Implementing Distributed Shared Memory Based on DCE

Bernd Dreier
University of Augsburg
Dept. of Mathematics
86135 Augsburg
Germany
Phone: +49 821 598–2116
dreier@Uni-Augsburg.DE

Theo Ungerer
University of Karlsruhe
Dept. of Computer Design and Fault Tolerance
76128 Karlsruhe
Germany
Phone: +49 721 608–6048
ungerer@informatik.Uni-Karlsruhe.DE

Abstract

The Distributed Computing Environment (DCE) software of the Open Software Foundation offers solutions for security problems and for shared file management in heterogeneous computer networks. It allows distributed programming by remote procedure calls and parallel programming by threads. Distributed shared memory in a computer network pretends a globally shared address space among networked computers.

By introducing distributed shared memory into DCE we raise the concept of threads to a higher level of concurrency — threads are spread over several machines. POSIX 1003.4a-compliant multithreaded programs are automatically transformed to execute on a computer network running DCE. Reprogramming is not necessary. The translator algorithms are concealed behind a precompiler, and a runtime system on top of DCE realizes the globally shared address space and distributes threads among different machines.

Keywords: Distributed Computing Environment, DCE, distributed shared memory, computer networks

1 Introduction

Today computer networks continue to grow both in size and importance. Local area nets, already linking personal computers and workstations, are merging into high-speed networks. Due to this fact, distributed operating systems are applied more often to networked computers. Some experts estimate that only 25 to 30 percent of the computing power available in a network is used [1, 2]. Utilizing this power will reduce users response time and in turn increase their work flow significantly.

Furthermore, in most cases distributed resources, such as under-utilized workstations, are already existent. Because no additional hardware is needed, the application of these resources is cheap. The main problems of distributed systems are upcoming security holes and missing standardization in heterogeneous networks. Some vendors such as Sun Microsystems supported distributed programming by Remote Procedure Calls and provided Network File Systems. But neither was the security problem solved satisfactorily, nor exists an integrated standard. The use of these partial solutions lead to a very complicated administration by a confusing configuration of networked computers.

The Distributed Computing Environment (DCE) software [2, 3, 4] of the Open Software Foundation provides an enormous opportunity to transform a group of networked computers into a single, coherent computing engine. By masking differences among various kinds of computers, DCE enables the utilization of distributed resources such as storage devices, CPUs and memory. DCE allows distributed and parallel programming as well as solutions for security problems and for shared file management. Because almost all major vendors support it, DCE seems to become a standard for distributed operating systems.

DCE supports parallel execution by POSIX 1003.4a-compliant threads, for distributed execution DCE remote procedure calls must be used. Unfortunately, the use of DCE RPCs requires reprogramming of software, already existing for parallel execution on a multithreaded system. Due to the lack of a common address space among different remote procedures, conversion of threads to RPCs leads to a complicated redesign of the program structure. The aim of our project is the automatic transformation of POSIX 1003.4a-compliant threads, as used in multi-

threaded operating systems like Solaris 2.3, OS/2 or Windows/NT, into distributed programs, that can be executed on different machines of a DCE network.

Chapter 2 introduces the parallel and distributed programming features of DCE, chapter 3 gives a motivation for implementing distributed shared memory on top of DCE. Chapter 5 demonstrates our approach on a example program, which is shown in chapter 4. The last three chapters give some experimental results, discuss possible optimizations, and draw the conclusions.

2 DCE

DCE uses three fundamental techniques to support distributed and parallel programming: The client-server model, the DCE Remote Procedure Calls (RPC), and DCE Threads.

Client and server are abstract terms. They can be considered as programs, for instance. Servers provide services that may be used by client programs.

DCE RPCs allow distributed computing by activating procedures on remote machines. In contrast to local procedures, RPCs do *not* share the same address space with the calling program. However, like local procedures, they execute synchronously: The calling program waits for the end of execution of the called procedure. Thus, RPCs support distributed, but not parallel computing.

With DCE RPCs a client is enabled to use services of a remote server: A client calls a procedure, which is performed on the server system. Besides the entire communication, the DCE RPC component accomplishes the necessary data marshalling. Therefore, the interface between client and server has to be defined exactly. DCE's Interface Definition Language (IDL) helps the programmer to accomplish this task. An example of an interface definition in IDL is provided in section 5.1.

Parallel computing is achieved by DCE Threads and their synchronization methods. DCE Threads are "lightweighted" processes. All threads of a client or a server belong to one "heavyweighted" process and access the same address space. Threads within a process share global data, open files, and any other resources available to the process. Threads from different processes, e.g. from client and server, do not share data. Threads may be executed in parallel on different processors of a single machine. However, threads of a single process cannot be distributed among different machines. To allow parallel and distributed computing in DCE, threads and RPCs are combined: The remote procedure is called from a previously created thread, all other threads of the process execute in parallel.

The necessary synchronization of parallel applications using DCE Threads is done by mutexes and condition variables. A mutex ensures the mutual exclusion of threads executing a critical section. The use of condition variables allows a thread to wait for a specific condition of shared data.

3 Why Distributed Shared Memory in DCE?

The distributed shared memory model [5, 6, 7] provides a common address space that is shared among all processors in a loosely coupled system — either a distributed memory multiprocessor or a computer network.

For loosely coupled systems, no physically shared memory is available. However, a software layer can provide a shared memory abstraction to the applications, using the services of the underlying (message passing) communication system. The shared memory model applied to loosely coupled systems is referred to as *distributed shared memory* [6]. An application can use the distributed shared memory just as it uses a normal local memory, except, of course, that the application's threads of execution can run on different processors or machines in parallel. Hence, software provided for time-shared uniprocessors or shared memory multiprocessors can also run on a computer network.

DCE Threads and their synchronization primitives are POSIX 1003.4a-compliant like threads in multithreaded operating systems such as Solaris, OS/2 or Windows/NT. By providing distributed shared memory in DCE, POSIX 1003.4a-compliant multithreaded applications, for instance originally written for multiprocessor workstations, can be spread over several machines of a network. In our approach, we provide a software layer on top of DCE, which allows multiple threads and their synchronization operations to execute in parallel in a computer network. This software layer, which consists of a precompiler and a runtime system, renders any reprogramming of the original software superfluous.

In contrast to our approach, distributing applications with DCE itself requires a strict client-server oriented design. In consequence, server programs must be devised, and threads rearranged using additional RPCs. This results in the necessity of extensive re-

programming of the original software.

Our implementation on top of DCE utilizes solutions provided by DCE, like security, file sharing, and — most important — the organization of communication in heterogeneous networks. Moreover, we choose DCE for its potential as a future standard in distributed operating systems.

4 Example Program

The multithreaded example program approximates π by using the rectangle rule to compute an approximation to the definite integral of $f(x) = \frac{4}{(1+x^2)}$ between 0 and 1.

```
#include "pi.h" /* includes standard headers and */
                /* contains prototype of eval    */

double total;
pthread_mutex_t total_mutex;
int number_workers, intervals;

void eval( int position )
{
  int first, current, last;
  double width, tmp, sum = 0;

  width = 1.0 / (double) (number_workers * intervals);
  first = position * intervals;
  last = first + intervals;

  for ( current = first; current < last; current++ )
  {
    tmp = (0.5 + (double) current) * width;
    sum += width * (4.0 / (1.0 + tmp * tmp));
  }
  pthread_mutex_lock( &total_mutex );
  total = total + sum;
  pthread_mutex_unlock( &total_mutex );
}

int main(void)
{
  int            i;
  pthread_t      worker_threads[MAX_NR_OF_THREADS];
  pthread_addr_t status;

  pthread_mutex_init( &total_mutex,
    pthread_mutexattr_default );

  /* reading of parameters left out */

  total = 0.0;

  for ( i=0; i<number_workers; i++ )
    pthread_create( &worker_threads[i],
      pthread_attr_default,
      (pthread_startroutine_t) eval,
      (pthread_addr_t) i );

  for ( i=0; i<number_workers; i++ )
    pthread_join( worker_threads[i], status );
}
```

5 Our Approach

In our approach we start with a POSIX 1003.4a-compliant multithreaded program. A precompiler translates the original multithreaded program into an IDL file and two seperate DCE-conform programs, called "master" and "slave" (see Fig. 1).

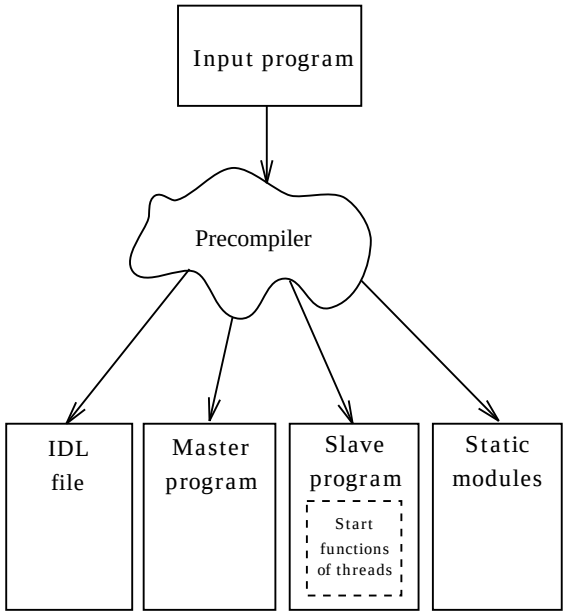

Figure 1: The precompiler

To allow dynamic distribution of servers we provide a facility called 'server-server', which consists of daemons running on each participating machine, and a runtime library accomplishing the communication with these daemons. The distribution of servers is done under terms of load balancing information. The 'server-server' and our load balancing facility are not described here. The shared memory of client and servers is pretended by a runtime library.

5.1 General Program Structure

This section outlines the actions necessary to transform the original multithreaded program into a distributed application. Figure 2 illustrates the transformation scheme. The main thread of the original program becomes a "master" program. The startroutine of a thread is transformed into a "slave" program, which is possibly distributed to different machines.

Creation of a thread in the original program results in two actions in the master program: A slave is started as a DCE server (possibly on a remote machine) and a RPC to this server is performed. Our master and slave programs virtually share the same address space as the threads of the original program.

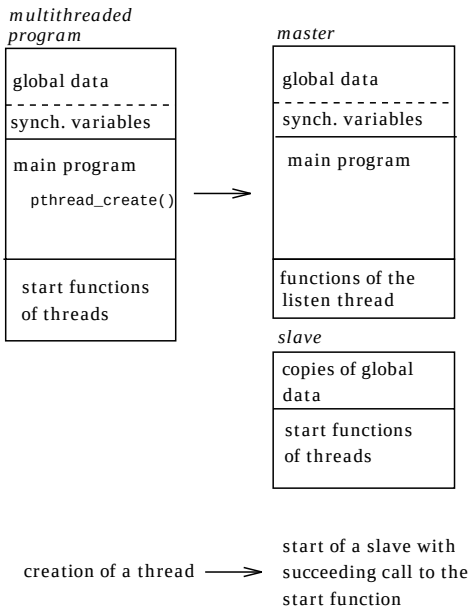

Figure 2: Transformation scheme

Using the function prototypes of the startroutines in the original program, an IDL file is generated by the precompiler automatically. C data types possibly implemented differently on various machines, are translated in DCE-compliant data types. Additionally, a parameter of the type `handle_t` identifies the slave to which a RPC of `eval(...)` is directed.

```
[uuid(006489e4-feb8-1d71-a31d-02608c2f76eb),
version(1.0)]

interface piapprox
{
void eval(
  [in] handle_t binding,
  [in] long position );
}
```

The master program is retrieved from the original program by omitting the startroutines of threads. Additional definitions for distributed shared memory are included and initialized by a function call `sm_init()`. Each `pthread_create(...)` call is replaced by a `rthread_create(...)` with two additional parameters, containing the name of the slave program and its DCE-binding. This call causes the startup of a slave program by the server-server on a remote machine. Subsequently, a local thread is created that performs a RPC to the startroutine in the slave. `sm_init()` and all functions beginning with `rthread_` are part of our runtime library for shared memory simulation.

```
#include "smsim.h"    /* definitions for            */
                      /* shared memory simulation   */
#include "piapprox.h" /* generated automatically by */
                      /* IDL compiler               */
#include "pi.h" /* modified header of        */
                /* the threaded program,     */
                /* prototype of eval removed */

/* definition of the global variables as idl_* types */
idl_long_float total;
pthread_mutex_t total_mutex;
idl_long_int number_workers, intervals;

int main(void)
{
  int            i;
  pthread_t      worker_threads[MAX_NR_OF_THREADS];
  pthread_addr_t status;

  pthread_mutex_init( &total_mutex,
    pthread_mutexattr_default );

  /* reading of parameters left out */

  sm_init(); /* initialize shared memory */
  total = 0.0;
  for ( i = 0; i < number_workers; i++ )
    rthread_create(
        &worker_threads[i], &binding[i],
        _slave_program,
        pthread_attr_default,
        (pthread_startroutine_t) eval,
        (pthread_addr_t)i )

  for ( i=0; i<number_workers; i++ )
    pthread_join( worker_threads[i], status );

  sm_cleanup(); /* cleanup of DCE information */
}
```

In the slave program the startroutine of the original program is embedded in a program frame, which makes the necessary settings for the DCE RPC component. The structure of the program frame is uniform over all possible slave programs. The DCE specific parts that form the program frame are not shown in this paper. The body of `eval(...)` is extended by the definition of local copies of the global variables.

```
#include "smsim.h"    /* definitions for            */
                      /* shared memory simulation   */
#include "piapprox.h" /* generated automatically by */
                      /* IDL compiler               */
#include <dce/idlbase.h> /* standard DCE includes   */
#include <dce/rpc.h>
#include "pi.h" /* modified header without */
                /* prototype of eval       */
```

```
void eval(
    /* additional parameter */
    /* [in] */ handle_t binding,
    /* [in] */ idl_long_int position )
{
  int first, current, last;
  double width, tmp, sum = 0;


  /* following variables have to be inserted */
  idl_long_float total;
  pthread_mutex_t total_mutex;
  idl_long_int number_workers, intervals;

  /* reading global variables for subsequent use */
  read_global( NUMBER_WORKERS, &number_workers,
    IDL_LONG_INT );
  read_global( INTERVALS, &intervals, IDL_LONG_INT );

  width = 1.0 / (double) (number_workers * intervals);
  first = position * intervals;
  last = first + intervals;

  for ( current = first; current < last; current++ )
  {
    tmp = (0.5 + (double) current) * width;
    sum += width * (4.0 / (1.0 + tmp * tmp));
  }
  /* first lock remote mutex, then read global data */
  rthread_mutex_lock( TOTAL_MUTEX );
  read_global( TOTAL, &total, IDL_LONG_FLOAT );
  total = total + sum;
  write_global( TOTAL, &total, IDL_LONG_FLOAT );
  rthread_mutex_unlock( TOTAL_MUTEX );
}
```

5.2 Synchronization

The synchronization operations on mutexes as used in the original program are available for synchronization of our master and slaves. In the slaves the synchronization operations are implemented by nested RPCs to the master. The master automatically creates a listen thread to accomplish global read accesses, global write accesses and synchronization events.

In the master of our example program the listen thread is started implicitly by `sm_init()`. This function establishes an interface to the slaves for synchronization events, read and write of global data, and initializes the runtime library of the server-server. Due to lack of space the code of `sm_init()` is not included in the sample program.

Synchronization operations in the slaves are transformed in RPC to the listen thread of the master. In our example program the `pthread_mutex_lock(...)` is translated in a `rthread_mutex_lock(...)`, thereby activating a function of the interface established by `sm_init()`. In consequence the listen thread of the master performs a local `pthread_mutex_lock(...)`. All other synchronization functions, including operations on condition variables, are handled the same way.

5.3 Implementing Distributed Shared Memory

Global data of the original program remain global data in the master program. All accesses on global data by the slaves are transformed in RPCs to the listen thread of the master. When translating a startroutine of a thread in the original program into a slave program, for each definition of a global variable, except for synchronization variables, a local variable definition is inserted in the slave program. Each appearance of a global variable as a `rvalue` is supplemented by a directly preceding `read_global(...)`, performed as a RPC, thereby copying the current value of the global variable in the master to the local variable in the slave. Likewise `write_global(...)` succeeds to each appearance of a global variable as a `lvalue`, thereby copying the current value of the local variable in the slave to the global variable of the master.

The variables `total`, `total_mutex`, `intervals`, and `number_workers` are global variables in the original program of our example. Except for the synchronization variable `total_mutex` we provide additional local variables in the slave program. Since `total` appears as a `rvalue` and as a `lvalue` in the statement `total = total + sum` in the original program, it is translated in the sequence

```
read_global( TOTAL, &total, IDL_LONG_FLOAT );
total+=sum;
write_global( TOTAL, &total, IDL_LONG_FLOAT );
```

in the slave program referring to the local variable `total`. The constant `TOTAL` is used to identify the global variable `total` in the master by the listen thread.

All global actions by the slaves, i.e. read, write, and synchronization operations, are implemented by RPCs — thus they execute synchronously. Therefore the order of read and write accesses of a single thread will remain unchanged. Moreover, the guaranteed sequential consistency [7] for synchronization operations on a single machine holds for all synchronization operations of our slaves. For global data accesses guarded by synchronization operations, the exclusive access specified in the original program is also guaranteed after our transformations. Therefore programs produced according to our method will have a similar behavior as the original program with respect to the consistency model.

6 Experimental Results

To assess the results of our project we have to compare the performance of our automatically created output program with the performance of a program, which is reprogrammed using RPCs and threads in a DCE-common programming style. The use of our transformation scheme renders reprogramming unnecessary. Due to communication overhead by additional RPCs we have to take a possibly worse performance of our output program against a newly designed program into consideration.

For the given example output program the use of four `IBM RS/6000` workstations achieves a speedup of 3.22 against the sample program of section 4 when evaluating a number of 100 000 000 intervals per worker. A reprogrammed, DCE-conform software made it up to a speedup of 3.38, equally using our server-server facility (see at the beginning of section 5) for dynamic start of servers. Without using the server-server the speedup could hardly be measured, because each server must be started manually.

The reason of the worse performance of our transformed against the redesigned program are four additional read/write RPCs and two additional lock/unlock RPCs of each worker. In the redesigned program four servers are started and RPCs to these servers are issued by previously created threads of the client. Therefore, global data are transfered as parameters of the RPCs, the synchronization operations are performed locally succeeding to the RPCs.

Of course, this good speedup could only be achieved, because the example is well suited for distribution. However, we expect worse performance, if the input program contains many global data accesses or many synchronizations, thereby increasing the communication overhead due to the induced read, write or synchronization RPCs. On the other hand, none of the many possibilities for optimization are used.

7 Optimizations

Out of the possible optimizations, we will discuss three examples briefly. For many applications, such as matrix multiplication, it is very inefficient to read only single data items, e.g. a single matrix element. Especially for read-only data, it is possible to transfer whole data structures instead of reading element by element. In the case of matrix multiplication the performance gain is easy to see: Instead of calling a remote procedure for each element of the input matrices, a single RPC for each matrix is performed.

The division of global data into several classes offers many possibilities of optimization: For shared read-only data only one read RPC in the slave program has to be executed. For global data exclusively accessed by a single slave, it is sufficient to perform only one write RPC before the end of the slaves execution. In the case of global data accessed by unsynchronized read and write accesses, a single read RPC preceding the first access and a single write RPC succeeding the last access is sufficient. This leads to a different, but not incorrect behavior of our transformed program, since a *correct* input program must not depend on a fixed order of accesses on unsynchronized data.

If exclusive read and write accesses are desired, these are always protected by synchronization operations. A further optimization is possible, if we use entry consistency [8], which is a more relaxed consistency model. Entry consistency guarantees correctness of data only after application of a synchronization primitive, which guards this piece of data. This allows us to restrict consistency to shared data which is protected by synchronization operations. Shared data is read only after a `pthread_mutex_lock(...)` on the mutex guarding this piece of shared data, and is written only before the corresponding `pthread_mutex_unlock(...)` occurs. In terms of DCE: Consistency of shared data is ensured only, if the mutex to which this piece of data is bound implicitly is locked previously. Hence, the transformed program of a correct multithreaded program will remain correct, if we read guarded data only *once* after a `rthread_mutex_lock(...)` on the guarding mutex and write it only *once* prior to the corresponding `rthread_mutex_unlock(...)`. I. e. not each occurence of the same variable as `rvalue` must be preceded by a read operation. Hence, network traffic can be reduced significantly. However, a procedure, which discovers the linkage of data and mutexes must be devised, because in POSIX 1003.4a-compliant programs this linkage exists only in programmers mind.

8 Conclusions

Our implementation uses only DCE-conform features to fulfill a precondition for the use of DCE services. This on-top-of-DCE implementation guarantees the compatibility to every DCE platform. Up to now the implementation of dynamic server distribution and the runtime library for distributed shared memory is accomplished. Moreover, we implemented several sample programs to compare the original POSIX 1003.4a-compliant multithreaded programs with the

programs generated by our intended precompiler. We are on the way to define and implement the detailed translator algorithms. Several ideas for optimization are devised, some of them are ready for implementation. Our current research concentrates on the optimization and the exact definition of translator algorithms.

References

[1] Open Software Foundation. *Guide to OSF/1: A technical synopsis.* O'Reilly & Associates, Inc., 632 Petaluma Avenue, Sebastopol, CA 95472, June 1991.

[2] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE.* O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol CA 95472, September 1992.

[3] John Shirley. *Guide to Writing DCE Applications.* O'Reilly & Associates, 103 Morris Street, Suite A, Sebastopol CA 95472, June 1992.

[4] Jr. Harold W. Lockhart. *OSF DCE Guide to Developing Distributed Applications.* McGraw-Hill, Inc., 1994.

[5] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[6] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54–64, May 1990.

[7] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, August 1991.

[8] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. *IEEE Compcon*, pages 528–537, 1993.