls. Table 1 lists the times necessary for transferring data of varying size between processes with remote procedure calls.

Size in bytes	Execution time in ms
1	7.8
10	7.9
100	8.2
1000	9.7
4000	17.9
8000	28.4

Table 1: Communication costs for data chunks of varying sizes (taken from [9]). Times were gathered between two Sun 3 / 80 processors running the Sun RPC software and Version 4.03 of the SunOS.

To circumvent this problem, [25] offers an elegant solution: client and server can be linked together into one operating system process while still transparently using the same communication interface. 'Remote' procedure calls are then mapped to ordinary procedure calls with only minimal overhead. This of course is very efficient, as long as only one client accesses the server exclusively. The backside is, that performance for other clients accessing this server degrades heavily, because the server has to share its resources with the linked client. In addition, the arguments we have given against execution of application code by the database server process apply here as well. That means, if the application program, which is linked with the server, crashes, the server crashes too and may not be able to guarantee data consistency. Nevertheless, for many situations this is an acceptable risk. As an example, consider a developer working in a private workspace with an interpreter tool. The interpreter together with its user interface is directly linked with the server, while extra browsing tools are still able to connect to the same server to additionally monitor and trace execution if necessary.

Before we discuss alternatives to the query-server approach, we briefly summarize its advantages and disadvantages:

+ Complete reuse of existing implementation.

+ Easy concurrency control (completely within server).

+ Fast in single-user mode, when server and client run in one process.

– All operations of the GRAS interface have to be realized as remote procedure calls (more than 100 procedures).

– High communication load.

– Server is performance bottle neck.

– Callbacks to clients impose high requirements on communication mechanisms.

3.2 Alternative Client / Server Distributions

To avoid the disadvantages of a query-server implementation, one can use a different client / server architecture. According to [9], these are object-, page-, and file-server architectures for object-oriented database management systems. As the GRAS data model has many analogies to OODBMS data models, we think it is feasible to apply the results to GRAS as well. The measurements of [9] suggest that there is no clear performance advantage for any of the approaches. They conclude, however, that a page-server seems beneficial when a good clustering strategy together with a large page buffer for clients is used, whereas an object-server is preferable for clients with a small buffer or for poorly clustered databases.

From our point of view the page-server seems the only reasonable alternative to a query-server implementation, because it has a much simpler server design than an object-server (the server does not need to know nodes, edges, or graphs). This leads to an even smaller interface between client and server than would be the case for an object-server system. Also, it is unclear, what needs to be transferred when an object (i.e. a node in GRAS) is accessed by a client in an object-server realization of GRAS. In section 2, we argued that structural information (nodes and edges) is separately stored from content information (attributes). So when transferring a node from an object-server, we would either have to collect all its attributes and also transfer them to the client, or transfer attribute information separately when needed. The first solution collects unnecessary data in case of navigational access, while the second increases communication without the benefit of exploiting clustering for attribute accesses. Even worse, for pure navigational access, we would, like in the query-server approach, not be able to utilize clustering to reduce communication costs at all, because each node had to be transferred separately from server to client.

Hybrid and adaptable approaches like mini-page locking proposed in [8] or de-escalating locking and communication granularity as in [5] might of course be beneficial alternatives for GRAS, too. Nevertheless, we think they have one severe drawback and that is increasing complexity of the server process and the communication protocols. Additionally, the simulation experiments conducted for these approaches show the main benefits of the adaptable protocols for workloads with quite some degree of data contention due to concurrent writes. As argued in sub-section 2.1, we expect few conflicts in a client / server scenario for an integrated design environment.

The file-server approach simply uses a network file system to maintain the database files and access pages within these files. Its main drawback from our point of view is the necessity to obtain pages and locks with two separate messages, one to the operating system for the file access and one to a server process maintaining the locks.

3.3 Page- and Query-Server Communication

The question to be answered next is, whether a page-server implementation really reduces communication costs in comparison to a query-server. This surely depends on many factors, like caching and clustering strategies and can therefore not be answered before a page-server is operational. To gain at least a qualitative insight on this issue, [24] performed studies with GRAS2, the query-server version of GRAS, to estimate how often communication would take place in either client / server realization.

To accomplish this, the access profiles of four different sessions with the PROGRES environment and one session with a generated prototype were examined. These sessions are

Medical diagnosis A generated prototype defining a database to manage diagnostical data of a hospital. Within the session, a database was built using the graph grammar transformations specified with PROGRES.

Expression Tree A small PROGRES specification was parsed and interpreted within the PROGRES environment.

Binary Tree A small specification was interactively created within the PROGRES environment and interpreted afterwards.

Ferry Man A PROGRES specification to solve the well known problem of a ferry man, who has to transport a wolf, a goat, and a cabbage across a river without risking one item being eaten by another. This specification was handled within two sessions. In session 1, an erroneous textual specification was parsed and some of the errors corrected. In session 2, the remaining errors were removed.

The studied sessions range from fully automatic execution of graph grammar rules (Medical diagnosis, Expression Tree) to completely interactive editing sessions (Ferry Man, especially session 2). This should also give an impression on the wide range of different access profiles to data the tools in a design environment have.

The first quantity measured is the total number of calls to the GRAS application programming interface (API calls) in each session. Each API call results in an RPC in the query-server implementation. We compare these numbers to the number of pages read and written from and to stable storage, which can be used as a lower bound for the number of transferred pages between client and server in a page-server system. The result is shown in figure 6. We see, that the number of API calls is much higher than the number of disk accesses in all cases, with the two Ferry Man sessions marking both extremes: whereas session 1 has 1600 times more API calls than disk accesses, in session 2 this factor is only 33. If we leave these two extremes aside, we still end up with a factor between 100 and 1000 between the number disk accesses and the number of API calls in each session.

When we use the number of disk accesses in a session as a lower bound for the number of page transfers in a page-server system, we now should also find some upper bound for this quantity. We have used the average number of different pages accessed between two checkpoints for this. A checkpoint in GRAS is used to mark graph states which can later be reestablished using GRAS' undo and redo mechanism. Typically, IPSEN tools set checkpoints for every user interaction they perform. PROGRES and the generated prototypes also use checkpoints to implement non-deterministic backtracking when executing graph grammar specifications. Here, we use two consecutive checkpoints as boundaries for a top-level transactions.

To see why the number of accessed pages between two checkpoints is an upper bound for the number of page transfers between server and client in a page-server environment, one has to be aware that the GRAS2 server, being a query-server, keeps data pages in its page cache between checkpoints. It only flushes pages of log-files back to stable storage to be able to apply the log in case of a crash. So the number of pages read from disk in each

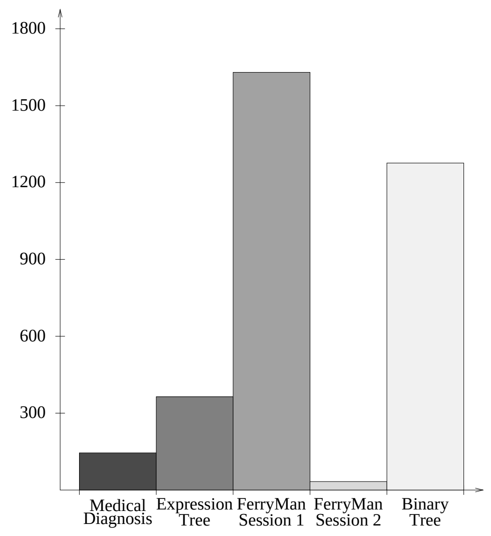

Figure 6: Ratios between number of API calls and disk accesses.

transaction may well be lower than the number of pages accessed. By using the total number of accessed pages for each transaction we even disregard the possibility of inter-transaction caching in a page-server system and assume that each page used in a transaction has to be transferred to the client process.

The ratios between the average number of API calls and the average number of accessed pages between two checkpoints are shown in figure 7. In all cases, the number of API calls is still more than an order of magnitude higher than the number of accessed pages. This means that we would expect a page-server to have at least 10 times fewer messages than a query-server. As table 1 shows, the time for transferring a page (2 k-Byte for GRAS2, 8 k-Byte for GRAS3) is only two to four times higher than the time for transferring a few bytes as parameters to an API call. Therefore, we conclude that the communication costs for a page-server implementation of GRAS will be lower than the costs for a query-server. Taken the ratios from our experiments, this might only be a factor of two in the worst case and up to a factor of 400 (Ferry Man session 1) in the best case. Both factors are not realistic, because we avoid the worst case by allowing inter-transaction caching in the page-server. But for inter-transaction caching of pages, we need a cache coherency protocol between client and server thereby increasing the number of messages

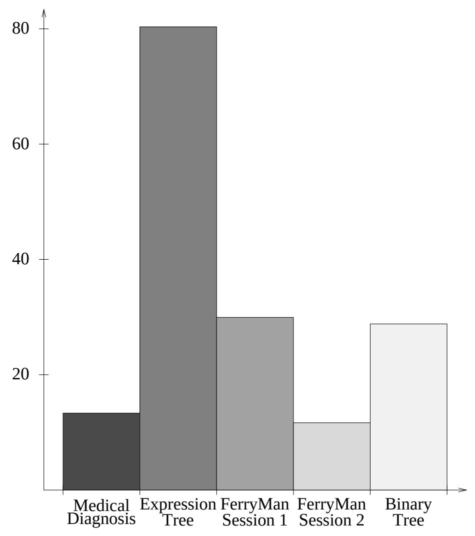

Figure 7: Ratios of average number of API calls to the average number of accessed pages between two checkpoints.

sent, so that we cannot achieve the best case, either.

4 The GRAS Page-Server

As mentioned in section 2, the page-server variant of GRAS is implemented in a complete new version of the system called GRAS3. The client / server cut for this system is placed within the PageStorage sub-system (cf. section 2.3). Analogous to figure 4 we can see a simple page-server scenario in figure 8. It illustrates how the complexity of database accesses now lies within the client processes. The page-server by itself is rather simple. It is only concerned with concurrency control and locking on page-level.

Since most of the database functionality is executed by the client process, callbacks between the DBMS and the application code are simple procedure calls in the page-server system. Compared with the query-server, where every callback was an RPC between client and server, this eases attribute evaluation to a great extend. The drawback, however, is that now every application accessing a graph needs to know how to evaluate an invalid attribute. This should be no problem for most applications, since code accessing an attribute

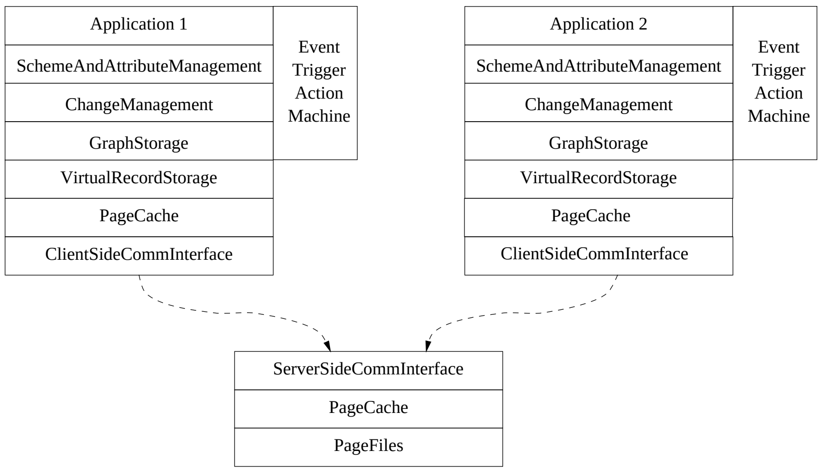

Figure 8: Simple scenario with one page-server and two clients.

normally will also be aware of its evaluation function. This is different for unspecific applications like graph browsers, which can only use schema information to access a database.

One solution for this problem would be to store the code for attribute evaluation functions together with the schema defining the attribute. This code could then be executed by all clients. Another solution is to request attribute evaluation from a different database client process. This could be issued by GRAS' event trigger machine or a similar mechanism like ToolTalk [16] which would even allow to start a client which is able to perform the task. Note that even though delegating attribute evaluation to a different client involves inter-process communication, the problems of nested callbacks should not occur here, since only the first request to evaluate an attribute is sent to the evaluating process, whereas all remaining evaluations are handled by that process internally, as depicted in figure 9. This is also the reason, why the communication overhead induced by this solution would not be higher than for the query-server system.

This leads to the second utilization of callbacks we have mentioned for the query-server: action execution by clients initiated by the event trigger machine. As can be seen in figure 8, every client process also has an EventTriggerActionMachine as part of its architecture. This means, also the callbacks for GRAS' trigger mechanism can be handled without inter-process

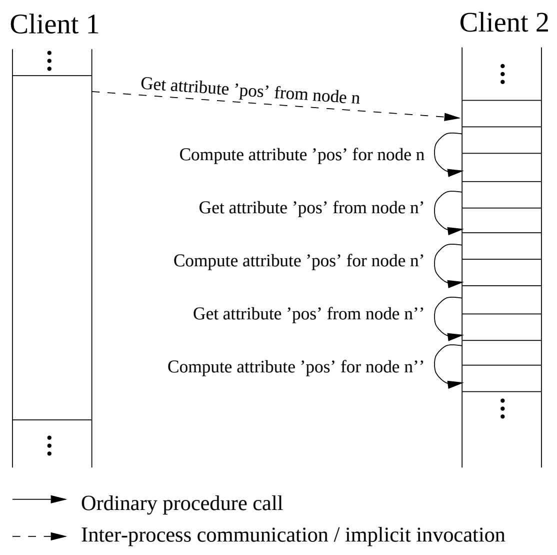
Client 1
Client 2
Get attribute 'pos' from node n
Compute attribute 'pos' for node n
Get attribute 'pos' from node n'
Compute attribute 'pos' for node n'
Get attribute 'pos' from node n''
Compute attribute 'pos' for node n''
Ordinary procedure call
Inter-process communication / implicit invocation

Figure 9: Call stacks of two client processes. Client 1 requests an attribute evaluation form Client 2.

communication, when only one client is involved. This should be the most common case, e.g. when triggers are used for consistency maintenance. However, IPSEN uses triggers also to synchronize clients (e.g. a graph browser is notified when it needs to update its display), and so we need a distributed event-trigger machine for GRAS3. In this sense, the server of figure 8 is too simple. The page-server of GRAS3 also has capabilities to promote triggers and trigger activations between clients as a separate sub-system not shown there. The communication overhead for the distributed event-trigger machine is still not higher than in the query-server case, though.

GRAS3 uses Modula-3's Network Objects [3] to realize inter-process communication. To deal with asynchronous callbacks, the system uses threads. This eases the implementation of e.g. the distributed event trigger machine. On the other hand, client processes now have to be aware of concurrent threads, too.

4.1 Cache Coherency protocol

In all data shipping client / server architectures, clients buffer data which is primarily maintained by the server. To achieve this, clients copy and transfer data from the server into their caches. They can then work on these copies and may even modify them locally. For the GRAS page-server, [24] developed a C^4-Protocol (concurrency control and cache coherency protocol) called CB-R/2Q. It is based on the CB-R protocol [12], but also integrates replica state and lock mode as is suggested by [28] to achieve some dynamism for propagation of updated pages.

The protocol is completely specified by a formal model enlightening the complex interactions between client cache management, nested transaction implementation, and client / server communication and also served as a guideline for the implementation. Table 2 shows in what lock modes / replica states a page can be, either viewed from a client or from the server. In figure 10, the state-transition diagram for a page from the clients point of view is depicted. As can be seen, instead of removing a page from the client cache after a callback, the protocol allows to keep the page with a P-lock. This indicates, that access to this page is not allowed, because it is used by a different client. Nevertheless, a page in lock-mode P can be dropped from the client-cache if necessary. Otherwise it will be updated when the client changing the page commits its transaction.

X	*exclusive*	Write lock, assures exclusive access for one client.
S	*shared*	Read lock, page is reserved for read access. This lock is only used by clients.
C	*cached*	Server: the page is buffered at client side but also available for read access to other clients. Client: the page is buffered but currently unused. It can be read, if the lock is locally upgraded into an S-lock.
P	*pending*	The page is cached, but locked by another client with an X-lock and therefore blocked (client only).
O	*out-of-date*	Virtual lock. Pages marked with this "lock" are out dated or not locked at all.

Table 2: Lock modes for pages.

The protocol implementation tries to reduce the communication overhead induced by the cache coherency protocol as far as possible. It does so

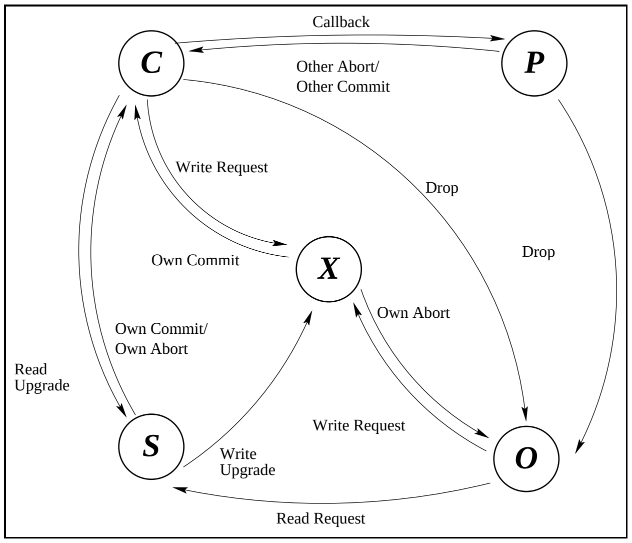

Figure 10: State transition diagram for lock modes of one page at the client.

by collecting update messages at transaction commit for several pages and piggybacking protocol messages on data messages whenever possible.

4.2 Performance of the Page-Server

To directly compare the performance of the two client / server architectures for GRAS (GRAS2 and GRAS3), we implemented the the OO1 benchmark [6] for both systems. The results indicate that the performance of the page-server system is better than that of the query-server system, given server and client are separate processes. A client linked together with the query-server, though, outperforms the page-server by a factor of 3 for read- and 10 for write-accesses. One reason of course is that no inter-process communication takes place with this "client / server" variant. Nevertheless, the new GRAS implementation still has much potential for optimizations on the client side, too.

5 Summary

We have presented our work on client / server distribution aspects in structure-oriented DBMS. We studied two different distribution alternatives using the graph DBMS GRAS as an example system. To clarify the context in which the system is used, we first gave a brief introduction to IPSEN and GRAS. Following that, we discussed two levels of client / server distribution for GRAS in detail, namely query- and page-server, outlining performance and protocol aspects. Both alternatives are implemented in two different versions of GRAS, which both are used for the tools developed at our department.

We investigated the communication costs of the two client / server approaches by measuring the access profile of existing applications for the query-server and estimating the number of page transfers in a corresponding page-server system. This analysis revealed a clear performance advantage for the page-server system.

The use of a page-server as underlying distribution mechanism made it necessary to redesign some of GRAS sub-systems to achieve the same functionality as with the query-server. One example for this is the event-trigger machine. The overall architecture, though, stayed the same. We also briefly reviewed a cache coherency protocol for the page-server which adds a dynamic propagation strategy to the CB-R protocol.

References

[1] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The Object-Oriented Database System Manifesto. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, page 395, Atlantic City, NJ, 23–25 May 1990. *SIGMOD Record* 19(2), June 1990.

[2] P. Bernstein. Database System Support for Software Engineering. In *Proc. of the 9th Int. Conf. on Software Engineering*, pages 166–178. IEEE Computer Society Press, 1987.

[3] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. SRC Research Report 115, Digital Systems Research Center, Palo Alto, February 1994.

[4] Thomas Brandes. Gras — design and implementation of a graph storage. Master's thesis, University of Dortmund, 1984.

[5] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):359–370, June 1994.

[6] R. G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[7] Honf-Tai Chou, David J. DeWitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the wisconsin storage system. *Software-Practice & Experience*, 15(10):943–962, oct 1985.

[8] Sheauyin Chu. Database support for electronic CAD applications: Performance and architecture. Technical Report UIUCDCS-R-94-1849, University of Illinois at Urbana-Champaign, December 1994.

[9] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A Study of three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proceedings of the Sixteenth Very Large Data Bases Conference*, pages 107–121, Brisbane, Australia, 1990.

[10] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments – The Goal Has not yet Been Attained. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the* 4^{th} *European Software Engineering Conference*, pages 145–162. Lecture Notes in Computer Science Nr. 717, Springer-Verlag, September 1993.

[11] Wolfgang Emmerich. *Tool Construction for Process-Centered Software Development Environments based on Object Databases.* PhD thesis, Dept. of Mathematics and Computer Science, University of Paderborn, Germany., 1995.

[12] Michael J. Franklin and Michael J. Carey. Client-server caching revisited. In *Proc. Int. Workshop on Distributed Object Management*, pages 252–274, Edmonton (Canada), August 1992.

[13] Felix Gatzemeier. Frameworks of Interactive Document Editing Environments with Often Changing Schemata. Master's thesis, RWTH Aachen, 1998. in german, to appear.

[14] Object Management Group. The common object request broker: Architecure and specification. Technical report, Object Management Group, 1991.

[15] Dennis Heimbinger. Experiences with an object manager for a process-centered environment. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[16] Astrid M. Julienne and Brian Holtz. *ToolTalk & Open Protocols - Inter-Application Communication*. Prentice Hall, 1994.

[17] R. Kadia. Issues encountered in building a flexible software development environment : Lessons from the arcadia project. In *Proceedings of ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 169–180, Dec 1992.

[18] Norbert Kiesel, Andreas Schürr, and Bernhard Westfechtel. Gras, a graph-oriented (software) engeneering database system. *Information Systems*, 20(1):21–51, 1995.

[19] Peter Klein. Designing software with modula-3. Technical Report 94-16, RWTH Aachen, 1994.

[20] A. M. Kotz, K. R. Dittrich, and J. A. Mülle. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. In M. Missikoff J.W. Schmidt, S. Ceri, editor, *Proceedings of the International Conference on Extending Database Technology (EDBT '88)*, volume 303 of *LNCS*, pages 76–91, Venice, Italy, March 1988. Springer.

[21] Katharina Mehner. Describing the Behavior of Software Systems. Master's thesis, RWTH Aachen, 1997.

[22] Oliver Meyer. Tools for a Reader-/Writer Environment with Often Changing Schemata. Master's thesis, RWTH Aachen, 1998. in german, to appear.

[23] M. Nagl, editor. *Building Tighthly Integrated Software Development Environments – The IPSEN Approach.* LNCS 1170. Springer-Verlag, 1996.

[24] Reiner Nix. Distributing a database management system: A workstation / server-architecture for the post-relational dbms gras. Master's thesis, RWTH Aachen, Department of Computer Science III, 1996.

[25] Wolfgang Reimesch. Design and implementation of a communication layer for database systems. Master's thesis, RWTH Aachen, Department of Computer Science III, 1995.

[26] Andreas Schürr. *Operational Specifications with Programmed Graph-Rewriting-Systems.* Dissertation, RWTH Aachen, Department of Computer Science III, 1991.

[27] Sun Microsystems, Inc. *Network Programming Guide*, 1990.

[28] Kevin Wilkinson and Marie-Anne Neimat. Maintaining consistency of client-cached data. In *Proceedings of the 16th VLDB Conference*, Brisbane, '90.

[29] Stefan Zohren. The gras-server – a client-server-realization of gras. Master's thesis, RWTH Aachen, Department of Computer Science III, 1992. in german.

[30] Albert Zündorf. *PROgrammed GRaph-rEwriting-Systems: Implementation and Use.* Dissertation, RWTH Aachen, Department of Computer Science III, 1995.