in: Proc. Int. Workshop on Research Issues on Data Engineering RIDE-IMS, Wien, 1993, pp. 152-160

A Remote Cooperation System Supporting Interoperability in Heterogeneous Environments

Michael Gesmann Andreas Grasnickel Harald Schöning
Sonderforschungsbereich 124
University Kaiserslautern
6750 Kaiserslautern, Germany
email: {gesmann, grasnick, schoenin}@informatik.uni-kl.de

Abstract

Federated database systems are heterogeneous with respect to data models and query languages but also with repect to underlying operating systems and hardware platforms. Besides transformations on the semantic level conversions on the data representation and communication protocol level have to be done. In this paper, we present the Remote Cooperation System RCS which provides client/server communication procedures independent of hardware and operating system aspects. Thus, the configuration of a system may change without affecting theparticipating databases and application tools. The RCS allows for asynchronous service invocation, thus enabling the simultaneous querying of several databases by one application. Furthermore, it supports multi-tasking of server processes and the element-wise transfer of results. Monitoring tools allow the assessment of several choices in system configuration and access algorithms embodied in the application processes.

1. Introduction

Database management systems (DBMSs) have become indispensable tools in many application areas. Databases have been established for many purposes, including public databases (PDBs) accessible from all over the world. Furthermore, within a single company there usually are several different databases storing data concerning various aspects of the company's activities. These databases may vary in aspects of the DBMS used, the hardware they run on, the modelling of data, etc. With the emerging need for an integrated view to all data of a company or data from several PDBs, there is a strong need for a uniform access to the data which hides the differences in the underlying DBMS, thus providing a "single system image" to the database user. However, it is not feasible to transfer all data of the various databases into one single DBMS. To overcome this problem, the concept of federated DBMS has been developed. A federated DBMS is a "collection of cooperating but autonomous component database systems" [7]. Autonomy means that every DBMS in the federation is independent of the others and may have its own algorithms, access protocols, etc. Hence,

Database Systems	
Differences in DBMS - data models (structures, constraints, query languages) - system level support (concurrency control, commit, recovery) Semantic Heterogeneity	
Operating Systems - file system - naming, file types, operations - transaction support - inter-process communication	Communication
Hardware / Systems - instruction set - data format & representation - configuration	

Figure 1.1 Aspects of heterogeneity in federated database systems [7]

cooperation can not be achieved by changing the DBMS but by providing a uniform access interface to them which transforms operations according to the specific DBMS's needs. The federation "… may involve a number of different types of heterogeneity - computer hardware, operating system, communication links and protocols, …" [10] which must be handled by the transformation (Figure 1.1). Examples for federated DBMSs handling hardware and operating system heterogeneity are Mermaid [9] and ADDS [1].

In general, one must assume that each of the participating DBMSs runs on a separate processor, with the application process (AP) providing a uniform data access interface running on yet another processor. In particular, the AP must be able to run on a variety of platforms in order not to restrict the access to the federated DBMS to a certain environment. The AP acts as a client of all DBMS of the federation. In many cases, access to more than one database will be necessary in order to evaluate one query issued at the AP level. In this case, the AP should be able to access the affected databases simultaneously in order to minimize response time by introducing parallelism. Hence, an asynchronous request to the underlying DBMS must be possible (in contrast, ADDS uses a synchronous RPC). The answers provided by the DBMS consists of a set of data records (e.g. tuples). Depending on the DBMS, this set will be delivered as a whole or element-wise. In the latter case, there is no need to wait for the DBMS's answer to be complete before transferring parts of it to the AP.

When sending queries to the DBMSs or receiving data from them, a conversion of the data representation might be necessary if AP and DBMSs run on different hardware platforms (for instance, the character codes or the representation of floating point numbers may differ). Furthermore, the AP has to cope with several communication protocols.

The requirements mentioned so far make it difficult to implement the AP. Furthermore, it is hard to evaluate the choices concerning parallel work and mode of answer transfer.

An obvious approach to the implementation of the AP is to separate the aspects originating in differences in the data models of the DBMS from communication aspects.

In this paper, we present the Remote Cooperation System (RCS) developed at the University of Kaiserslautern to cope with the latter domain. The RCS provides mechanisms for a client/server communication with the following features:

- Requests are issued asynchronously. Thus, several requests to the same or to different servers may run simultaneously.
- Results of requests may be transferred in several parts.
- The client program remains independent of several configuration aspects which are made transparent by the RCS:
  - the hardware platform and operating system the client runs on,
  - the hardware platform and operating system the servers run on.

  As a consequence, the data representation transformations are also handled by the RCS.
- Monitoring facilities showing the dynamic behavior of the federated client/server system are available.

With the help of the RCS, the AP may be implemented independent of the physical configuration of the federated systems. The AP and each of the DBMSs may migrate to another hardware platform or operating system without the need for changes in the AP's program. Furthermore, the RCS may also be employed for the implementation of the DBMSs themselves.

Several operating systems (OS) provide asynchronous RPC services which have a functionality similar to that of the RCS [2, 5, 6, 8, 11]. However, they are part of the OS and therefore do not enable communication among heterogeneous OS. In addition, the RCS provides two features not found in RPC proposals: the coupling of the communication with a multi-tasking facility and the capability of sending partial results which is needed to support pipelining parallelism within the execution if results are large and are computed part by part.

In the following section, we will detail the facilities for client/server communication offered by the RCS. In section 3, we will then discuss how to cope with heterogeneity and how to change the configuration of the federated system. Section 4 presents a brief overview of implementation aspects. The monitoring tools are described in section 5. The final section will give a short conclusion.

2. A Tool to Implement Client/Server Systems

The client/server concept is a generally used principle for decomposing large software systems into small and comprehensible units, called servers. Each server provides a distinct set of operations, the services, to be used by other system components. The execution of such a service is carried out by its respective server and might involve further requests to the same or to other servers. Beyond the pure input-output functionality, performance is a major concern in the development of contemporary applications and therefore the exploitation of parallelism is indispensable. To enable an implementation by concurrent programs, two different servers must not share any common data and consequently, all information passing between them is restricted to the parameters and results of service calls. The RCS is designed to facilitate the implementation of client/server architectures as parallel systems. In this section, a brief overview is given of the RCS's operations, their semantics and usage.

Generally, client/server cooperation is characterized by two-way communication: The client asks for some service to be executed by passing the name of the service and the appropriate parameters to the server. After completing the necessary computations, the server returns the result to the client. In the following, the term "task" is used to refer to the course of computation performed by the server due to a service invocation. To support the communication, the RCS provides the following operations:

- **Remote_Service_Invocation** (<servicename>, <servername>, <arglist>) is used by the client to invoke the service <servicename> with arguments <arglist> at the server <servername>.
- **Accept_Task** (<servicename>, <arglist>, <context>) is called by the server to receive the requested service code along with its arguments. After a call of this operation, a new task becomes active. The meaning of <context> will be explained later in this section.
- **Reply_Task** (<result>) is called by the server after computing the requested service in order to return the result to the client. Thereafter, the respective task is finished.

Note that there is no argument telling the server the name of its client. The RCS automatically keeps track from which clients a server's tasks have originated. When calling

Reply_Task, the RCS returns the result to the appropriate client, who requested the currently active task. In this respect, the RCS provides a task-oriented communication.

To take advantage of potential parallelism within an individual task, calls of Remote_Service_Invocation must not block the execution, since blocking service invocations prevents the client from requesting multiple services simultaneously and immediately resuming its own execution. Thereby, both parallelism among multiple servers and client/server parallelism is prevented with respect to the execution of an individual task. To overcome these limitations, RCS implements an asynchronous communication protocol by providing separate operations for invoking services and receiving the results:

- Any call to Remote_Service_Invocation returns a unique task number, which identifies the service invocation.
- **Get_Service_Result** (<taskno>, <result>) is then used to receive the result of the service identified by the task number.

In order to facilitate smooth client/server parallelism, an operation enabling the transmission of partial results is needed. Clients invoking time consuming services might prefer to receive the partial results as soon as possible. Using single result services for this purpose, client/server parallelism is interrupted after each single reply. In order to avoid these unnecessary synchronization points between client and server, the following operation is introduced:

- **Reply_Part_Of_Task** (<result>) is called by the server after computing a partial result in order to return it to the client. In contrast to Reply_Task, the respective task is not finished and remains active.

In the above discussion, server and services were treated as abstract concepts, which have to be mapped to concrete processing units of the operating system, the processes. In principle, there are two possible alternatives:

- Every individual task is executed within its own service process, created for this purpose. This is called single-tasking, multi-processing.
- All tasks of the same server are executed within the same server process. This is called multi-tasking, single-processing.

With single-tasking, multi-processing, every call to Remote_Service_Invocation triggers the creation of a new instance of the respective service process. After completing its computation and sending back the results, the service process terminates and leaves the system. The problem with this approach is the vast number of process switches incurred by every individual task. Obviously, there are at least two additional process switches produced by one service invocation: The first when the newly created process starts execution and the second after its termination. Moreover, during the execution of the service process situations may arise, where the execution has to wait until specific events occur, e.g. the termination of subtasks or the validation of global data. Handling this with individual service processes means to either block the process in busy-waiting mode and thereby wasting the respective CPU resources until the event occurs or to initiate a process switch and disable process execution until the reception of a signal (or some other method of inter-process communication), telling the emergence of the event. In case of long lasting waiting situations, the first alternative incurs an extreme waste of CPU resources and therefore the second alternative is the only reasonable approach in economical respect. Nevertheless, even the suspension of service processes incurs additional costs: Two additional process switches are necessary in a waiting situation.

Executing all tasks of the same server within the same server process, as with multi-tasking, single-processing, leaves the handling of waiting situations to just this process. In order to avoid a waste of processing resources, the server must be able to execute multiple tasks in a time-shared manner. On the occurrence of a waiting situation, the server process has to save the execution context (values of variables etc.) of the current task and the event(s), the task is waiting for, and to suspend the currently active task. Thereafter, a resumable task has to be chosen and executed. In comparison to multi-processing, multi-tasking yields two advantages: First, no process switches are necessary due to service invocations or waiting situations. Instead, the RCS provides operations to facilitate context switches between different tasks. Compared to process switches, we expect the overhead incurred by context switches to be less costly, since many unnecessary activities (in our case), e.g. saving the process registers, loading a new process image and initializing the run-time system of the programming language, can be avoided. Second, it is possible to apply server specific scheduling strategies rather than leaving the scheduling to the operating system. Therefore, application dependant preferences regarding the order of task execution may be reflected in the task scheduling.

For performance reasons, RCS implements the multi-tasking approach while relaxing the single-processing paradigm. With single-processing, of course, parallelism between the tasks of the same server is prevented. Therefore, the RCS allows for a static replication of server processes. The multi-tasking facility is to be controlled by the application using the following operations:

- **Break_Task** (<set_of_subtasks>, <context>) is used to inform the RCS that the current task is not to be resumed until one of the specified subtasks are completed. The context contains all the information about the current state of execution, which is necessary to properly resume the task's execution. Note that this is a non-blocking operation: After calling Break_Task, the server process is responsible for activating another task by calling

Accept_Task. Its context parameter comprises the state information of a previously suspended task. Obviously, the context data structure must contain detailed information about where to resume the execution.

- **Break_Task_Until_All_Terminated** (<set_of_subtasks>, <context>) serves the same purpose as Break_Task with the only difference that task execution will only be resumed, when all of the specified subtasks are finished.
- **Look_For_Service_Termination** (<set_of_subtasks>) and **Wait_For_Service_Termination** (<set_of_subtasks>) are both used to inquire the RCS which of the indicated subtasks are finished and return their respective task numbers. Wait_For_Service_Termination blocks the execution of the server process in busy-waiting mode while Look_For_Service_Termination produces a return value telling the application whether there are finished subtasks.

Within one server, dependencies may occur between arbitrary tasks due to shared global data structures. To enable synchronization between those tasks, RCS provides a simple event mechanism. The operation **Wait_For_Event** (<event_no>, <context>) suspends the current task until the specified event is raised. By calling **Signal_Event** (<event_no>), all suspended tasks waiting for this event are freed and might be resumed after a subsequent call to Accept_Task.

Having explained the RCS's task handling operations, their semantics are summarized in a state-transition diagram of the tasks (cf. Figure 2.1). Once a task is invoked, it is in one of the states *active*, *ready*, or *waiting*. When a client invokes a new task by calling Remote_Service_Invocation, a new task is created within the specified server and assigned to the *ready*-queue. As soon as the server process calls Accept_Task, one of the *ready* tasks is selected according to the scheduling strategy and becomes *active*. The selected task is then executed until one of the following happens:

- The execution completes and the server process calls Reply_Task, in which case the task leaves the system. Note, however, its result can still be accessed by the client calling Get_Service_Result.
- The execution cannot proceed until one or more subtasks have finished, in which case the server process calls one of the Break_Task operations and the task's state changes to *waiting*.
- The execution cannot proceed due to a dependency to a concurrent task. After calling Wait_For_Event, the task's state changes to *waiting*, too.

After suspension of the current task, the server process is responsible for calling Accept_Task and thereby activating another task from the *ready*-queue. The transition from the *waiting* to the *ready* state depends on the circumstances under which the task was suspended. It becomes *ready* again, as soon as either one or all of the specified subtasks have finished (when suspended by one of the Break_Task operations) or when the specified event is raised (when suspended by Wait_For_Event). A task leaves the system, when its state is *active* and Reply_Task is called. In contrast, calls of Reply_Part_Of_Task do not change the task's state.

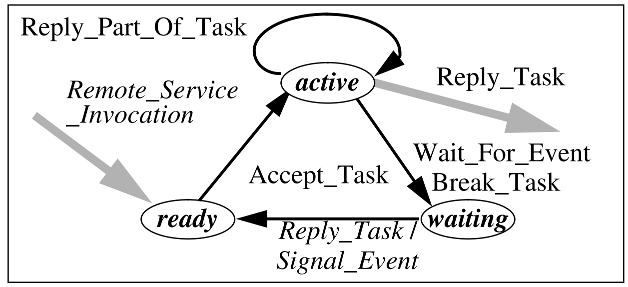

Figure 2.1: State-transition diagram for tasks

In conclusion of the RCS's application interface a short example is presented to demonstrate the use of the RCS's operations. Two server processes are introduced, named CLPROC and SEPROC, which cooperate using the RCS's primitives. Obviously, CLPROC acts as a server process for other processes not shown in the example. After initialization, CLPROC is looping through a program structure starting with the activation of a task (*1*), in the following referred to as PRIMETASK. Depending on the operation the task is supposed to perform, the control flow jumps to the appropriate service label (*2*). Thereafter, the code following the label "first_invoc" is executed (omitting how the

```
BEGIN CLPROC;
LOOP
  Accept_Task (…); (*1*)
  CASE service OF
  opi : CASE context.state OF (*2*)
    first_invoc:
      t1 := RSI (… opj);
      t2 := RSI (… opk);
      state := second_invoc;
      Break_Task_Until_
        All_Terminated (t1,t2, context)
    second_invoc: (*3*)
      Get_Service_Result (t1);
      Get_Service_Result (t2);
      Reply_Task (…);
    ENDCASE;
  ENDCASE;
ENDLOOP;
END CLPROC.

BEGIN SEPROC;
LOOP
  Accept_Task (...);
  CASE service OF
  opj :
    Reply_Task (…);
  opk :
    Reply_Task (…);
  ENDCASE;
ENDLOOP;
END SEPROC.
```

Figure 2.2: Example with two servers

context has been initialized). Eventually, two services op_j and op_k of SEPROC are invoked and some time later PRIMETASK cannot proceed without having the results of both subtasks. Before PRIMETASK can be suspended by calling Break_Task_Until_All_Terminated, the appropriate state information has to be set. After the suspension of PRIMETASK, the control-flow loops back to Accept_Task and goes on with a different task (if there is any).

Let us now look at SEPROC, which has just received the two subtasks of PRIMETASK. Eventually, one of them becomes the active task, gets performed and sends back the computed result. As soon as both tasks are *finished*, PRIMETASK changes from *waiting* to *ready* state and eventually to *active* state. This is the very moment, when the control flow in CLPROC leaves Accept_Task with the context from PRIMETASK. The control flow now jumps via the appropriate service to the code following the label "second_invoc" (*3*) (the label's name must have been saved in the context prior to the suspension of PRIMETASK). After receiving the results of both subtasks with Get_Service_Result the execution proceeds and finally finishes by calling Reply_Task. After that, another task might be activated and ever so on. PRIMETASK leaves the system, as soon as its client receives the result by calling Get_Service_Result. In Figure 2.2, the dotted arrows show the information flow between the RCS primitives of the two servers.

3. Configuration in Heterogeneous Environments

The RCS is designed to facilitate the implementation of distributed client/server architectures. A substantial requirement for distributed applications is configurability, i.e., the system may be executed on a single host as well as on multiple hosts of different machine types without any changes to the application's program code. Configurability of a client/server system means to enable the user to specify a configuration prior to system execution. In RCS-based systems the user may specify:

- Which servers are to be executed within the same process and
- how the processes are to be assigned to the hardware, i.e. to the processors.

To render the application's software independent of the present configuration, the RCS's operations have to hide all configuration aspects from the application program. In order to achieve this goal, the RCS is capable of transparently taking advantage of the appropriate communication medium. Servers within the same process pass their parameters by simply copying them within the address space of the common process. Servers within different processes running on the same host may use shared memory or resort to some other inter-process communication facility provided by the underlying operating system. Similarly, servers on different hosts have to use network facilities for communication. Generally, the parameters passed between client and server may comprise pointer-structured data, e.g. linked lists or trees etc. To copy this kind of data, appropriate procedures are needed to traverse these data structures and reconstruct them in another address space. To manage the transportation over a network, even linearization and delinearization routines and, in case of incompatible data representations between the involved machine types, appropriate conversion routines are needed.

In the RCS approach, the application programmer provides both the code for the server process and a procedural interface, which is linked with the client process. The interface contains procedure definitions for each service provided by the server and appropriate procedure definitions to receive the results from the server. The procedures' implementation is automatically generated by a tool. The code generated by that tool applies all the necessary transformations to the parameters with respect to the configuration at run time. Finally, all clients are linked with these automatically generated interfaces of all the services they want to cooperate with. Note that there is no reason to re-compile the system after changing the configuration. From the application programmers point of view, a server's interface is used as if the service were executed within the clients address space, no matter how the system actually is configured.

In the above discussion, we left out the question of how to replicate a server. As long as all replicas reside on the same host, the replication is invisible to the application programs and automatic load balancing is provided by the RCS. All relevant data structures for load balancing are maintained in the shared memory (cf. section 4). Conversely, replication of servers among different hosts is not provided by the RCS. However, the same effect might be achieved by configuring a server to multiple hosts, each time using a different server name. By doing so, the RCS treats them as different servers, which just happen to have the same functionality. Consequently, the application programmer is responsible for load balancing, in that he has to decide, which server is called for some service.

4. Implementation

The implementation of the RCS has to meet the concepts we have presented in the previous sections, i.e. multi-tasking as well as asynchronous activation and parallel execution of tasks. Furthermore, it has to hide the communication primitives to achieve location and architecture transparency. At the same time, it should select the most efficient communication mechanism if there are several choices. In addition, the RCS has to manage all the tasks a server has to process. In this section, we present the basic concepts of our implementation of the RCS.

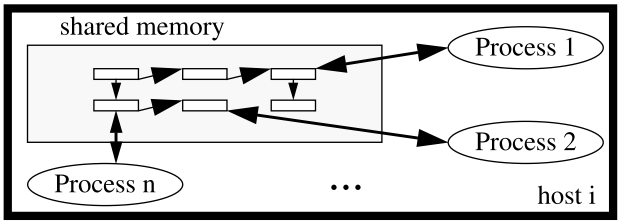

Figure 4.1: Communication via shared memory

Communication mechanisms

First of all, we want to look at the communication mechanisms. We support communication via shared memory in a closely coupled system as well as via messages in a loosely coupled system. We assume that clients and servers on one processor or in a closely coupled system of processors communicate via shared memory. This is the fastest communication mechanism, because we only have to manipulate some data structures. There is no need to create connections to send messages and to use sophisticated protocols to guarantee reliable communication between client and server.

As presented in section 2, communication between client and server mainly occurs in task activation and result transmission. Every process has central data structures in the shared memory which may be manipulated by the RCS of other processes. Figure 4.1 sketches this mechanism.

On the other hand, we have to implement communication in a distributed system via messages. There are two different ways to integrate message passing mechanisms:

- Message based mechanisms are included in each process. Then the RCS internally has to distinguish between the two communication modes and the code of the RCS becomes more sophisticated. Furthermore, the size of the processes increases.
- The communication across host boundaries is done by a separate process. Its main task is to accept service calls from local clients for remote servers, to send the parameters to the remote processor, to receive the answer, to accept tasks from and to send answers to remote clients. This implementation has several advantages over the first one. First of all, the client process can continue its work much earlier because it does not have to take care about reliable transmission of the parameters, decomposition of data into small packages, composition of result messages, etc. Moreover, we have a homogeneous view inside the RCS of a client. This means that there is no difference in the communication of a client with a local server and with a remote server. Just as in the case of local communication the client manipulates shared data structures, which are then read by the communication process. After receiving a result message, the communication process puts the result into these data structures.

We have implemented the second mechanism. Figure 4.2 illustrates this architecture. We assume that the additional process switches which only occur if there is communication across host boundaries are cheap compared to the time for transmitting the data.

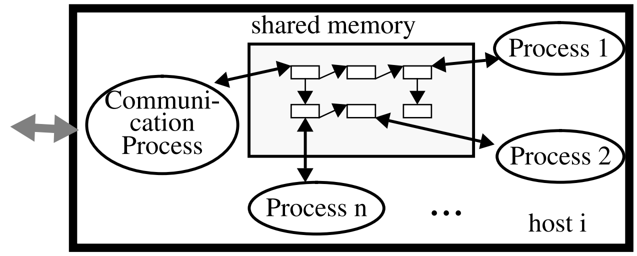

Figure 4.2: Communication via shared memory and a communication process

Note that the application of the RCS is not aware of the communication mechanism employed and that the data exchange across processor boundaries is done in a hardware independent format if necessary. Hence, from this point of view we are independent from a concrete configuration.

Task management

The second duty of the RCS is to manage the tasks a server has to process. In the following, we present the most important data structures for implementing the task management (Figure 4.3).

The root of our data structure is a server vector. Each server in the system has a unique number. Every entry in this vector represents one server which may be realized by multiple processes. These processes all have the same type, i.e., they correspond to the same program represented by process_type_control_blocks (ptcb). After initialization each entry refers to the ptcb of the communication process. If a process is running on a host it calls the function **Rc_Init** to initialize necessary data structures. This function allocates a ptcb if it does not yet exist. It assigns a pointer to the servers' entries in the vector, which refer to this ptcb. All tasks from clients on this host must be sent to the process type which is specified in a server's entry.

The ptcb contains information about the servers this process type implements, pointers to lists of task_control_blocks (tcb) which represent the tasks these servers have to process, a pointer to a list of received an-

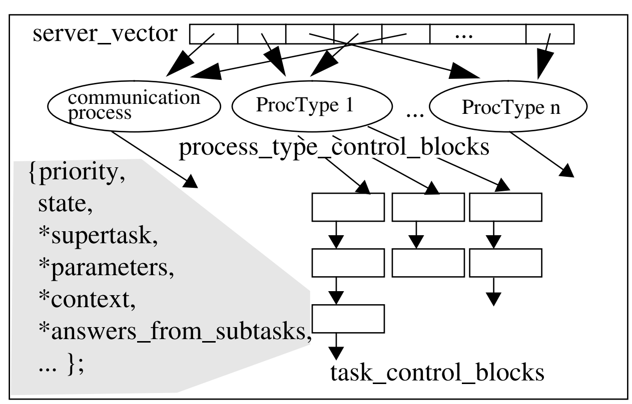

Figure 4.3: Data structures for task management

swers and some additional information for task management. For each state tasks may have, there is a separate list. It is easy to see that ptcbs represent types of processes but not the processes themselves. This simplifies task administration drastically as will be seen later. On the other hand, this means that a server on a local host cannot be included in more than one process type. Hence, we cannot combine servers arbitrarily to processes. But this restriction holds only for a single host, on different hosts we can integrate a server in different process types. Since it does not make sense to start a server more than once on the same processor (because this causes unnecessary process switches) this is not a restriction in a distributed system. In a system with shared memory we can combine the servers into one process. If this process is running more than once the code may be shared.

Each tcb represents one task. It contains information about the task's parameters, its state, its supertask, i.e., the task which has initiated it, and a list of received answers to its subtasks. Furthermore, it contains a pointer to the context which has been described in section 2. Storing the context in the tcb allows the migration of tasks between processes of the same process type running on the local host.

We illustrate the usage of these data structures by the execution of PRIMETASK (cf. section 2).

The server vector and the ptcbs are already initialized. By calling Remote_Service_Invocationa supertask initiates PRIMETASK. It allocates a new tcb in the shared memory, calculates the scheduling priority, and adds it to the process type's list of *ready* tasks, which is sorted by the scheduling priority. All tasks for all servers in a process type for all processes of this process type are treated equivalently. This leads to a very simple scheme for load distribution on a single host. The client will not assign the load to specific processes, but the available resources (running processes) determine the load distribution. They take a task for execution when they have finished or suspended another one.

After returning from Accept_Task the server executes the selected task (PRIMETASK). It analyses the parameters and initializes a context. Later on PRIMETASK suspends itself, because it has to wait for the results of its subtasks. Its state changes to *waiting*. The tcb will be taken from the list of *active* tasks and will be inserted into the list of *waiting* tasks. Other tasks can be processed while the answers of subtasks are being computed. For reasons of simplicity, the answers of the subtasks will be connected to a list of received answers in the ptcb for the moment. When performing the next Accept_Task the answers will be assigned to the corresponding client task. If a task becomes *ready*, its scheduling priority is calculated and it is inserted into the list of *ready* tasks. Afterwards, the first task of this list is selected as the new task for activation. When PRIMETASK is selected, the context contains all information for its continuation.Using the function Look_For_Service_Termination**,** PRIMETASK gets the number of a sub-task whose answer exists. For this purpose, the function has to look for answers in the list of answers in the ptcb as well as in the list of answers in the tcb of PRIMETASK. With Get_Service_Result the task gets the answer.

PRIMETASK terminates by calling Reply_Task which sends the answer to the client. It creates the result parameters in the shared memory, allocates an answer block and adds it to the list of answer blocks of the client task's ptcb. Afterwards it erases the tcb.

The implementation of the RCS consists of two parts. There is a special communication process which does all message handling with remote servers and with remote clients. It offers a reliable communication between different hosts. This process is transparent to the application. The second part is embodied by the set of procedures at the interface of the RCS which we have presented in section 2 (Accept_Task, Break_Task, etc.). They are easy to implement, because they do not have to distinguish whether a server is located on the same host or on a remote host.

The main problem is to synchronize the access of different processes to shared data structures by simultaneously running processes. For synchronization, we must not be too restrictive to enable as much parallelism as possible. In our case the access of the RCS interface functions to the data structures for the task management are critical. All these functions only need a very short time to manipulate the data structures. Therefore we use latches, which are busy waiting, if they cannot immediately get the requested resource. If an operating system does not support latches we use semaphores instead. Another problem arises if an operating system does not offer shared memory. Then the servers in different processes always have to communicate via messages. Reliable message based mechanisms have to be included in every process.

5. Monitoring

We need tools for monitoring and analyzing the systems performance in order to evaluate the selected scheduling strategies and priorities as well as the configuration. We want to investigate

- the dynamics when working on one specific task in the system,
- the load distribution, i.e. the number of tasks grouped by their state, and
- the load dependencies between servers, processes, processors and hosts.

For this purpose, we have developed a measuring tool which logs the calls of the RCS functions corresponding to changes of a task's state as illustrated in Figure 2.1. In observing the states of all tasks in the system and all calls of interface functions of the RCS we get a detailed insight into the system's dynamics and task execution.

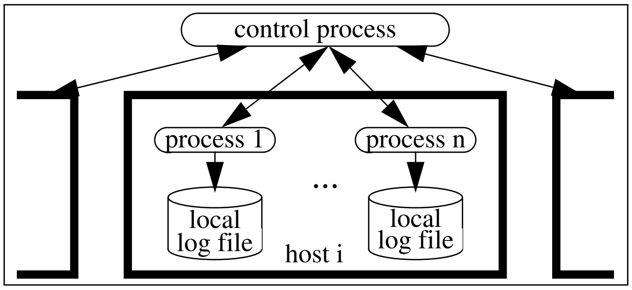

Figure 5.1: Architecture of the measurement tool

The architecture of our measuring tool is illustrated in Figure 5.1. During run time, every process produces a log file which contains the information mentioned above. For this purpose, we use an event driven approach. On every call of an interface function and on every change of a task's state, we write some log information. This log entry contains a time stamp, the event type and special information for single events. After terminating the measurement, these local log files are merged to a global log file which contains all events that have occurred. This global log file can be used for further detailed interpretation.

The main problem is to get a correct global log file, if the monitoring is done on a distributed system with independent clocks, which are not synchronized. The ordering of events from the different files cannot be done by only using the time stamps. Therefore, we extend the architecture by a new control process, which manages the start and the end of a measurement. It sends a message to all processes of the active configuration. After receiving this message each process takes a time stamp (bom) before staringthe measurement. To finish the measurement, the control process again sends a message to the participating processes which then again take a time stamp (eom) and finish their measurement.

Using these time stamps and the local log files, we define a virtual time for the global log file of our measurement. We suppose that the mean variation of time needed to send the start and stop messages from the control process to the processes of the configuration is very small compared to the duration of our measurement. Therefore, we can assume that the bom and the eom are recorded simulateously in each process. Hence, on all processors the time stamp with the smallest values at the beginning and at the end of the measurement on each processor happen at the same time and define the local start and end time of the measurement, respectively. Furthermore, we assume that the local clocks always run with a constant speed, especially there is no adjustment between the different clocks. This situation is illustrated in Figure 5.2. The real time is given, the two clocks of the two processors have different times, and the clock on a single processor runs with constant speed.

With these assumptions we can define a virtual time which reflects the real sequence of the recorded events when their local time stamp is converted to this virtual time. On all processors the local start time of a measurement (t_{ls}) is defined as 0. To compensate the different speeds of the clocks which is expressed by the differences of local end (t_{le}) and start time of the measurement on different hosts, we calculate the virtual time t_v of a local time t_l:

$$t_v = \frac{t_{re} - t_{rb}}{t_{le} - t_{lb}}(t_l - t_{lb})$$

Here t_{re} and t_{rb} are the real end and real start time of the measurement respectively. For simplicity we can use the begin and the end time of one processor as t_{rb} and t_{re}.

To verify whether our assumptions about the speed of the clocks and the time for transferring the begin and the end message are valid we investigate the merging of the local log files. There are some logical precedence relationships which must be true in the global log file. For example, a task can only be accepted by Accept_Task if it has been initiated by Remote_Service_Invocation and the answer can only be received by Get_Service_Result if it already has been sent by Reply_Task. If these relationships are never violated we assume that the errors caused by time variations during our measurement can be ignored.

The global log file can be used for further analysis. We implemented a visualization tool to illustrate the task load and the task relationships between servers. Each server is represented by a visualization object which consists of three fields indicating the total number of tasks in the represented server, the number of tasks which are actual ready for execution and the number of actual *active* tasks. The numbers are represented by different colors. Beginning from blue for zero tasks the color becomes more red the more tasks there are. Furthermore, the number of super-task/sub-task relationships between servers is represented by arcs which connect these servers. The thickness of the arcs expresses the actual number of these relationships.

To support the user of this tool we have implemented a few features to simplify the analysis. Servers in one process type may be grouped together in one output object. This combination may also be done for processes on one processor and for processors on one host. By using this feature the user can concentrate on the interesting level of abstraction. Furthermore, he can get curves which illustrate the changing of the load of a component over the time. The x-axis shows the passed time, the y-axis shows the number of *active*/*ready*/existing tasks for the server/process/processor/host. The speed of the output may be increased, decreased and it may be stopped by the user.

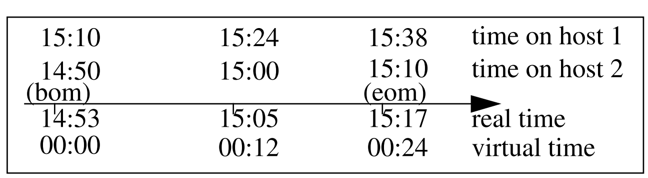

Figure 5.2: Different times on the hosts

Figure 5.3 gives an example of the output of this tool on a monochrome display. In the upper right corner there is an administration window to specify the input file and to control the speed of the output. The squares visualize servers on the processors “processor 2” and “processor 3”. One square visualizes all servers on “processor 1”. The lines between the squares show where client/server relationships may occur. The arrows show actually existing client/server relationships. The size of the circles in the middle of the squares visualize the number of existing tasks. The black portion in the circle shows the part of *ready* tasks.

6. Conclusions

In this paper, we have presented the Remote Cooperation System which provides a client/server communication mechanism which makes system configuration and heterogeneity transparent to the programs using it. The RCS on the client side offers functions to invoke a service asynchronously, to look or wait for its termination, and to get its result (or results). On the server side, tasks may be accepted, answered, or suspended. The RCS supports a multi-tasking implementation of servers. Multiple servers may be combined to one process. On the other hand, servers may be replicated. On a multiprocessor system, load balancing among replicas of a server is handled by the RCS.

The RCS chooses the cheapest communication mechanism available between client and server. This varies from copying within one address space over usage of shared memory among processes to network based message-passing with data representation transformations. The decision about the data transfer method is based on the RCS’s knowledge about the actual configuration and is transparent to the user (i.e., the client and server programs). An appropriate implementation guarantees that the application processes are not burdened with the communication overhead caused by data transfer over a network.

Monitoring facilities integrated into the RCS allow for an evaluation of configurations and strategies of service invocations. Since the RCS may also be used to implement the servers as cooperating groups of processes, the monitoring allows for an evaluation of query processing strategies in this context, too.

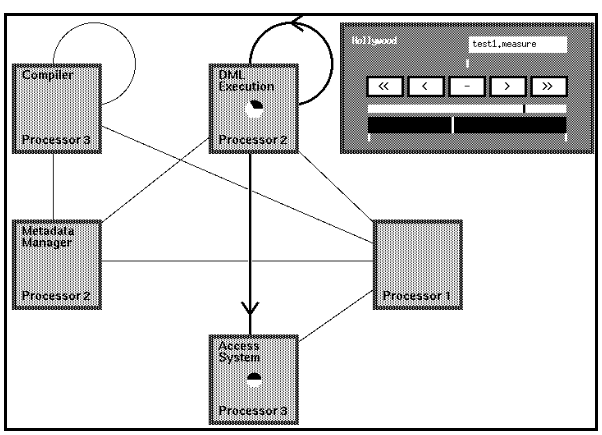

Figure 5.3: Exemplary output of the analysis

The RCS is an appropriate tool for the handling of communication in a system of federated database management systems, since it relieves the DBMSs and applications involved from the need to observe heterogeneity in hardware and data representation. An arbitrary DBMS may be included into the federation by just providing a server program which accepts the calls from an application and passes them to the DBMS, and sends the results of DBMS operations back to the application. If semantic transformations are done by the application this server program is trivial when using the RCS functionality.

The RCS has been implemented on various hardware and operating system platforms, including SIEMENS BS2000, SUN-OS (on SUN-3 and SUN-4 systems), AEGIS (on Apollo workstations), and DYNIX (on SEQUENT Symmetry), thus allowing client/server communication among these heterogeneous systems. Its functionality has been evaluated in the context of the PRIMA project at the University of Kaiserslautern [3, 4].

7. References

[1] Breitbart, Y. J., Tieman, L. R.: ADDS - A Heterogeneous Distributed Database System, in: Schreiber, F., Litwin, W. (Eds.): Distributed Data Sharing Systems, North Holland Publishing Co., 1985, pp. 7-24.

[2] Dineen, T.H. et al: The Network Computing Architecture and System: An Environment for Developing Distributed Applications, in: Proc. Usenix Conf., Phoenix, Az., 1987, pp. 385-398.

[3] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. 13th International Conference on Very Large Data Bases VLDB ’87, Brighton, United Kingdom, 1987, pp. 433-442.

[4] Hübel, C., et al: Using PRIMA-DBMS as a Testbed for Parallel Complex-Object Processing, in: Proc. Workshop on Research Issues on Data Engineering: Transactions and Query Processing, Tempe, Az., 1992, pp. 38-45.

[5] Jones, M.B., Rashid, R.F.: Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems, in: Proc. ACM OOPSLA Conf. 1986, pp. 67-77.

[6] Liskov, B. et al: Communication in the Mercury System, in: Proc. Hawaii Int. Conf.on System Sciences, 1988, pp. 178-187.

[7] Sheth, A.P., Larson, J. A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, in: ACM Computing Surveys, Vol. 22, No. 3, 1990, pp. 183-236.

[8] SUN Microsystems: Remote Procedure Call Specification, SUN Microsystems INC, 1985

[9] Templeton, M., et al: Mermaid - Experiences with Network Operation, in: Proc. 3rd Int. Conference on Data Engineering, Los Angeles, 1986, pp. 292-300.

[10]Thomas, G., et al: Heterogeneous Distributed Database Systems for Production Use, in: ACM Computing Surveys, Vol. 22, No. 3, 1990, pp. 237-260.

[11]Walker, E.F., Floyd, R., Neves, P.: Asynchronous Remote Operation Execution in Distributed Systems, in: Proc. 10th Int. Conf. on Distributed Computing, Paris, 1989, pp. 253-259.