

High Level Interprocess Communication
Primitives for a Prolog
to C-Parallel Translator

Amaury de Cazanove

CENG Technical Report 93-37

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4484

September 1993

**HIGH LEVEL INTERPROCESS COMMUNICA-
TION PRIMITIVES FOR A PROLOG TO C-
PARALLEL TRANSLATOR**

by

Amaury de Cazanove

**Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2563**

ABSTRACT

Processing logic programs in parallel is much more complex than for conventional applications. Indeed, the inference mechanism, particularly the environment stacking and the runtime traversal of a search tree, cannot be easily implemented on distributed memory machines. The USC Research Team has therefore designed a parallel execution model for logic programming by applying the principle of data-driven execution to the inference mechanism of PROLOG. This model is highly adaptable to fundamentally different parallel architectural platforms, such as distributed memory multiprocessors, multi-threaded and data-flow architectures.

In order to efficiently target the environment to a range of parallel machines, a parallel compiler has been implemented. It compiles the operational semantics of PROLOG and produces an intermediate graphical form independent of any particular architecture. This design may then be translated by appropriate translators into the machine codes needed to run the application on specific architectures.

This paper introduces the parallel execution model with particular emphasis its low level part: the binding environment. Unlike other previously tested binding schemes often geared toward single address space machines, this one is highly optimized for execution on non-single address space architectures.

Another purpose of this research is the implementation of high level interprocess communication primitives, which will facilitate communication between processes within a single uni-processor machine, and also within several ones linked by network. In the future, these primitives will be imperative to the function of a translator designed specifically for Sparc stations.

ACKNOWLEDGMENTS

I first want to thank my advisors for this training period: Dr. Jean-Luc Gaudiot, associate professor, for welcoming me in his research group and Pr. Bernard Lecussan, Chairman of the ENSAE Computer Science Department for trusting me and offering me the opportunity to come at the University of Southern California.

I am also very grateful to HC Kim, Ph-D student, for his guidance, his kindness and his patience throughout all the five months of my training session.

I am also very thankful to all the members of all the EE-Systems Department at USC, more particularly Chinhyun Kim, Dae-Kyun Yoon, Moez Ayed, Namhoon Yoo, and Hung-Yu Tseng for their constant help and friendship.

Finally, I won't forget to thank Miss Catherine Montagna and Rohini Montenegro for their administrative help.

CONTENTS

1 INTRODUCTION	6
2 BACKGROUND	7
2-1 Prolog Terminology	7
2-2 Parallelism in Prolog	9
2-3 Parallel Perspective of Logic Programming Languages	13
2-4 Issues in Distributed Implementation	13
3 PARALLEL EXECUTION MODEL	15
3-1 Goals of the Design	15
3-2 Approaches	15
3-3 Features	16
4 BINDING ENVIRONMENT	17
4-1 Presentation and Objectives of our Model	17
4-2 Terminologies	17
4-3 Identification of Overhead in the Close Binding	17
4-4 The Principles of Functional Binding	20
4-5 Conclusion and Remarks	22
5 INTERPROCESS COMMUNICATION PRIMITIVES	23
5-1 Internal Structure of a Translator	23
5-2 Interprocess Communication	24
5-3 FIFOs	25
5-4 Message system calls	29
5-5 Sockets	31
5 CONCLUSION	36

CONTENTS

APPENDIX	37
A FIFO INTERPROCESS COMMUNICATION	38
A-1 FIFO_message.h	38
A-2 FIFO_primitives.c	39
A-3 FIFO_example1.c	42
A-4 FIFO_example2.c	44
B MESSAGEV INTERPROCESS COMMUNICATION	46
B-1 MESSV_message.h	46
B-2 MESSV_primitives.c	47
B-3 MESSV_example1.c	50
B-4 MESSV_example2.c	52
C SOCKET-BASED INTERPROCESS COMMUNICATION.....	54
C-1 UNIX_socket_primitives.c	54
C-2 UNIX_socket_example1.c	59
C-3 UNIX_socket_example2.c	61
C-4 UNIX_socket_example3.c	63
C-5 UNIX_socket_example4.c	65
C-6 INTERNET_socket_primitives.c	67
C-7 INTERNET_socket_example1.c	72
C-8 INTERNET_socket_example2.c	74
C-9 INTERNET_socket_example3.c	76
C-10 INTERNET_socket_example4.c	78
REFERENCES	80

1 INTRODUCTION

Logic programming based on universally quantified Horn clauses is becoming an accepted programming paradigm for symbolic computation. Thus, Prolog is one of the most popular logic programming language because of its many advantages in terms of ease of programming and declarative semantics.

Often, applications in symbolic computing are complex and demand enormous computation. The parallel processing is a promising answer to such requirements. Indeed, massively parallel architectures are becoming commercially available, and logic languages present intrinsic features for parallel execution, e.g., Or-parallelism, And-parallelism, Stream-And-parallelism and Unification-parallelism. However, the expected scale-up in performance has not materialized due to the inherent difficulty in programming these parallel applications. Indeed, much effort is still needed in developing a parallel software technology both at the low level (model of execution) and at the high level (programming languages).

The USC Research Group has therefore investigated a parallel execution model [1], characterized by its adaptability for implementation on various parallel architectures. In order to verify the performances of this model, a compiler for a pure logic kernel (i.e., a subset) of the PROLOG language, in conjunction with appropriated translators towards parallel machines is expected to be operational for the end of October 1993.

Finally, in the prospect of testing the functionality and the reliability of our Parallel Execution Model in an multi-processes environment, (and not in the goal of improving the performance over normal Prolog sequential systems), high level interprocess communication primitives have been implemented. They allow communication between processes located either on a unique machine or on separate ones linked by network. In the future, these primitives will be exploited by a translator dedicated to Sparc stations.

The organization of this report is as fallow:

- in section 2, we bring to mind the PROLOG terminology, the different kind of parallelism that have been identified for PROLOG, and review research undertaken elsewhere.
- in section 3, we briefly present the parallel execution model, called "Non Deterministic Data-Flow Parallel Execution Model" (NDFPEM).
- in section 4, we detail and assess a binding scheme which is highly optimized for non-single address space systems.
- section 5 contains the implementation issues of high level communication primitives.
- in section 7, we draw some concluding remarks, and outline possible future work that will be done for this project.

2 BACKGROUND

2-1 Prolog Terminology

This subsection provides a very brief introduction to Prolog, intended to familiarize the “non-Prolog” readers with the language terminology, syntax, program structure and execution semantics.

This is the foundation for understanding the different types of parallelism that exist and how a parallel execution model may support them.

A Prolog program comprises a set of **clauses** and a **query**. Program semantics can be either **declarative** as a set of formulae, or **operational** as functions. *Figure 2-1* shows a sample program and its declarative meaning.

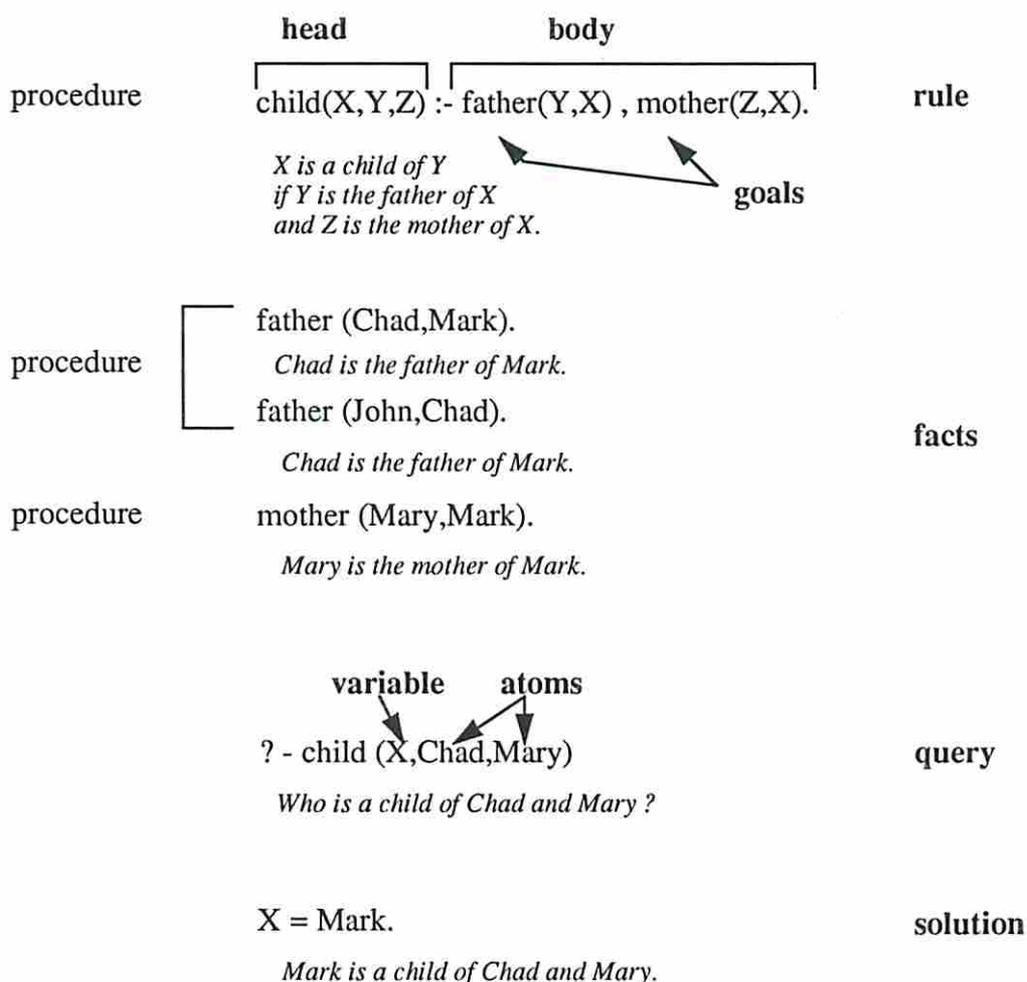


Figure 2-1: Components of a Prolog Program

A clause contains a **head** and an optional **body**, separated by the :- symbol. A clause body contains one or more **predicates**, which are referred to as **goals**. The collection of all clauses that have the same name and **arity** (the number of arguments) is called a **procedure**.

Prolog is a Logic Programming language, and as a programming language, it is distinguished from procedural or functional languages (C or Lisp for example) by its **logic** variables, **unification** operation, and built-in support for **backtracking**.

Prolog programs operate on terms, which can be either **simple** or **compound**. A simple term can be **atomic** or **variable**: an atomic term cannot change his value during execution, but a variable term may be bound to (i.e., assigned the value of) another term as a result of unification. A compound term is comprised of a **functor** and several terms: the functor is composed of a non-numeric atomic term and an **arity**, which specifies the number of terms contained in the compound term; these terms may be simple or compound.

Prolog variables are called logic variables; variables bound together form an equivalence class. **Binding** one variable to another term is the same as binding all variables in its equivalence class to the same term. **Dereference** (an operation common to all Logic Programming languages) refers to the operation that resolves a chain of references.

Unification is another operation common to all Logic Programming languages. It is pattern matching with variable substitutions. Unification operates on two terms and it can either succeed or fail. Unification succeeds under the following conditions: if both terms are atomic and identical (a simple pattern matching); if one term is a variable (unification binds the variable term to the other one); or if the functors and terms of two compound terms all unify successfully.

Backtracking is a technique that implements non-deterministic behavior. Prolog execution begins with a programmer provided query and attempts to prove the query using the facts and rules which are stored in the program. At the beginning, the query is added to a set of goals to be proven, S, as its first member. During program execution, a goal is removed from the set S. Clauses are tried to see if a clause head will unify with the goal. If the unification is successful, predicates in the clause body are added to S with variable bindings resulting from unification, and the clause to be tried next is marked as a target for backtracking. Another goal from S is selected and the execution proceeds. The query is proven true if all goals in S are proven. Backtracking is invoked when unification fails. It resets variable bindings made during the failed execution and removes goals added to S by the failed execution, such that the execution can proceed as if the clauses invoked by the failed execution had nether taken place. If the unification fails and no marked clauses can be found, the query fails.

2-2 Parallelism in Prolog

With its simple syntax and regular structure, a Prolog program is inherently an AND/OR tree. An AND-node corresponds to a predicate, an OR-node to a clause. Execution of the program is primarily a depth first, left to right traversal of the tree nodes. All the sibling AND-nodes are traversed depth first, left to right, whereas an OR-node is traversed only if all siblings to the left of it had failed. Backtracking allows for automatic exploration of previously untried alternatives. It is also the cause for a great deal of complications in efficient parallel implementation.

Figure 2-2 shows a Prolog program with its corresponding program tree. The arrows show the traversal of the nodes, which is equivalent to the execution of the program. The thin arrows show the forward execution, while the large arrows show backward execution.

The work done at each node consists of unifying the calling parameters with the head argument of the clause, and setting up the parameters for calls to its subgoals. In addition, the work in the body of the node may involve applying some functional primitives known as built-ins for arithmetic operations, input/output, data structure manipulations, and code alterations.

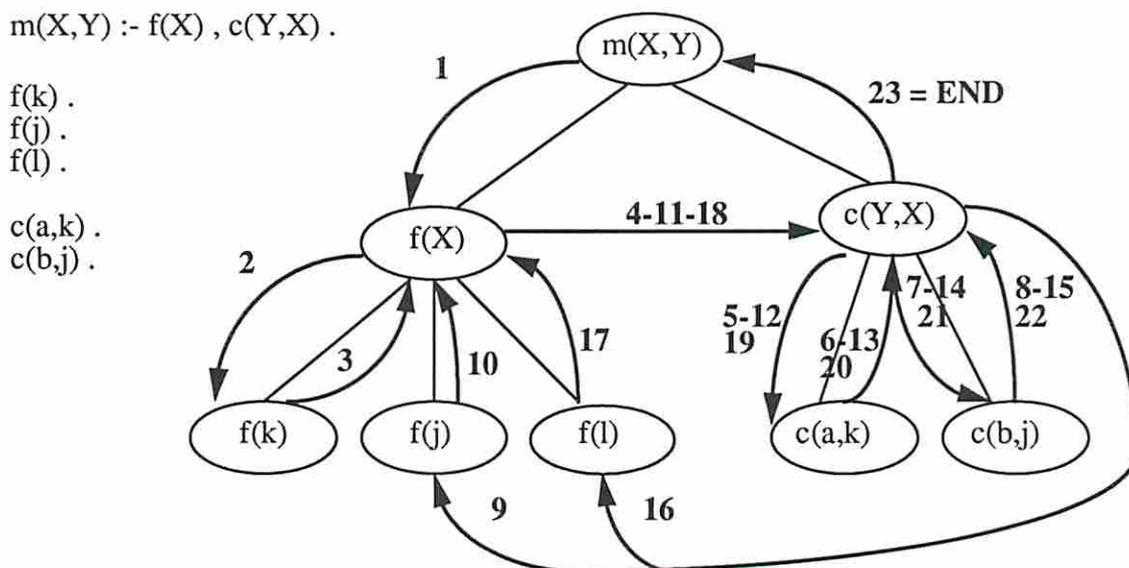


Figure 2-2: Prolog Program and Corresponding Program Tree

Here, we can see that the query has two solutions:

- at step 6, with X and Y binded respectively to k and a;
- at step 15, with X and Y binded respectively to j and b;

2-2-1 AND-Parallelism

Inspecting the program tree, it seems natural that the branches of the tree can be executed in parallel. When the partitioning is done at a clause node, where calls to subgoals are to be done in parallel, it is known as AND-parallelism. *Figure 2-3* shows the partitioning of the tree in *figure 2-2*, where the spawned process are separated from the root process with dashed lines.

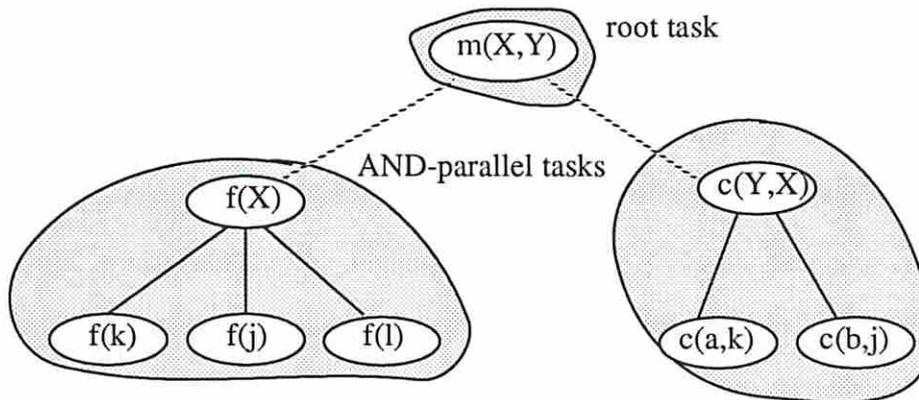


Figure 2-3: AND-parallel tree

The main difficulty with AND-parallelism is the problem of binding conflict, where more than one AND subtree executing in parallel attempt to bind the same variable to different values (e.g., variable X in the figure above). We have to making sure that substitutions are consistent across all literals.

Other problems include:

1. Keeping track of the success/failure of individuals literals.
2. Determining when the clause fails as a whole.

2-2-2 OR-parallelism

It comes from the observation that there are usually multiple clauses with the same predicate symbol in the clause head. When the program execution is partitioned at a procedure node, with broken branches to clause nodes which show alternative clauses that give several solutions, the parallelism exploited is known as OR-parallelism. The task which executes one of the OR-branches can continue with the next goal in the parent's clause. In *Figure 2-4*, the task completing the

first clause of $f(X)$ continues with the next goal $c(Y,X)$, with X now instantiated to the value k . Thus the results are passed down the execution tree and the final solutions are available at the leaf tasks.

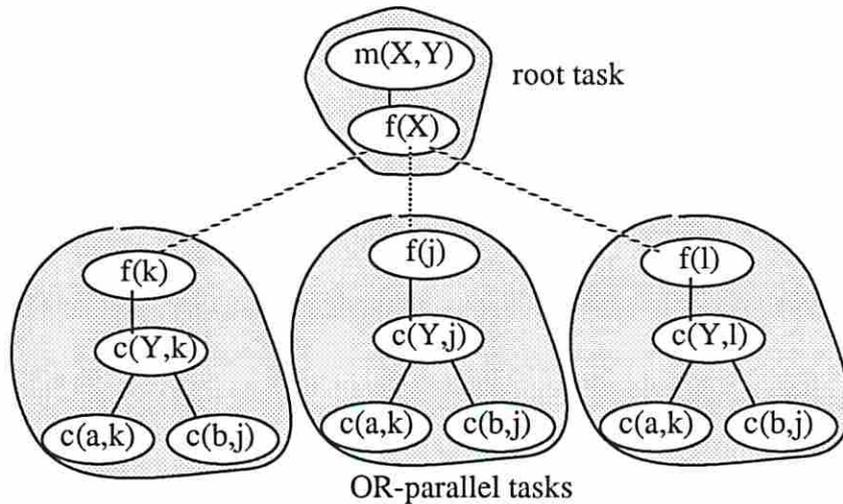


Figure 2-4: OR-Parallel Tree

When OR-parallelism is combined with AND-parallelism, the results of the OR-tasks may be passed back to the parent AND-task. In *Figure 2-5*, the goals $f(X)$ and $c(Y,X)$ are executed in AND-tasks. OR-tasks are then spawned to execute the clauses of f in parallel. The results of these OR-tasks are passed back to the parent task. The OR-tasks do not proceed with the next $c(Y,X)$ because it is already being executed by an AND-task.

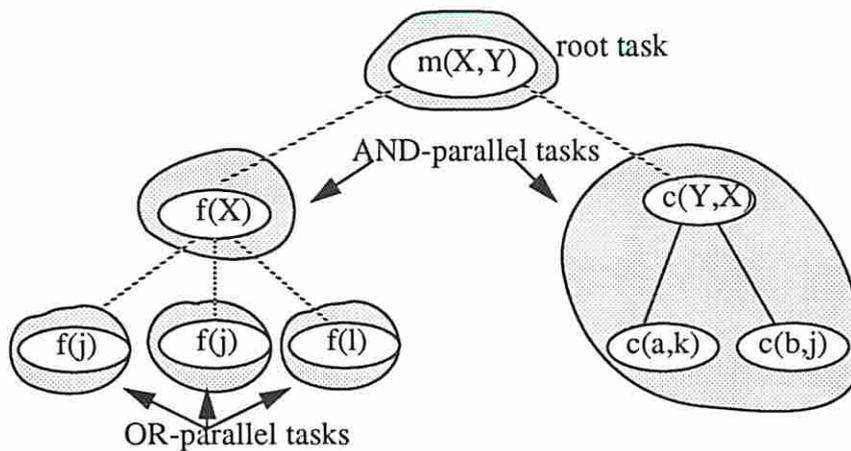


Figure 2-5: AND-OR Parallel Tree

OR-parallel clauses may share arguments variables in the head, but bindings of these arguments must be hidden from the ancestor nodes until that OR-node is actually traversed (in the sequential semantic order). The challenge in implementing OR-parallelism is to resolve the binding conflicts in a space and time efficient manner. For example, the OR-tasks of the goal $f(X)$ may attempt to bind X at the same time. Thus, each of these OR-subtrees must contain a separate binding environment, as we will see further in *section 4*.

2-2-3 Other Types of Parallelism

Other type of parallelism have been identified for Prolog. Considering the following example:

```
?- m(s(...), [---], X) .
m(s(...), [---], X) :- a(1,X), b(X) .      (m1)
a(1,X) :- ...                             (a1)
a(1,[3,5]) :- ...                          (a2)
a(2,X) :- ...                              (a3)
b([ ]) .                                    (b1)
b([H|T]) :- ... (H), b(T) .                (b2)
```

Stream-parallelism exists when a producer goal can pass a stream of values (elements of a list) to the consumer goal in a pipelined fashion. In the example above, $a(1,X)$ is the producer of X while $b(X)$ is the consumer; a and b can be executing in parallel, with b operating on a element in the list X while a is producing the next element.

Search-parallelism allows the heads of all clauses in a procedure to be unified with a given subgoal. This can be viewed as a simplification of OR-parallelism. In the example above, the search for the clauses that can match with $a(1,X)$ can be carried on in parallel, resulting in the list of two clauses $[(a1), (a2)]$.

Unification-parallelism carries out the unification of the arguments in the clauses head in parallel. In the example above, the structure $s(...)$ and the list $[---]$ in clauses head of $m1$ can be unified with their calling arguments in the query m concurrently.

Depth-parallelism carries out the unification of the head of a clause concurrently with the unification of a subgoal of the clause. In the example above, the unification of the arguments in m can be done concurrently with the unification of arguments in a .

2-3 Parallel Perspectives of Logic Programming Languages

The parallel implementation of logic languages entails an added level of complexity, even though parallelism in logic programs is, to some extent, already specified implicitly in their operational semantic. Indeed, the strategy for parallel scheduling must address not only the issue of delivering parallelism, but also that of minimizing the search space. For instance, given two AND-independent goals a and b which can be reached in respectively m and n inferences, the total amount of inferences is $O(mn)$ if the two goals are searched sequentially, whereas only $O(m+n)$ is performed if the goals are executed in parallel. On the other hand, for a given search space of a Prolog program, the search strategy employed in a parallel model will determine the overall processing time.

Moreover, the sequential nature of logic languages renders difficult their parallel execution: the selection rule, e.g., the left-most try among alternatives in Prolog, forces sequentially. In turn, this requires a stricter control and adds some constraints during Or-parallelism execution. Further, the semantic of logic languages entails the maintenance and traversal of search tree at run-time. The shared property of the tree favors centralized scheduling strategies, thus renders inefficient their implementation on multiprocessor systems with a non-single address space.

2-4 Issues in Distributed Implementation

A number of parallel models have been proposed. They differ from each other in the way they handle the various forms of parallelism and in their runtime scheduling strategies. As a testimony of the proximity of the logic language semantics to the single address space, it should be noted that the majority of these models are geared toward the shared memory multiprocessor model. In contrast, only a few are geared to distributed memory systems.

The data-flow model has been proposed as an efficient answer to the programmability issue in massively parallel machines. It provides for dynamic scheduling upon operand availability and has the potential to hide communication latency, while maximally exposing available parallelism. The clear benefit of this computation model is the improvement of the programmability.

In general, locality of reference indicates that normally the working set of Prolog involves 2 or 3 levels in the tree. Thus, the fine granularity supported by the process models may require high communications if they allow unrestricted access. Binding schemes in distributed models are directed to insure access only to environments in the local address space. To this end, the *variable importation* method [2] employs backward unification along with head (forward) unification, using import vectors. The *closed binding* approach [3], a successor of the variable importation

method, efficiently uses memory by using closing operations without import vectors. In [3], Conery's closed binding is optimized to avoid copying of grounded structures. Apparently, the above binding schemes trade the local access costs in additional computation from the auxiliary operations, i.e., back unification or environment closing. This extra computation may be intolerable when programs entail high rates of instance terms, e.g., lists and structures, since explicit duplication of the terms is required at every such auxiliary operation.

In order to materialize the expected scale-up in performance on various parallel architectures, we designed a parallel execution model, called the "Non-deterministic Data-Flow Parallel Execution Model" (NDFPEM) [1]. This model, described in *section 3*, is compound of a number of components at various layers that are highly optimized particularly for non-single address space architectures. At the highest layer, it provides a platform for distributed scheduling on top of self-organized execution principles. At its lowest, the model provides a novel binding scheme, presented in *section 4*, particularly geared to distributed environments.

3 PARALLEL EXECUTION MODEL

3-1 Goals of the Design

An important requirement of this parallel execution model for PROLOG is the preservation of compatibility with the sequential semantics of PROLOG, excluding addition or modification of syntactic structures of standard PROLOG. Another requirement for this model is the ability to achieve scalable performance during execution of applications on large scale parallel architectures. Finally, the third requirement is a high adaptability to profoundly different architectural platforms.

Due to the above considerations, the primary goals of the design are to:

- investigate a design to relax the tight relationship of PROLOG semantics with single address spaced architectures,
- provide a robust ground for distributed scheduling,
- investigate a binding scheme that supports the distributed nature of our target machines and avoids the causes of overhead in the “closed binding” [3].

3-2 Approaches

In order to achieve the previously stated goals, we rely upon the data-driven approach. Considering that for a given environment, the evaluation of a clause produces a new environment, we interpret a clause as a function. The environment at the point of the clause invocation, and the environments produced from the evaluation, are viewed as the input and output of the function, respectively. Along with this interpretation, we apply the principles of data-driven computation to the inference mechanism of PROLOG, particularly for the execution control and the unification process. Finally, we designed a graphical intermediate representation for logic languages, the Abstract Logic Programming Graph (ALPG), which will be used as a vehicle for code translation on the target architectures. The model thus established is called Non-deterministic-Data-Flow-Parallel-Execution-Model (NDFPEM).

Consequently, the data-driven approach leads to a relaxation of the tight relationship between PROLOG semantics and single address-spaced architectures by eliminating the shared search tree. Moreover, the self-organizing inference paradigm embedded in our NDF graph provides a robust ground for distributed scheduling.

3-3 Features

The NDFPEM includes the following distinct features. First, it is designed to exploit various forms of parallelism: OR-, AND-, and Stream-AND- parallelism. Moreover, fine grain parallelism internal to OR-tasks is supported by the ability to schedule multiple tasks on a processor. Second, the NDFPEM is highly optimized for non-single address space architectures, specifically distributed memory multiprocessors and architectures with hardware support for fine-grain parallelism, such as multi-threaded and data-flow machines. The model is also suitable for distributed- or multi-processing of logic languages in general workstation environments. Finally, the data-driven execution of the NDFPEM attempts to be efficient, doing away with overhead incurred from the process management required in general AND/OR-process models.

4 BINDING ENVIRONMENT

4-1 Presentation and Objectives of our Model

As we said previously in *section 2-4*, for a thread of tasks on a **processing element** (PE), the distributed models should maintain the efficiency of shared models. Focussed on the sources of the inefficiency, we observed that the closing operation in closed binding [3] may be a major source of overhead. In order to relieve the overhead from distributed models, we identified the exact details of the overhead and investigated the solution. The resulting binding scheme is called **functional binding** (FB) [10]. As a hybrid of non-closed and closed binding, the FB is directed to capture the efficiency of the non-closed binding, while insuring the restricted access of the closed binding. Thus, the FB facilitates the distributed implementation as one viable solution to massively parallel computation of logic programming.

4-2 Terminologies

In this subsection, we repeat the definitions of terminologies used in general closed binding schemes. An **environment** (E) is a set of variables that are reachable in terms of binding and reference. A **closed environment** is a set of frames E such that no pointers or links originating from E refers to slots in frames that are not members of E.

The **closing** operation means a transformation of a closed environment of two frames, working frame (WF) and reference frame (RF). Before the transformation, there may exist references from one frame to the other, so that neither is a closed environment. When we close WF with respect to RF, WF becomes a closed environment and all inter-frame references will originate from the RF. *Figures 4-1* shows a example of closing operation.

4-3 Identification of Overhead in the Closed Binding

In general, the idea of closed environment is valuable for distributed models. Though, in real implementation, it embeds source of overhead.

First, closing operations requires scanning an environment to check references reaching outside the environment. This includes scanning all structures reachable from the environment.

Second, when a unbound structure is updated for closing, it is required to make a copy of the

Program: :- p(x₁,f(y₁)).
 p(x₂,y₂) :- q(x₂,y₂).

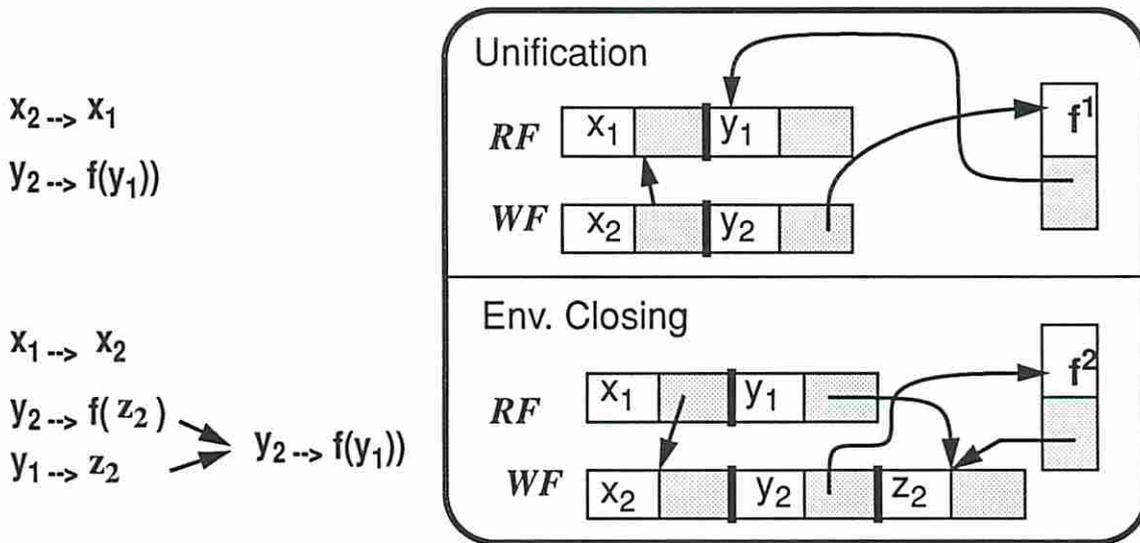
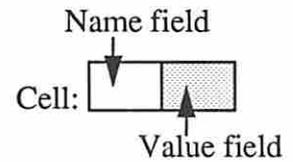
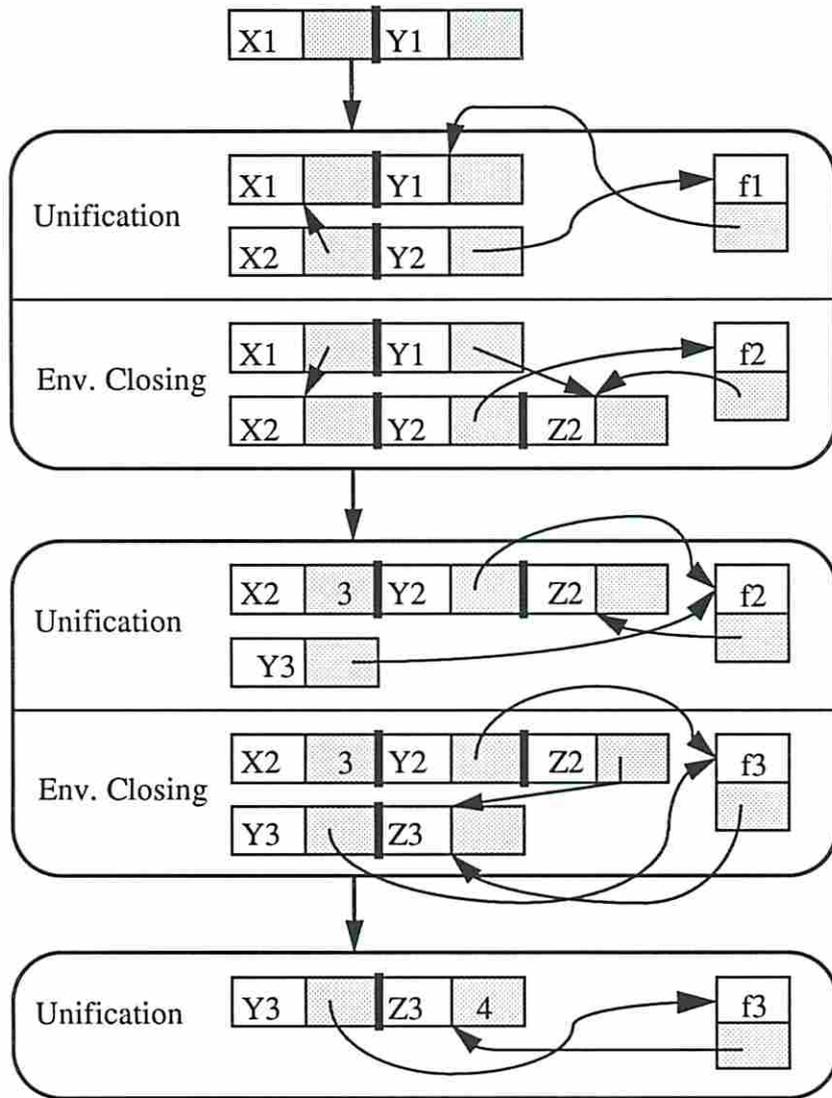


Figure 4-1: An example of closing operation.

structure. Closed binding provides an opportunity of structure sharing in that variables are always placed in frame and slots in heap memory and have always pointers to the variables in frames. In reality, the indirection capability can hardly be exploited for structure sharing, since the offset value in the heap is not fixed during closing. Thus, in real application, the closed binding always assumes structure copying for every instantiating of a variable inside a structure.

Finally, when a slot in a working frame has a reference binding to a variable in a reference frame, closing operation entails an extension of the WF to import the variable of the RF.

In the *figure 4-2*, we use a shortened notation in which a reference is expressed simply with an arrow. The figure shows a set of unifications for a sequence of OR-tasks. Unification and closing operations are enclosed in a circle and only environment frames are specified since the arguments are straightforward in the program. It also illustrates the three sources of overhead. More particularly, during closing operations, the reference of a slot in structure *f* is directed to point a slot in different frames and the offset value doesn't remain the same. Namely, the offset values for the three references are 2, 3 and 2 respectively. This leads to the updating of the slot with appropriate values, resulting in 3 copies of the structure *f*.



Program:

`:- p (X1,f(Y1)).`

`p(X2,Y2) :- q(X2,Y2).`

`q(3,Y3) :- r(3,Y3).`

`r(3,f(4)).`

Figure 4-2: An example of the environment closing

The closed binding attempts to meet the requirement demanded in distributed models, i.e., restricted reference, by applying closing concept to the level of OR-task. Namely, every OR-task is maintained with an environment closed against over levels of the tree. The closed binding is uniformly applied to a thread of tasks on a PE. However, the closing operation turns out redundant since the environment for those tasks will always stay local to the PE; therefore, the overhead from the closing operations on a intra-PE thread can not be justified with the restricted access.

4-4 The Principles of Functional Binding

The key idea of the FB scheme is to maintain the environment of an OR-task to be closed to the PE in which the task is scheduled, instead of to the task. Namely, the variables outside the environments are reachable once they are in the local address space of the PE. This enlargement of the closed range from a task to a PE eliminates the need of closing for an intra-PE thread of tasks.

In the FB scheme, variables are divided into two classes; if a variable appears in arguments of structures, it belongs to an **instance** class; otherwise, it belongs to a **non-instance** class. The two classes are dealt with differently:

- for non-instance variables, the effect of environment closing is implicitly achieved at the unification; more precisely, at the unification, the child environment is insulated from the parent environment; a rule specifying a reference direction sets reference bindings from parent to child variables (in the unification algorithm, binding direction is determined such that the environment closing is performed implicitly in unification process; global variables always bind references to formal variables).
- the instance variables are managed such that the structure copying is avoided. Instance variables are named globally so that the variables are viewed with unique name both inside and outside of a task. The bindings of the instance variables in a task are maintained locally in an auxiliary structure allocated to the task. At forward execution (tree expansion), the pointer to the parent's auxiliary structure is passed to the child as part of an input environment, thus every reference to a binding for an instance variable are performed locally in the task. At backward execution (tree retraction, appears during backtracking or resolution of program termination), the pointer to a child auxiliary structure storing bindings of instance variables in a task is passed to the parent's task as part of the output environment.

Thus, back unification at the successful completion of a task is a simple retrieval of bindings, since no explicit closing is performed in this scheme. Namely, bindings for variables in the parent frame are retrieved from those of local variables if they exist, whereas tags of variable in the parent frame are changed from the reference to the variables if the bindings do not exist.

As we say previously, the auxiliary store is utilized to maintain the bindings of instance variables, in order to avoid the structure copying on intra-PE node expansion. The store has a form of a **hash table** in *Figure 4-3 (a)*. As the duplication of the store is always performed in a single address space, we achieve a duplication with a light overhead by just providing a copy of a head as shown in *Figure 4-3 (b)*. Further, the enqueueing of an element to the store is shown in *Figure 4-3 (c)*.

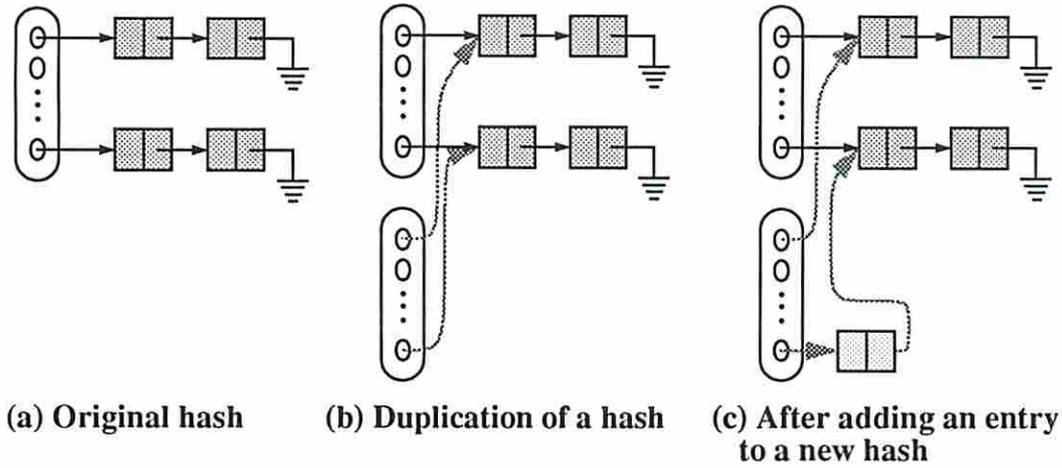


Figure 4-3: Operation with Hash Tables

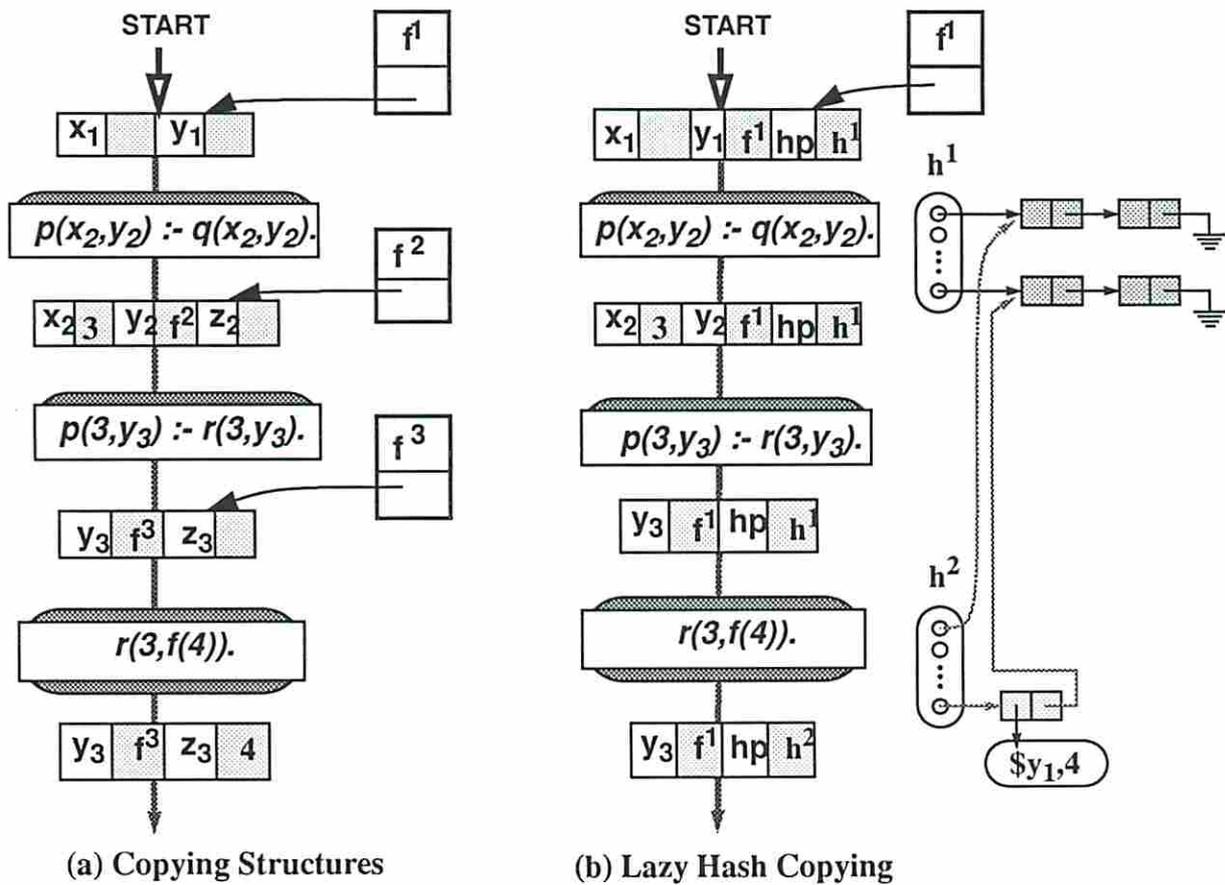


Figure 4-4: Overhead comparison between Structure Copying and Auxiliary Structure

4-5 Conclusion and Remarks

Overall, the FB scheme reduces significantly the extra overhead in an intra-PE task thread considered to be inevitable for a non-single memory space. The reduction of the overhead comes from the following benefits:

- structure copying is not required in an intra-PE task thread, only copying of hash table heads is necessary;
- there are no explicit closing operations both at unification and at back-unification; thus needless of scanning structures entailed in closed binding.

Instead, two new issues are raised in the FB scheme:

- the management of instance variables, especially the detection and the naming convention;
- the management of auxiliary structures for storing the binding of instance variables.

Now, the point is to evaluate the gain in performance brought by this novel binding scheme, and more particularly to verify that the overhead introduced by this new scheme is significantly smaller than the one provided by the “closed binding scheme”. In particular, the management of the instance variables, which are named globally, should remain a critical problem on distributed memory machine implementations.

5 INTERPROCESS COMMUNICATION PRIMITIVES

5-1 Internal Structure of a Translator

Figure 5-1 shows an overall view of a typical translator for our parallel model. It exploits the set of functions given by the Abstract Logic Programing Graph (ALPG), unique and independent of any particular architecture, and translates it into a parallel-C code for a specific underlying machine.

This translator is compound of two main elements, a parallel-C code generator and a runtime system. The parallel-C code generator is in fact the set of ALPG functions writing in C and thus adapted to run on the specific target machine. To perform this, the generator needs the help of a runtime system which provides low level services. It implements primitive languages features, in contrast to the standard library which is typically implemented in the language itself.

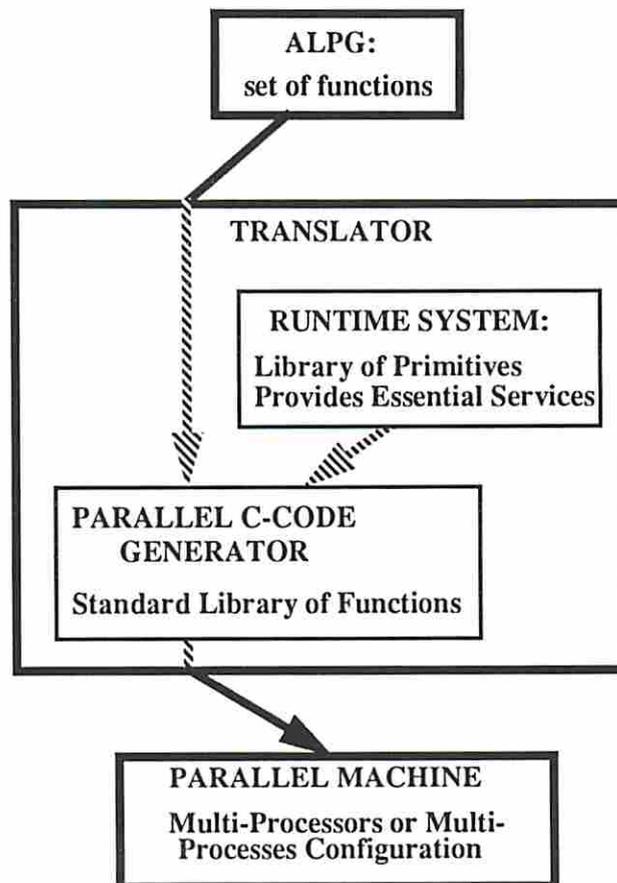


Figure 5-1: Overall View of the Translator

The task of writing a parallel program for a specific machine involves:

- deciding how to decompose the computation into parallel sub-computations,
- mapping these parallel sub-computations to specific processors or processes,
- deciding when to execute them (scheduling),
- and expressing these decisions using the primitives provided by the machine.

The runtime system provides dynamic load balancing, which includes the two first requirements, scheduling, and also garbage collection (i.e., management of dynamically-allocated storage).

A similar translator for Sparc stations will be implemented in the very near future. It will produce an executable code for a unique Sparc station in a multi-processes configuration, and also for several such workstation linked by Internet. We present below a set of primitives that will be part of this expected translator's runtime system (useful for mapping features).

5-2 Interprocess communication

5-2-1 Definitions

A process is an execution environment that consists of three segments: instruction segment, user-data segment, and system-data segment (includes attributes such as current directory, process-ID, open file descriptors, etc.).

A program is a collection of instructions and data that is kept in an ordinary file on disk, which is used to initialize the instruction and user-data segments of a process.

5-2-1 Interprocess Communication in our Application

Several mechanisms (at least 10) exist under UNIX to allow processes to communicate. We review here four of them that could answered our application requirements, which are:

- to allow bidirectional communication,
- to allow two processes with non-common ancestor to communicate,
- to allow communication with different message types,
- reliability (no lost of message, no destination error,...)
- speed: exchanges must be fast.

We could have used **pipes** to insure the interprocess communication. A pipe is a one-way communication mechanism well known to shell users. But pipes have four majors disadvantages, too constraining for our project:

- a pipe is a one-way communication mechanism, and using a same pipe for a two-ways communication between two processes is possible but very complicated;
- reads and writes are not guaranteed to be atomic, prohibiting the use of pipes when there are multiple readers or multiple writers;
- pipes are a simple solution for communicating between processes that are related, typically parent and child. It is also impossible for two processes with no common ancestor to communicate, which is intolerable in our application;
- finally, pipes might be too slow; the data has to be copied from the writing user process to the kernel and back again to the reader; no actual I/O is performed, but the copying alone can take too long for our purpose.

FIFOs, or named pipes, solve the three first disadvantages of pipes. A FIFO exists as a special file, and any process with permission can open it for reading or writing. Atomicity is guaranteed: the bytes written or read via a single system call are always contiguous. Hence, both multiple writer and multiple readers are easily handled. FIFOs are also easy to program. The problem is that they don't eliminate the fourth disadvantage of pipes: as we will demonstrate, they are sometimes to slow.

Message System Calls, a interprocess communication feature of System V, regroups all the advantages of FIFOs, and it is faster. A message is a small amount of data (500 bytes, say) that can be send to a message queue. Messages can be of different types, and any process with appropriate permissions can receive messages from a queue. As we will see, this mechanism provides very good results in terms of performance. However, Message System Calls present several drawbacks: they are complex, incompletely document and above all, they are nonportable, which entails several problems when transporting the application from one machine to another one.

Socket-Based Interprocess Communication is a framework for transport-level programming. Sockets are the endpoints of communication channels. Two unrelated processes can create sockets separately and then send messages between them. Messages can be of different types and sizes. The communication is bidirectional, reliable, error-free, fast, and possible between any two processes that reside either on the same or on separate machines linked by Internet network. We have selected this communication mechanism, because it happened to be the one which gave the best answer to the semantic and performance requirements of our application.

5-3 FIFOs

A FIFO combines features of a file and a pipe. Like a file, it has a name, and any process with appropriate permissions may open it for reading or writing. Unlike with pipe, then, unrelated processes may communicate over a FIFO, since they need not rely on inheritance alone to access it.

Once opened, however, a FIFO acts more like a pipe than a file. Written data is read back in first-and-first-out order, and single write and read system calls are guaranteed to be atomic, provided the amount read or written doesn't exceed the capacity of the FIFO, which is the same as the capacity of a pipe (implementation dependant, but at least 4096 bytes). Data once read can't be read again.

Plain FIFOs don't work well as message carriers; some additional mechanism has to be added. We thus implemented four primitives, **queue**, **send**, **receive** and **rmqueue**. Here are their headers:

```
static int queue(key)    return the queue ID, and create one if necessary.
    long key;

boolean send(dstkey, buf, nbytes)
boolean receive(srckey, buf, nbytes)
    long dstkey, srckey;  destination and source key;
    struct msgbuf *buf;   buf points to the message;
    int nbytes;          size of the message.

void rmqueue(key)       remove the queue identified by key.
    long key;
```

Messages are sent to a **queue**, identified by a long integer called the **key**. **buf** points to the message, which is **nbytes** in **length**. A message may consist of any arbitrary data, but it must begin with a long integer that is unused. That is, a message might be structured like this:

```
struct msgdata
{
    long unused;
    char data[100];
}
```

This allow room for 100 bytes of message data. The header declare **buf** as a pointer to a **char** instead of a pointer to a **struct msgdata** so that the actual message structure can vary with the application.

The queue can get full. If it does, **send** blocks until it empties enough to hold the message being sent.

receive takes the oldest message off the queue and copies it to storage pointed to by **buf**. It is a serious error if the **nbytes** argument to **receive** does not exactly equal the corresponding argument to the call to **send** that deposited the message. The receiver must know the size of the message at

the head of the queue.

In order for two processes to communicate with **send** and **receive**, they must know each other's receiving queue keys. Each can then send messages to the other's queue. They could communicate with only one queue, but they then have to synchronize themselves to avoid problem inherent to pipe characteristic when using in a two-way communication purpose. We avoid it by using two pipes instead of one for two ways communication. In the case of a centralized server and its clients, the server's key should be established in advance and told to each client. Each client makes up its own key, its process number here, since that guaranteed uniqueness. A client passes its key to the server with each request for service so the server knows on which queue to respond.

Here's how to implement send and receive with FIFOs: a queue is a FIFO. We'll construct its name by converting the key to a string and prefixing it with `/tmp/fifo`. For example, the queue corresponding to key 12345 would be `/tmp/fifo12345`.

Our implementation must avoid two major problems. First, we don't want a sender to block permanently if the expected receiver never opens its FIFO for reading. This might hang the server if a client terminates abnormally, which would deny services to all clients. But we also don't want the sender to give up too soon if the receiver isn't ready, because in starting the application system we have to create several processes (the server and its initial clients), and it might take a few seconds for every participant to get rolling. So a sender sets `O_NDELAY` when opening a FIFO for writing, and, if it fails because no client had the FIFO open, it sleeps for a while and tries again. After a few time it gives up. The symbolic constant `NAPTIME` is equal to the number of seconds to sleep; `MAXTRIE` is equal to the number of tries.

We care a lot less about whether the receiver blocks. In our application, a client can safely blocks waiting for the server to respond.

The second major problem is that a server with many clients can easily run out of descriptors (20 per process). This can occur if a parent node in a Prolog Tree has to create several child nodes which can then run in parallel. A process has three standard file descriptors (0, 1, and 2), one for its receiving queue, one for each client's queue, and one for each data file. Since all client FIFOs can't be open at once, the obvious solution is to open a FIFO, write the message, and then close it to free the file descriptor. But opens take too long, so this would slow down communication too much. After all, it's when the server has many clients that efficiency matters most.

A good solution is to use a scheme analogous to that used by virtual memory systems. Keep a limited number of FIFOs open, say seven. When a eighth is needed, close an open one and use its file descriptor. The best one to close is the one that will be needed furthest in the future (this can be proven), but unfortunately, the server can't predict the future. So a reasonable compromise is to close the FIFO that was least recently used, on the theory that a FIFO that hasn't been used for

a while won't be used for a while yet. If the clients make server request in round-robin order, our scheme will fail, since the least-recently-used FIFO is precisely the wrong one to close. But if each client tends to bunch its requests, which is the case in our application where only one transaction is generally executed between two processes at a time (forward or backward execution, see *section 4-4*), that least-recently-approach works well.

We can keep a table of the open file descriptors like this:

```
static struct
{
    long key;
    int fd;
    int time;
}
```

Whenever we read or write a FIFO, we report the current time in the **time** member of the corresponding **fifo** element. If we need to usurp a file descriptor, we just go through the array looking for the element with the oldest time. We close its **fd** member, open the new FIFO, and store the key, file descriptor, and current time in that element.

Since we are using the time member only to record relative times, we don't need to actually interrogate the computer's clock to store the current time. We can implement a local variable (**clock**) each time **send** and **receive** is called, and then store the current value of **clock** in the time member.

Figure 5-2 explicates the way FIFOs interprocess communication primitives have to be use, which is rather easy. The server calls "send" which creates a server's FIFO if it doesn't exist, and "waits" for the client's FIFO to be created. If it is not, the server will give up after several tries; else, it sends the message to the client. At the end of the transaction, the two FIFO queue names are removed.

The code for these primitives, as well as two example programs that exploit and test them is visible in *appendix A*. The results of these tests showed that the FIFOs are very reliable, easy to use, but also much too slow for our application: the average time for a exchange is about 0,5 second! This is the reason why we abandoned them.

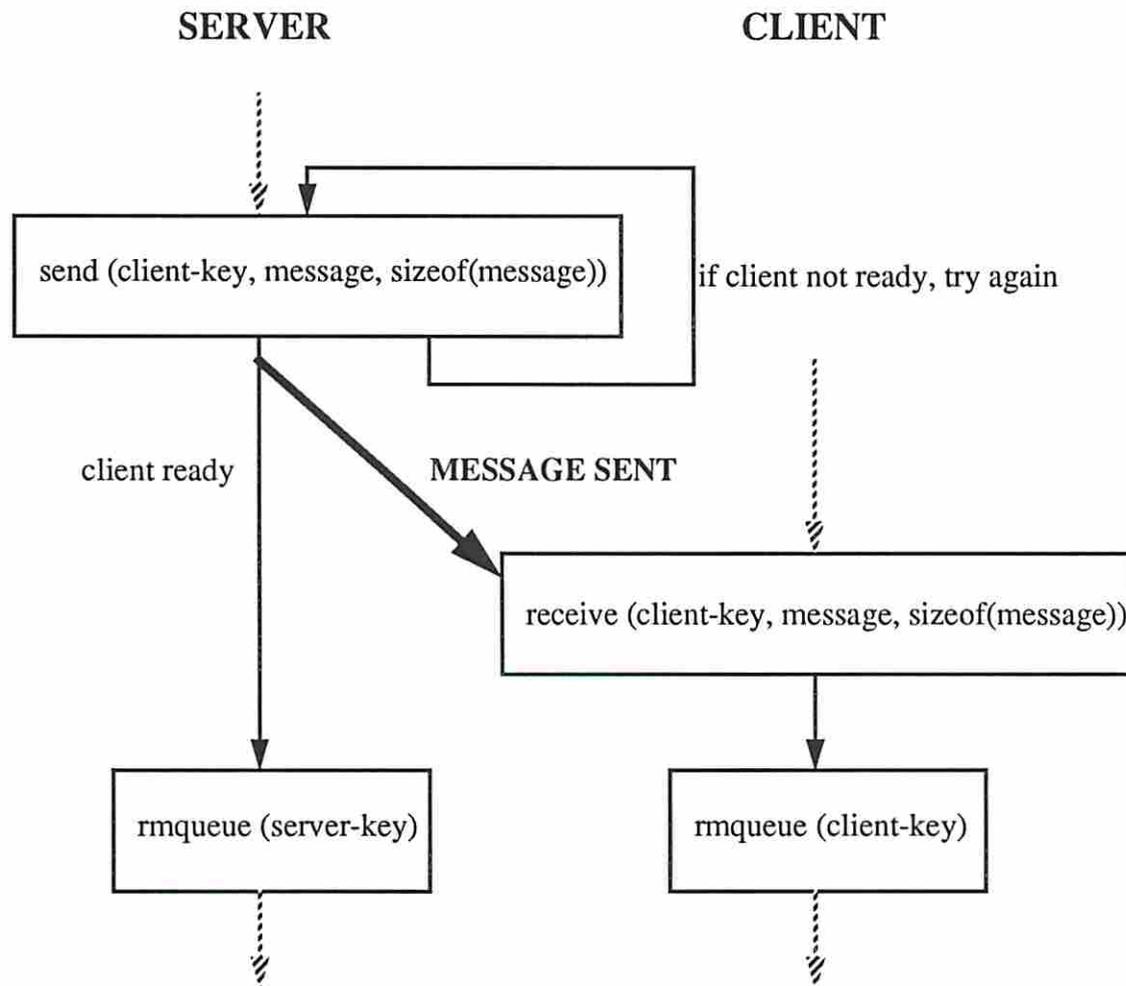


Figure 5-2: FIFOs Interprocess Communication Primitive Calls

5-4 Message System Calls

Four system calls (visible bellow) are available in System V, handle messages analogously to `send` and `receive`, which were implemented in the previous section with FIFOs. The SystemV scheme is much more elaborate, however, and has many more options (that we won't use in our application).

```
int msgget(key, flags)
      key-t key;
      int flags;
```

```
return the message queue-ID, and create one if necessary
queue key (long)
option flags
```

<i>int msgsnd(qid, buf, nbytes, flags)</i>	send message to queue
<i>int msgrcv(qid, buf, nbytes, mtype, flags)</i>	receive message
<i>int qid;</i>	queue-ID
<i>struct msgbuf *buf;</i>	buf points to the message
<i>int nbytes;</i>	size of the message
<i>long mtype;</i>	type of the message
<i>int flags;</i>	option flags
<i>int msgctl (qid, cmd, sbuf)</i>	control message queue
<i>int qid;</i>	
<i>int cmd;</i>	command
<i>struct msqid_ds *sbuf;</i>	pointer to status buffer

To use messages we start with **msgget**, which is analogous to **open**. It takes a key, which must be a long integer, and returns an integer called the queue-ID. A queue-ID is like a file descriptor, except that any process that knows its value may use it (it doesn't have to be inherited to be valid). Like a file descriptor, a queue-ID is an index into a table inside the kernel. By translating external keys to queues-IDs, a time-consuming look up can be avoided when the queue has to be referenced.

The flags argument to **msgget** is needed to see if the queue already exists; if so, it isn't created. This avoid a clash with an existing queue, which would be possible where an actual key to be used. In our application, this feature isn't useful: we'll use the process-ID as the key for a private key.

Once we have a queue-ID, we can call **msgsnd** to put a message on it. The second argument, **buf**, points to an arbitrary structure that must begin with a long integer greater than zero, called the message type (the message structure defined in the previous section qualify). **nbytes** is the number of bytes in the message exclusive of the message type. The last argument, **flags**, is normally 0, causing **msgsnd** to block if the queue is full. If it is set instead to **IPC_NOWAIT** (which acts like **O_NDELAY** on FIFOs), then **msgsnd** returns immediately with an error if the queue is full.

The receiver calls **msgrcv**. **nbytes** is set to the size of the largest message that will fit in the storage area pointed to by **buf**, again exclusive of the message type member. Since the size of the actual message received can be less than **nbytes**, it is returned as the value of **msgrcv** (analogous to **read**). If the receiver wants messages only of a certain type, **mtype** is set to the type number. Otherwise, it should be zero, in which case the oldest message on the queue, regardless of type, will be received (what we chose to implement). If no appropriate message is on the queue, the receiver blocks if **flags** is zero. If **IPC_NOWAIT** flag is on, the receiver returns with an error

instead of blocking.

`msgctl` interrogates or controls various properties of the queue such as access permissions, ownership, and capacity. We'll skip the details, not useful in our application.

From these four system calls, we implemented four primitives very similar to the FIFO ones: they've got the same headers, and are used in the same way (see *figure 5-2*). The main difference comes from the fact that the sender or the receiver blocks only if the queue is respectively, full or empty. This is OK because unlike FIFOs, the sender can put messages on the queue without waiting for the receiver; there is no concept of "open for reading".

The MessageV interprocess primitives are visible in *Appendix B*. We tested them with exactly the same two programs we implemented for the FIFOs primitives. In comparison, they happened to be much faster. However, we met a lot of problems in term of reliability when transporting our software from one machine to another one: system errors occurred from time to time, like "no place left on device", and it was impossible to determine the exact origin of the problem with the documentation we possessed. For this reason, we decided to not use them.

5-5 Sockets

Processes can communicate via UNIX sockets. Sockets are the endpoints of communication channels. When sockets are created by different programs, they have to be named to refer to one another. Names generally must be translated into addresses for use.

The space from which a address is specified by a **domain**; there are several such domains for sockets, in particular **UNIX** domain and **Internet** domain. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to it by given its pathname. UNIX domain names, thus, allow communication between any two processes located on the same machine and that are able to access the socket pathnames. The INTERNET domain is the UNIX implementation of the DRPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between separate machines linked by network.

Communication can be either through a **stream** socket or by **datagram**. Stream communication implies a connection. The communication is bidirectional, error-free, reliable, and, as in pipe, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to `write()` or only part of the data from a single call, if there is not enough room from the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return

error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connection. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The choice between sockets streams and datagrams has been made by carefully considering the semantic and performance requirements of our application. Stream can be both advantageous and disadvantageous. One disadvantage could be that, since a process is only allowed a limited number of file descriptors (20), there is a limit on the number of streams that a process can have opened at any given time. This doesn't really imply any restriction in our application, since each socket is immediately closed after an exchange (and so its corresponding file descriptor is freed), so that even if a server has to communicate with over 20 clients, it will always dispose of available file descriptors. An other drawback is that for delivering a short message, the stream setup and teardown can be unnecessarily long. Weighed against this are the reliability built into the streams, and the fact that using datagrams increase the complexity of the program, which must now concern itself with lost or out of order messages. Considering that in our application, the safety of the communication is one of our major preoccupation, we relied upon the principle of socket stream.

Here we detail the set of primitives we have implemented and explain how to use them.

To allow two processes to communicate, both of them has to create its own socket by calling the following primitive, which returned a socket file descriptor (integer):

```
create_socket()
```

Then, in order for processes to rendezvous, the server must assign a name to its socket by calling "bind()" system call.

Names in the UNIX domain are path names. When a name is bound into the name space, a file (vnode) is allocated in the file system. If the vnode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause a directory to fill up with these objects. That's why we remove systematically the name when the communication is over by calling "unlink()".

Internet address specify a host address and a delivery slot, or port, on that machine. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed.

The server then calls "listen()", which marks the socket as willing to accept connections and ini-

tializes the queue of pending connections (in case of several clients attempt to connect more or less simultaneously). When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of “listen()”.

Then, the “accept()” call takes a pending connection request from the queue if one is available, or blocks waiting for a request. It also returns a new file descriptor (socket) and messages are written or read from the connection socket.

All these features are included in the primitives:

server_send_data() the server writes the message
server_receive_data() the server reads the message

A client initiates a connection with the server using “accept()”, specifying the address to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the client can begin to read or send messages. The messages will be delivered in order without message boundaries. The connection is destroyed when either socket is closed (or soon thereafter). To perform that, we implemented:

client_receive_data() the client reads the message
client_send_data() the client sends the message

After the communication is done, we safely close the two sockets using:

delete_socket()

Figure 5-3 and *5-4* shows the primitive calls used in typical communications: on the first one, the server is the sender; on the second one, the server is the receiver.

The code of all these primitives, as well as several example programs which exploit them, is visible in *Appendix C*. The socket-based interprocess mechanism has showed great performances in terms of reliability and speed. Even if they are a bit more complicated to use than the two previous mechanisms, FIFOs and MessageV, they will be further use by the translator dedicated to Sparc station.

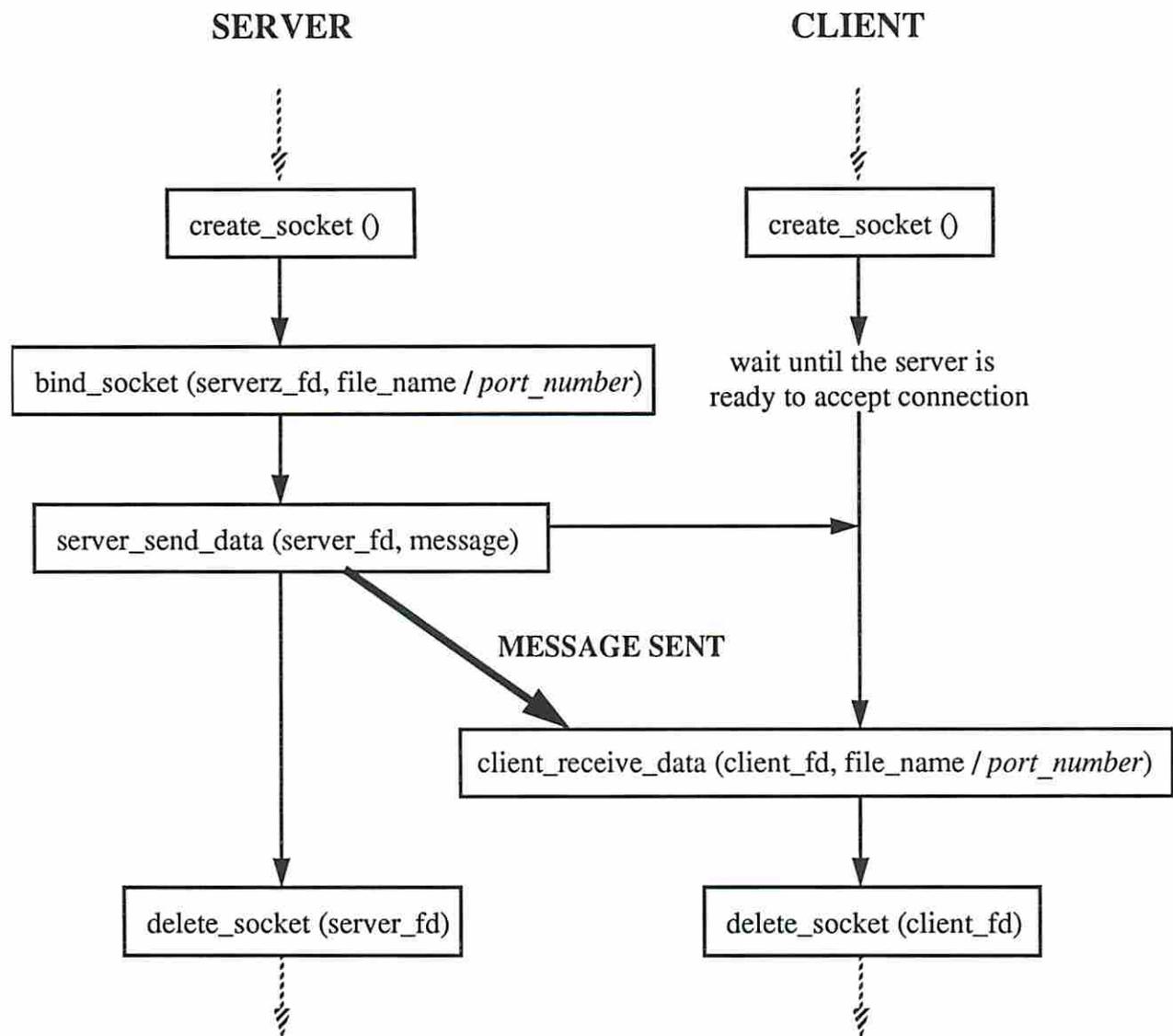


Figure 5-3: Socket_primitive calls - the server is the sender

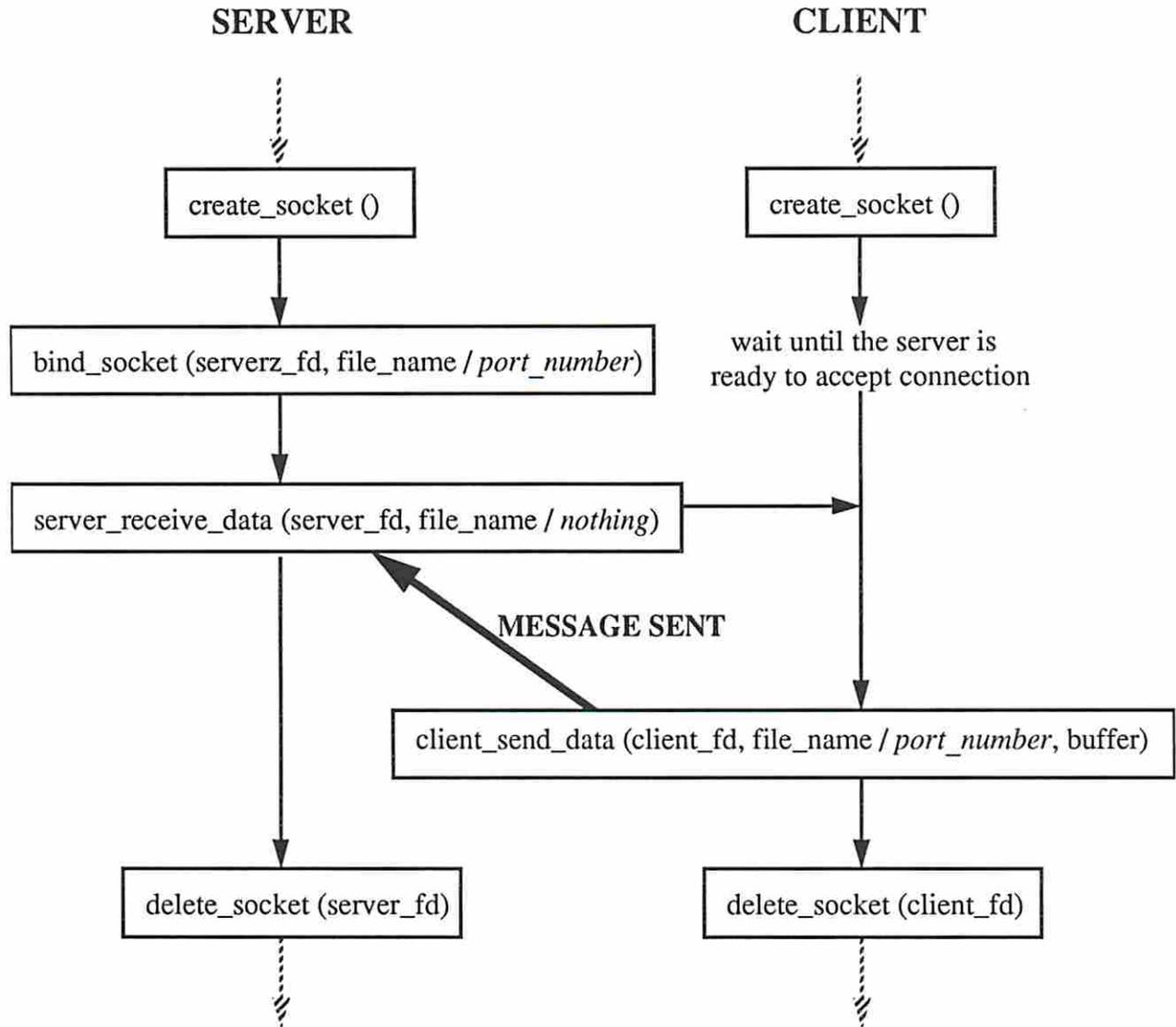


Figure 5-3: Socket_primitive calls - the client is the sender

6 CONCLUSION

In this report, we have briefly described a novel parallel execution model for logic programming, characterized by its adaptability for implementation on large scale architectural platforms. This model applies the principle of data-driven execution to the inference mechanism of Prolog.

Particularly emphasis was placed on its lowest level, the binding environment. The USC Research Team designed the "Functional Binding Scheme", which is highly optimized for non-single address space machines. Since this new model deals with the management of global variables, which could prove to be source of overhead on distributed memory implementation, it is necessary to further evaluate its real benefit in comparison with the already existing "Closed Binding Scheme".

To this end, the USC Research Group is implementing a parallel compiler for a pure logic kernel (i.e, a subset) of the Prolog language, in conjunction with specific translators for various parallel machines. Input a Prolog programs will enable the compiler to generate a graphical intermediate representation, which can be then tailored to the target machines through appropriate translators.

One of these translators will be specific to a Sparc station in a multiple processes configuration, and also to several such workstations linked by network. The purpose of this application is to expand the expressive power of Prolog to encompass distributed applications, and also to test the functionality and the reliability of our parallel execution model. It is not geared toward a performance improvement over normal Prolog sequential systems.

Therefore, we found necessary to implement a comprehensive set of interprocess communication primitives by studying, testing, and comparing several interprocess communication systems available under UNIX. In our opinion, "Socket-Based Interprocess Communication" proved to be the one which answered the best our application requirements, that is, speed, reliability, different message types exchanges and unrelated processes communication.

Finally, in the prospect of significantly improve the execution of Prolog programs using parallel architectures, further work for this project will be the implementation of translators for the Fujitz's AP1000 and for the Motorola's Monsoon machines. This should be done for the end of this year.

APPENDIX

FIFO_message.h

```

/*****
/*****
/**
/** This file contains the message structure used by the
/** FIFO interprocess communication mechanism
/**
/*****
/*****

#include <stdio.h>

typedef struct {
    long unused;
    int pid;
    int number;
} MESSAGE;

```

93/09/02
10:39:24

FIFO_primitives.c

1

```

/*****
/*****
/**
/** This file contains the set of primitives used by the FIFO's
/** interprocess communication mechanism
/**
/*****
/*****

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAXOPEN 20 /* number of message queues allowed */
#define NAPTIME 5 /* number of seconds to sleep before next trie */
#define MAXTRIES 3 /* number of tries to open a FIFO for writing when */
/* no reader have the FIFO open */

typedef enum{FALSE, TRUE} BOOLEAN;

/*****
/*
/* This function prints system call error message and terminate
/*
/*****

void syserr(msg) /* print system call error message and terminate */
char *msg;
{
extern int errno, sys_nerr;
extern char *sys_errlist[];

fprintf(stderr, "ERROR:%s (%d", msg, errno);
if (errno > 0 && errno < sys_nerr)
fprintf(stderr, ":%s\n\n", sys_errlist[errno]);
else
fprintf(stderr, ")\n\n");
exit(0);
}

/*****
/*
/* This function constructs a fifo name from key and returns an
/* queue ID
/*
/*****

static char *fifoname(key)
long key;
{

```

93/09/02
10:39:24

FIFO_primitives.c

2

```

static char fifo[20];
sprintf(fifo, "/tmp/fifo%d", key);
return(fifo);
}

static int openfifo(key, flags) /* return fifo fd */
long key;
int flags;
{
    static struct
    {
        long key;
        int fd;
        int time;
    } fifos[MAXOPEN];
    static int clock;
    int i, avail, oldest, fd, tries;
    char *fifo;
    extern int errno;

    clock++;
    avail = -1;
    for (i = 0; i < MAXOPEN; i++)
    {
        if (fifos[i].key == key)
        {
            fifos[i].time = clock;
            return(fifos[i].fd);
        }
        if (fifos[i].key == 0 && avail == -1)
            avail = i;
    }
    if (avail == -1) /* all fds in use; find oldest */
    {
        oldest = -1;
        for (i = 0; i < MAXOPEN; i++)
            if (oldest == -1 || fifos[i].time < oldest)
            {
                oldest = fifos[i].time;
                avail = i;
            }
        if (close(fifos[avail].fd) == -1)
            return(-1);
    }
    fifo = fifoname(key);
    if (mkfifo(fifo) == -1 && errno != EEXIST)
        return(-1);
    for (tries = 1; tries <= MAXTRIES; tries++)
    {
        if ((fd = open(fifo, flags | O_NDELAY)) != -1)
            break;
        if (errno != ENXIO)
            return(-1);
        printf("dodo\n");
        sleep(NAPTIME);
    }
    if (fd == -1)
    {
        errno = ENXIO; /* sleep may have miss up */
        return(-1);
    }
    if (fcntl(fd, F_SETFL, flags) == -1) /* clear O_NDELAY */
        return(-1);
    fifos[avail].key = key;

```

93/09/02
10:39:24

FIFO_primitives.c

3

```

    fifos[avail].fd = fd;
    fifos[avail].time = clock;
    return(fd);
}

/*****
/*
/* This function sends a message to the destination queue (key)
/*
/*
*****/

BOOLEAN send(dstkey, buf, nbytes)
    long    dstkey;
    struct msgbuf *buf;
    int     nbytes;
{
    int fd;
    if ((fd = openfifo(dstkey, O_WRONLY)) == -1)
        return(FALSE);
    return(write(fd, buf, nbytes) != -1);
}

/*****
/*
/* This function receives a message from the source queue (key)
/*
/*
*****/

BOOLEAN receive(srckey, buf, nbytes)
    long    srckey;
    struct msgbuf *buf;
    int     nbytes;
{
    int fd, nread;
    if ((fd = openfifo(srckey, O_RDONLY)) == -1)
    {
        printf("receive missed");
        return(FALSE);
    }
    while((nread = read(fd, buf, nbytes)) == 0);
    sleep(NAPTIME);
    return(nread != -1);
}

/*****
/*
/* This function removes the message queue identified by the key
/*
/*
*****/

void rmqueue(key)
    long key;
{
    int errno;
    if (unlink(fifoname(key)) == -1 && errno != ENOENT)
        syserr("unlink");
}

```

93/09/01
12:44:41

FIFO_example1.c

1

```

/*****
/*****
/**                                     **/
/** This source creates a child process clone to the parent's one.    **/
/** They communicate by using FIFO. The parent send two messages to  **/
/** the child who return the first one only.                          **/
/**                                     **/
/*****
/*****

#include "FIFO_primitives.c"
#include "FIFO_message.h"

main()
{
    int parent, child;
    MESSAGE m, n;

    /*****
    /*****
    /*                                     */
    /* fork creates a new process with almost exact copies of instruction, */
    /* user-data, and system-data segments. After fork return, both parent */
    /* and child processes receive the return. The return value is        */
    /* different, however, which is crucial, because it allows their       */
    /* subsequent actions to differ:                                       */
    /* - the child receives a 0 return value;                               */
    /* - the parent receives the process-ID of the child.                  */
    /*                                     */
    /*****
    /*****

    if ((child = fork()) == -1) /* creates a child process */
    {
        perror("fork");
        exit(0);
    }
    else
    {
        if (child != 0)          /* this the parent, the return value is <> 0 */
        {
            parent = getpid();
            printf("parent = process # %d\n\n", parent);

            /* send message m to the child */
            m.pid = getpid();      /* send address for eventual return */
            m.number = 1;         /* send value 1 */
            if (!send(child, &m, sizeof(m)))
                syserr("send1\n");

            /* receive message n from the child */
            setbuf(stdout, NULL); /* turns off buffering */
            if (!receive(m.pid, &n, sizeof(n)))
                syserr("received2");

            /* send message n to the child */
            n.number = 3;
            if (!send(n.pid, &n, sizeof(n)))
                syserr("send1\n");

```

93/09/01
12:44:41

2

FIFO_example1.c

```
/*
*****
** The following instruction "sleep" is necessary to allow the
** child process to print the successful message "data receive"
** It also indicates the time necessary for the four exchanges:
** almost 5 seconds !!!
*****
sleep(5);

rmqueue(parent);      /* remove the parent queue */
}

if (child == 0)        /* this is the child; the return value is = 0 */
{
    child = getpid();
    printf("child = process # %d\n\n", child);

    /* receive message m from the parent */
    setbuf(stdout, NULL); /* turns off buffering */
    if (!receive(child, &m, sizeof(m)))
        syserr("received1\n");

    /* send message n to the parent */
    n.pid = getpid();
    n.number = 2;
    if (!send(m.pid, &n, sizeof(n)))
        syserr("send2");

    /* receive message n from the parent */
    setbuf(stdout, NULL); /* turns off buffering */
    if (!receive(n.pid, &n, sizeof(n)))
        syserr("received2");
    printf("Have I well received 3 from the son ? %d\n\n", n.number);

    rmqueue(child);      /* remove the child queue */
}
}
}
```

93/09/01
12:44:52

FIFO_example2.c

1

```

/*****
/*****
/**
/** This source creates a child process clone to the parent's one.
/** They communicate by using FIFO. The parent send two messages to
/** the child who return them.
/**
/*****
/*****

#include "FIFO_primitives.c"
#include "FIFO_message.h"

main()
{
    int parent, child, sol;
    MESSAGE m, n;

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*****
    /*****

    if ((child = fork()) == -1) /* create a child process */
    {
        perror("fork");
        exit(0);
    }
    else
    {
        if (child != 0) /* this the parent, the return value is <> 0 */
        {
            parent = getpid();
            printf("parent = process # %d\n\n", parent);

            /* send message m to the child */
            m.pid = getpid(); /* sent address for eventual return */
            m.number = 1; /* sent value 1 */
            if (!send(child, &m, sizeof(m)))
                syserr("send1\n");

            /* receive message n from the son */
            setbuf(stdout, NULL); /* turns off buffering */
            if (!receive(m.pid, &n, sizeof(n)))
                syserr("received2");
            printf("Have I well received 2 from the son ? %d\n\n", n.number);

            rmqueue(parent); /* remove the parent queue */
        }
    }
}

```

93/09/01
12:44:52

FIFO_example2.c

2

```
if (child == 0)          /* this is the child; the return value is = 0 */
{
    child = getpid();
    printf("child = process # %d\n\n", child);

    /* receive message m from the parent */
    setbuf(stdout, NULL); /* turns off buffering */
    if (!receive(child, &m, sizeof(m)))
        syserr("received1\n");

    /* send message n to the parent */
    n.pid = getpid();
    n.number = 2;
    if (!send(m.pid, &n, sizeof(n)))
        syserr("send2");

    rmqueue(child);      /* remove the child queue */
}
}
```


MESSV_primitives.c

```

/*****
/*****
/**
/** This file contains the set of primitives used by the Message
/** System V interprocess communication mechanism
/**
/*****
/*****/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAXOPEN 20 /* number of message queues allowed */

typedef enum{FALSE, TRUE} BOOLEAN;

/*****
/*
/* This function prints system call error message and terminate
/*
/*****/

void syserr(msg)
char *msg;
{
extern int errno, sys_nerr;
extern char *sys_errlist[];

fprintf(stderr, "ERROR:%s (%d", msg, errno);
if (errno > 0 && errno < sys_nerr)
    fprintf(stderr, ":%s\n", sys_errlist[errno]);
else
    fprintf(stderr, ")\n");
exit(0);
}

/*****
/*
/* This function returns an queue ID (creates one if necessary)
/*
/*****/

static int openqueue(key)
long key;
{
static struct
{
long key;
int qid;
} queues[MAXOPEN];
```

93/09/02
10:53:33

MESSV_primitives.c

```

int      i, avail, qid;
extern int errno;

avail = -1;
for (i=0; i<MAXOPEN; i++)
{
    if (queues[i].key == key)
        return(queues[i].qid);
    if (queues[i].key == 0 && avail == -1)
        avail = i;
}
if (avail == -1)
{
    errno = 0;
    return(-1);
}
if ((qid = msgget(key, 0666|IPC_CREAT)) == -1)
{
    syserr("msgget");
    return(-1);
}
queues[avail].key = key;
queues[avail].qid = qid;
return(qid);
}

/*****
*/
/* This function sends a message to the destination queue (key) */
/*
*/
/*****

BOOLEAN send(dstkey, buf, nbytes)
long      dstkey;
struct msgbuf *buf;
int      nbytes;
{
    int qid;
    if ((qid = openqueue(dstkey)) == -1)
        printf("send missed");
    buf->mtype = 1;
    return(msgsnd(qid, buf, nbytes - sizeof(buf->mtype), 0) != -1);
}

/*****
*/
/* This function receives a message from the source queue (key) */
/*
*/
/*****

BOOLEAN receive(srckey, buf, nbytes)
long      srckey;
struct msgbuf *buf;
int      nbytes;
{
    int qid;
    if ((qid = openqueue(srckey)) == -1)
        return(FALSE);
    return(msgrcv(qid, buf, nbytes - sizeof(buf->mtype), 0L, 0) != -1);
}

```


93/09/02
11:12:19

MESSV_example1.c

1

```

/*****
/*****
/**
/** This source creates a child process clone to the parent's one.
/** They communicate by using Message System calls. The parent sends
/** two messages to the child who returns the first one only.
/**
/*****
/*****

#include "MESSV_primitives.c"
#include "MESSV_message.h"

main()
{
    int parent, child;
    MESSAGE m, n;

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*****
    /*****

    if ((child = fork()) == -1) /* creates a child process */
    {
        perror("fork");
        exit(0);
    }
    else
    {
        if (child != 0) /* this the parent, the return value is <> 0 */
        {
            parent = getpid();
            printf("parent = process # %d\n\n", parent);

            /* send message m to the child */
            m.pid = getpid(); /* send address for eventual return */
            m.number = 1; /* send value 1 */
            if (!send(child, &m, sizeof(m)))
                syserr("send1\n");

            /* receive message n from the child */
            setbuf(stdout, NULL); /* turns off buffering */
            if (!receive(m.pid, &n, sizeof(n)))
                syserr("received2");

            /* send message n to the child */
            n.number = 3;
            if (!send(n.pid, &n, sizeof(n)))
                syserr("send1\n");

```

93/09/02
11:12:19

MESSV_example1.c

2

```
    /* remove the parent queue */
    rmqueue(parent);
}

if (child == 0)          /* this is the child; the return value is = 0 */
{
    child = getpid();
    printf("child = process # %d\n\n", child);

    /* receive message m from the parent */
    setbuf(stdout, NULL); /* turns off buffering */
    if (!receive(child, &m, sizeof(m)))
        syserr("received1\n");

    /* send message n to the parent */
    n.pid = getpid();
    n.number = 2;
    if (!send(m.pid, &n, sizeof(n)))
        syserr("send2");

    /* receive message n from the parent */
    setbuf(stdout, NULL); /* turns off buffering */
    if (!receive(n.pid, &n, sizeof(n)))
        syserr("received2");
    printf("Have I well received 3 from the son ? %d\n\n", n.number);

    /* remove the child queue */
    rmqueue(child);
}
}
```

93/09/02
11:12:30

MESSV_example2.c

1

```

/*****
/*****
/**                                     **/
/** This source creates a child process clone to the parent's one.    **/
/** They communicate by using Message System calls. The parent sends **/
/** two messages to the child who returns both of them.              **/
/**                                     **/
/*****
/*****

#include "MESSV_primitives.c"
#include "MESSV_message.h"

main()
{
    int parent, child, sol;
    MESSAGE m, n;

    /*****
    /*****
    /*                                     */
    /* fork creates a new process with almost exact copies of instruction, */
    /* user-data, and system-data segments. After fork return, both parent */
    /* and child processes receive the return. The return value is         */
    /* different, however, which is crucial, because it allows their        */
    /* subsequent actions to differ:                                       */
    /* - the child receives a 0 return value;                               */
    /* - the parent receives the process-ID of the child.                  */
    /*                                     */
    /*****
    /*****

    if ((child = fork()) == -1) /* create a child process */
    {
        perror("fork");
        exit(0);
    }
    else
    {
        if (child != 0)          /* this the parent, the return value is <> 0 */
        {
            parent = getpid();
            printf("parent = process # %d\n\n", parent);

            /* send message m to the child */
            m.pid = getpid();      /* sent address for eventual return */
            m.number = 1;         /* sent value 1 */
            if (!send(child, &m, sizeof(m)))
                syserr("send1\n");

            /* receive message n from the son */
            setbuf(stdout, NULL); /* turns off buffering */
            if (!receive(m.pid, &n, sizeof(n)))
                syserr("received2");
            printf("Have I well received 2 from the son ? %d\n\n", n.number);

            /* remove the parent queue */
            rmqueue(parent);
        }
    }
}

```

MESSV_example2.c

```
if (child == 0)          /* this is the child; the return value is = 0 */
{
    child = getpid();
    printf("child = process # %d\n\n", child);

    /* receive message m from the parent */
    setbuf(stdout, NULL); /* turns off buffering */
    if (!receive(child, &m, sizeof(m)))
        syserr("received1\n");

    /* send message n to the parent */
    n.pid = getpid();
    n.number = 2;
    if (!send(m.pid, &n, sizeof(n)))
        syserr("send2");

    /* remove the child queue */
    rmqueue(child);
}
}
```

UNIX_socket_primitives.c

```

/*****
/*****
/**
/** This program contains all the primitives used to allow communication
/** between processes within the UNIX domain (processes run on the same
/** machine).
/**
/**
/*****
/*****

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_NAME 10

/*****
/*****
/**
/** This primitive prints system call error message and terminate
/**
/**
/*****
/*****

void syserr(msg)
char *msg;
{
extern int errno, sys_nerr;
extern char *sys_errlist[];

fprintf(stderr, "ERROR:%s (%d", msg, errno);
if (errno > 0 && errno < sys_nerr)
    fprintf(stderr, ":%s\n\n", sys_errlist[errno]);
else
    fprintf(stderr, ")\n\n");
exit(0);
}

/*****
/*****
/**
/** This primitive set up all the necessary fields in the "sockaddr_un"
/** structure, which contains the UNIX pathname.
/**
/**
/*****
/*****

struct sockaddr_un get_UNIX_addr(file_name)
char file_name[MAX_NAME];
{
struct sockaddr_un sunix;

sunix.sun_family = AF_UNIX;

```

UNIX_socket_primitives.c

```
strcpy(sunix.sun_path, file_name);

return (sunix);
}

/*****/
/**                                          **/
/** This primitive creates a stream socket in the UNIX domain          **/
/**                                          **/
/*****/

int create_socket()
{
    int socket_fd;

    socket_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (socket_fd == -1)
    {
        syserr("opening stream socket");
        exit(0);
    }
    printf ("socket created # = %d\n\n", socket_fd);
    return (socket_fd);
}

/*****/
/**                                          **/
/** This primitive closes the socket defined by its file descriptor    **/
/**                                          **/
/*****/

void delete_socket(socket_fd)
    int socket_fd;
{
    if (close(socket_fd) == -1)
        syserr("close socket");
    exit(0);
}

/*****/
/**                                          **/
/** This primitive binds a UNIX socket name to the socket defined by its **/
/** socket file descriptor entry argument. This name is actually an UNIX **/
/** address returned by the "get_UNIX_addr" call.                        **/
/** This primitive is used by the server only.                          **/
/**                                          **/
/*****/

void bind_socket(socket_fd, file_name)
    int socket_fd;
    char file_name[MAX_NAME];
{
    struct sockaddr_un address;

    address = get_UNIX_addr(file_name);
    if (bind(socket_fd, (struct sockaddr *)&address, sizeof(struct sockaddr_un)) == -1)
    {
        syserr("binding stream socket");
    }
}
```

UNIX_socket_primitives.c

```
        exit(0);
    }
}

/*****
**
** This primitive is to be used by the server only, just after the "bind
** socket" one, and in relation with "client_receive_data".
** The server sets up a queue for incoming connection requests using
** "listen()". Then it initiates a connection using "accept". The
** "accept" call returns a new file descriptor which is use to send a
** message to the client.
**
*****/
void server_send_data(socket_fd, buffer)
    int socket_fd;
    char buffer[1024];
{
    int msgsock;

    /* Start accepting connection */
    listen(socket_fd, 5);

    msgsock = accept(socket_fd, (struct sockaddr *)0, (int *)0);
    if (msgsock == -1)
        syserr("accept");

    /* write the message for the client */
    else if (write(msgsock, buffer, 1024) == -1)
        syserr("writing on stream socket");
    if (close(msgsock) == -1)
        syserr("close socket");
}

/*****
**
** This primitive is used by the client only, in relation with "server_
** send_data".
** First, it looks up to the server socket UNIX address it wishes to
** connect by using "get_UNIX_addr" call. Then it requests a connection
** using "connect". If the connection is established by the server, the
** two processes can communicate.
** Here, the client reads a message sent by the server.
**
*****/
void client_receive_data(socket_fd, file_name)
    int socket_fd;
    char file_name[MAX_NAME];
{
    char buf[1024];
    struct sockaddr_un server;
    int rval;

    /* Connect socket using get_UNIX_addr call */
    server = get_UNIX_addr(file_name);

    if (connect(socket_fd, (struct sockaddr *)&server, sizeof (struct sockaddr_un)) == -1)
```

UNIX_socket_primitives.c

```
{
    delete_socket(socket_fd);
    syserr("connecting stream socket");
    exit(0);
}

/* read the message sent by the server */
do
{
    if ((rval = read(socket_fd, buf, 1024)) == -1)
        perror("readind stream message");
    else
        printf("message read : %s\n", buf);
}
while (rval > 0);

/* removed the name of the socket on the current directory */
unlink(server.sun_path);
}

/*****
/**
/** This primitive is to be used by the server only, just after the "bind
/** socket" one, and in relation with "client_send_data".
/** The server sets up a queue for incoming connection requests using
/** "listen()". Then it initiates a connection using "accept". The
/** "accept" call returns a new file descriptor which is use to receive a
/** message from the client.
/**
/**
/*****/

void server_receive_data(socket_fd, file_name)
    int socket_fd;
    char file_name[MAX_NAME];
{
    int msgsock;
    char buf[1024];
    int rval;

    /* Start accepting connection */
    listen(socket_fd, 5);

    msgsock = accept(socket_fd, (struct sockaddr *)0, (int *)0);
    if (msgsock == -1)
        syserr("accept");

    /* read the message sent by the server */
    do
    {
        if ((rval = read(msgsock, buf, 1024)) == -1)
            perror("readind stream message");
        else
            printf("message read : %s\n", buf);
    }
    while (rval > 0);
    if (close(msgsock) == -1)
        syserr("close socket");

    /* removed the name of the socket on the current directory */
    unlink(file_name);
}
```

UNIX_socket_primitives.c

```

/*****
/**
/** This primitive is used by the client only, in relation with "server_
/** send_data".
/** First, it looks up to the server socket UNIX address it wishes to
/** connect by using "get_UNIX_addr" call. Then it requests a connection
/** using "connect". If the connection is established by the server, the
/** two processes can communicate.
/** Here, the client send a message to the server.
/**
/**
*****/

void client_send_data(socket_fd, file_name, buffer)
    int socket_fd;
    char file_name[MAX_NAME];
    char buffer[1024];
{
    struct sockaddr_un server;

    /* Connect socket using get_UNIX_addr call */
    server = get_UNIX_addr(file_name);

    if (connect(socket_fd, (struct sockaddr *)&server, sizeof (struct sockaddr_un)) == -1)
    {
        delete_socket(socket_fd);
        syserr("connecting stream socket");
        exit(0);
    }

    /* write the message for the server */
    else if (write(socket_fd, buffer, 1024) == -1)
        syserr("writing on stream socket");
}

```

UNIX_socket_example1.c

```

/*****
/*****
/**
/** This is a example program using sockets within the UNIX domain.
/** A parent process sends a message to a child process.
/** - The parent is the server: it creates a socket, binds a name to it,
/** listens for a connection and accepts it to write the message.
/** - The child is the client: it creates a socket, asks to connect it
/** to the parent's one, and reads the message
/**
/*****
/*****

#include "U_socket_primitives.c"

/*****
/*****
/**
/** This two values may be adjusted with the appliation requirements
/**
/*****

#define NAME "riri" /* pathname */
#define DATA "message sent to the client-child" /* data exchanged */

main()
{
    int parent, child; /* process ID (not necessary) */
    int server_fd, client_fd, msg_sock; /* socket file descriptors */
    char file_name[MAX_NAME];
    char buffer[1024];

    /* initialize the file_name, filfull "buffer" */
    strcpy(file_name, NAME);
    strcpy(buffer, DATA);

    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using file system name */
    bind_socket(server_fd, file_name);

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*****

```

UNIX_socket_example1.c

```
/* **** */
if ((child = fork()) == -1)
{
    perror("fork");
    exit(0);
}
else
{
    if (child != 0)          /* this the parent, the return value is <> 0 */
    {
        /* Start accepting connection and write message to the client */
        server_send_data(server_fd, buffer);
        sleep(1); /* not necessary, allow a good print of the message result */

        /* delete the server socket */
        delete_socket(server_fd);
    }

    if (child == 0)        /* this is the child; the return value is = 0 */
    {
        child = getpid();
        printf("child id = %d\n\n", child);

        /* Create a socket */
        client_fd = create_socket();

        /* Read the message after requesting for connection to server socket */
        client_receive_data(client_fd, file_name);
    }
}
}
```

UNIX_socket_example2.c

```

/*****
/*****
/**
/** This is an example program using sockets within the UNIX domain.
/** A parent process receives a message sent by a child process.
/** - The child is the server: it creates a socket, binds a name to it,
/**     listens for a connection and accepts it to write the message.
/** - The parent is the client: it creates a socket, asks to connect it
/**     to the child's one, and read the message
/**
/*****
/*****

#include "U_socket_primitives.c"

/*****
/**
/** This two values may be adjusted with the appliation requirements
/**
/*****

#define NAME "riri" /* pathname */
#define DATA "message sent to the client-parent" /* data exchanged */

main()
{
    int      parent, child; /* process ID (not necessary) */
    int      server_fd, client_fd, msg_sock; /* socket file descriptors */
    char     file_name[MAX_NAME];
    char     buffer[1024];

    /* initialise "file_name", filfull "buffer" */
    strcpy(file_name, NAME);
    strcpy(buffer, DATA);

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*****
    /*****

    if ((child = fork()) == -1)
    {
        perror("fork");
        exit(0);
    }
    else
    {

```

UNIX_socket_example2.c

```
if (child != 0)          /* this the parent, the return value is <> 0 */
{
    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* Create a socket */
    client_fd = create_socket();

    /******
    /* The following instruction "sleep" is here to insure that the    */
    /* server (here the child) has had the time to create its own    */
    /* socket and is ready to accept connection BEFORE the client    */
    /* attempts to make a request.                                    */
    /******
    sleep(1);

    /* Read the message after requesting for connection to server socket */
    client_receive_data(client_fd, file_name);

    /* delete the client socket */
    delete_socket(client_fd);
}

if (child == 0)          /* this is the child; the return value is = 0 */
{
    child = getpid();
    printf("child id = %d\n\n", child);

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using file system name */
    bind_socket(server_fd, file_name);

    /* Start accepting connection and write message to the client */
    server_send_data(server_fd, buffer);

    /* delete the server socket */
    delete_socket(server_fd);
}
}
```

93/09/03
20:21:30

UNIX_socket_example3.c

1

```

/*****/
/*****/
/**
/** This is a example program using sockets within the UNIX domain.
/** A parent process receives a message sent by a child process.
/** - The parent is the server: it creates a socket, binds a name to it,
/** listens for a connection and accepts it to receive the message.
/** - The child is the client: it creates a socket, asks to connect it
/** to the parent's one, and sends the message.
/**
/*****/
/*****/

#include "U_socket_primitives.c"

/*****/
/**
/** This two values may be adjusted with the appliation requirements
/**
/*****/

#define NAME "riri" /* pathname */
#define DATA "message sent to the server-parent" /* data exchanged */

main()
{
    int parent, child; /* process ID (not necessary) */
    int server_fd, client_fd, msg_sock; /* socket file descriptors */
    char file_name[MAX_NAME];
    char buffer[1024];

    /* initialize the file_name, filfull "buffer" */
    strcpy(file_name, NAME);
    strcpy(buffer, DATA);

    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using file system name */
    bind_socket(server_fd, file_name);

/*****/
/*****/
/**
/** fork creates a new process with almost exact copies of instruction,
/** user-data, and system-data segments. After fork return, both parent
/** and child processes receive the return. The return value is
/** different, however, which is crucial, because it allows their
/** subsequent actions to differ:
/** - the child receives a 0 return value;
/** - the parent receives the process-ID of the child.
/**
/*****/
/*****/

```

UNIX_socket_example3.c

```
/* **** */
if ((child = fork()) == -1)
{
    perror("fork");
    exit(0);
}
else
{
    if (child != 0)          /* this the parent, the return value is <> 0 */
    {
        /* Start accepting connection and read message from the client */
        server_receive_data(server_fd, file_name);

        /* delete the server socket */
        delete_socket(server_fd);
    }

    if (child == 0)        /* this is the child; the return value is = 0 */
    {
        child = getpid();
        printf("child id = %d\n\n", child);

        /* Create a socket */
        client_fd = create_socket();

        /* Write the message after requesting for connection to server socket */
        client_send_data(client_fd, file_name, buffer);
    }
}
}
```

UNIX_socket_example4.c

```

/*****
/*****
/**
/** This is an example program using sockets within the UNIX domain.
/** A parent process sends a message to its child process.
/** - The child is the server: it creates a socket, binds a name to it,
/**     listens for a connection and accepts it to read the message.
/** - The parent is the client: it creates a socket, asks to connect it
/**     to the child's one, and write the message.
/**
/*****
/*****

#include "U_socket_primitives.c"

/*****
/**
/** This two values may be adjusted with the appliation requirements
/**
/*****

#define NAME "riri" /* pathname */
#define DATA "message sent to the server-child " /* data exchanged */

main()
{
    int      parent, child; /* process ID (not necessary) */
    int      server_fd, client_fd, msg_sock; /* socket file descriptors */
    char     file_name[MAX_NAME];
    char     buffer[1024];

    /* initialys "file_name", filfull "buffer" */
    strcpy(file_name, NAME);
    strcpy(buffer, DATA);

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*****
    /*****

    if ((child = fork()) == -1)
    {
        perror("fork");
        exit(0);
    }
    else
    {

```

UNIX_socket_example4.c

```
if (child != 0)          /* this the parent, the return value is <> 0 */
{
    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* Create a socket */
    client_fd = create_socket();

    /******
    /* The following instruction "sleep" is here to insure that the      */
    /* server (here the child) has had the time to create its own      */
    /* socket and is ready to accept connection BEFORE the client      */
    /* attempts to make a request.                                     */
    /******
    sleep(1);

    /* Write the message after requesting for connection to server socket */
    client_send_data(client_fd, file_name, buffer);
    sleep(1); /* not necessary, allow the message result to be well printed*/

    /* delete the client socket */
    delete_socket(client_fd);
}

if (child == 0)          /* this is the child; the return value is = 0 */
{
    child = getpid();
    printf("child id = %d\n\n", child);

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using file system name */
    bind_socket(server_fd, file_name);

    /* Start accepting connection and read message sent by the client */
    server_receive_data(server_fd, file_name);

    /* delete the server socket */
    delete_socket(server_fd);
}
}
```

93/09/08
12:23:35

INTERNET_socket_primitives.c

1

```

/*****
/*****
/**
/** This program contains all the primitives used to allow communication **
/** between processes within the Internet domain (processes can run on **
/** separate machines linked by network). **
/** **
/*****
/*****

```

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/msg.h>

```

```

#define MAX_NAME 48

```

```

/*****
/**
/** This primitive prints system call error message and terminate **
/** **
/*****

```

```

void syserr(msg)
char *msg;
{
    extern int  errno, sys_nerr;
    extern char *sys_errlist[];

    fprintf(stderr, "ERROR:%s (%d", msg, errno);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, ":%s)\n\n", sys_errlist[errno]);
    else
        fprintf(stderr, ")\n\n");
    exit(0);
}

```

```

/*****
/**
/** This primitive set up all the necessary fields in the "sockaddr_in" **
/** structure, which contains the Internet address (i.e., protocol, local **
/** machine address, local port), and returns it to the caller. **
/** **
/*****

```

```

struct sockaddr_in *inet_socketaddr (host, port)
char *host;
int port;
{
    struct sockaddr_in  sin;

```

INTERNET_socket_primitives.c

```
struct hostent      *hp;
char                local_host[MAX_NAME];

if (host == NULL) /* default to current host */
{
    if (gethostname(local_host, MAX_NAME) == -1)
    {
        syserr("gethostname()");
        exit(0);
    }
    host = local_host;
}

hp = gethostbyname(host);
bzero ((char *)&sin, sizeof(struct sockaddr_in));
bcopy ((char *)hp -> h_addr, (char *)&(sin.sin_addr), hp -> h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = htons(port);

return (&sin);
}

/*****
**
** This primitive creates a stream socket in the Internet domain
**
*****/

int create_socket()
{
    int socket_fd;

    socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd == -1)
    {
        syserr("opening stream socket");
        exit(0);
    }
    printf ("socket created # = %d\n\n", socket_fd);
    return (socket_fd);
}

/*****
**
** This primitive closes the socket defined by its file descriptor
**
*****/

void delete_socket(socket_fd)
int socket_fd;
{
    if (close(socket_fd) == -1)
        syserr("close socket");
    exit(0);
}

/*****
**
*****/
```

INTERNET_socket_primitives.c

```
/** This primitive binds a Internet socket name to socket defined by the **/  
/** socket file descriptor entry argument. This name is actually an **/  
/** Internet address given by the "inet_socketaddr" call. **/  
/** This primitive is used by the server only. **/  
/** **/  
/*****/  
  
void bind_socket(socket_fd, port_number)  
    int socket_fd;  
    int port_number;  
{  
    struct sockaddr_in *address;  
  
    /* Name socket using incoming port number */  
    address = inet_socketaddr (NULL, port_number);  
    if (bind(socket_fd, address, sizeof (struct sockaddr_in)) == -1)  
    {  
        syserr("binding stream socket");  
        exit(0);  
    }  
}  
  
/*****/  
/** **/  
/** This primitive is to be used by the server only, just after the "bind **/  
/** socket" one, and in relation with "client_receive_data". **/  
/** The server sets up a queue for incoming connection requests using **/  
/** "listen()". Then it initiates a connection using "accept". The **/  
/** "accept" call returns a new file descriptor which is used to send a **/  
/** message to the client. **/  
/** **/  
/*****/  
  
void server_send_data(socket_fd, buffer)  
    int socket_fd;  
    char buffer[1024];  
{  
    int msgsock;  
  
    /* Start accepting connection */  
    listen(socket_fd, 5);  
  
    msgsock = accept(socket_fd, (struct sockaddr *)0, (int *)0);  
    if (msgsock == -1)  
        syserr("accept");  
  
    /* write the message for the client */  
    else if (write(msgsock, buffer, 1024) == -1)  
        syserr("writing on stream socket");  
    if (close(msgsock) == -1)  
        syserr("close socket");  
}  
  
/*****/  
/** **/  
/** This primitive is used by the client only, in relation with "server_ **/  
/** send_data". **/  
/** First, it looks up to the server socket UNIX address it wishes to **/  
/** connect by using "get_UNIX_addr" call. Then it requests a connection **/  
/** using "connect". If the connection is established by the server, the **/  
/** **/>
```

INTERNET_socket_primitives.c

```
/** two processes can communicate. */
/** Here, the client reads a message sent by the server. */
/** */
/*****/

void client_receive_data(socket_fd, port_number)
int socket_fd;
int port_number;
{
    char          buf[1024];
    struct sockaddr_in *server;
    int          rval;

    /* Connect socket using inet_sockaddr call */
    server = (struct sockaddr_in *)inet_socketaddr(NULL, port_number);
    if (connect(socket_fd, (struct sockaddr *)server, sizeof (*server)) == -1)
    {
        delete_socket(socket_fd);
        syserr("connecting stream socket");
        exit(0);
    }

    /* read the message sent by the server */
    do
    {
        if ((rval = read(socket_fd, buf, 1024)) == -1)
            perror("reading stream message");
        else
            printf("message read : %s\n", buf);
    }
    while (rval > 0);
}

/*****/
/** */
/** This primitive is to be used by the server only, just after the "bind */
/** socket" one, and in relation with "client_send_data". */
/** The server sets up a queue for incoming connection requests using */
/** "listen()". Then it initiates a connection using "accept". The */
/** "accept" call returns a new file descriptor which is use to receive a */
/** message from the client. */
/** */
/*****/

void server_receive_data(socket_fd)
int socket_fd;
{
    int msgsock;
    char buf[1024];
    int  rval;

    /* Start accepting connection */
    listen(socket_fd, 5);

    msgsock = accept(socket_fd, (struct sockaddr *)0, (int *)0);
    if (msgsock == -1)
        syserr("accept");

    /* read the message sent by the server */
    do
    {
```

INTERNET_socket_primitives.c

```
    if ((rval = read(msgsock, buf, 1024)) == -1)
        perror("readind stream message");
    else
        printf("message read : %s\n", buf);
}
while (rval > 0);
if (close(msgsock) == -1)
    syserr("close socket");
}

/*****
**
** This primitive is used by the client only, in relation with "server_
** send_data".
** First, it looks up to the server socket UNIX address it wishes to
** connect by using "get_UNIX_addr" call. Then it requests a connection
** using "connect". If the connection is established by the server, the
** two processes can communicate.
** Here, the client send a message to the server.
**
**
*****/

void client_send_data(socket_fd, port_number, buffer)
    int socket_fd;
    int port_number;
    char buffer[1024];
{
    struct sockaddr_in *server;

    /* Connect socket using inet_sockaddr call */
    server = (struct sockaddr_in *)inet_socketaddr(NULL, port_number);
    if (connect(socket_fd, (struct sockaddr *)server, sizeof (*server)) == -1)
    {
        delete_socket(socket_fd);
        syserr("connecting stream socket");
        exit(0);
    }

    /* write the message for the client */
    else if (write(socket_fd, buffer, 1024) == -1)
        syserr("writing on stream socket");
}
}
```

INTERNET_socket_example1.c

```

/*****
/*****
/**
/** This is a example program using sockets within the INTERNET domain. **/
/** A parent process sends a message to a child process. **/
/** - The parent is the server: it creates a socket, binds a name to it, **/
/**     listens for a connection and accepts it to write the message. **/
/** - The child is the client: it creates a socket, asks to connect it **/
/**     to the parent's one, and reads the message **/
/** **/
/*****
/*****

#include "E_socket_primitives.c"

/*****
/**
/** This message may be adjusted with the appliation requirements **/
/** **/
/*****

#define DATA "message sent to the client-child" /* data exchanged */

int     port_server = 0; /* port number */

main()
{
    int     parent, child; /* processes ID (not necessary) */
    int     server_fd, client_fd; /* socket file descriptors */
    char    buffer[1024];

    /* copy the message in "buffer" */
    strcpy(buffer, DATA);

    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* we intialize the port address with the parent process ID */
    port_server = parent;

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using port_server */
    bind_socket(server_fd, port_server);

    /*****
    /*****
    /**
    /** fork creates a new process with almost exact copies of instruction, **/
    /** user-data, and system-data segments. After fork return, both parent **/
    /** and child processes receive the return. The return value is **/
    /** different, however, which is crucial, because it allows their **/
    /** subsequent actions to differ: **/
    /** - the child receives a 0 return value; **/
    /** - the parent receives the process-ID of the child. **/
    /** **/
    /*****
    /*****

```

INTERNET_socket_example1.c

```
/*  
*/  
if ((child = fork()) == -1)  
{  
    perror("fork");  
    exit(0);  
}  
else  
{  
    if (child != 0)          /* this the parent, the return value is <> 0 */  
    {  
        /* Start accepting connection and write message to the client */  
        server_send_data(server_fd, buffer);  
        sleep(1); /* not necessary, allow a good print of the message result */  
  
        /* delete the server socket */  
        delete_socket(server_fd);  
    }  
  
    if (child == 0)          /* this is the child; the return value is = 0 */  
    {  
        child = getpid();  
        printf("child id = %d\n\n", child);  
  
        /* Create a socket */  
        client_fd = create_socket();  
  
        /* Read the message after requesting for connection to server socket */  
        client_receive_data(client_fd, port_server);  
  
        /* delete the socket */  
        delete_socket(client_fd);  
    }  
}  
}
```

INTERNET_socket_example2.c

```

/*****
/*****
**
** This is an example program using sockets within the INTERNET domain. **
** A parent process receives a message sent by its child process. **
** - The child is the server: it creates a socket, binds a name to it, **
**   listens for a connection and accepts it to write the message. **
** - The parent is the client: it creates a socket, asks to connect it **
**   to the child's one, and read the message **
**
/*****
/*****

#include "E_socket_primitives.c"

/*****
/*****
**
** This message may be adjusted with the appliation requirements **
**
/*****
/*****

#define DATA "message sent to the client-parent" /* data exchanged */

int      port_server = 0;                          /* port number */

main()
{
    int      parent, child;          /* process ID (not necessary) */
    int      server_fd, client_fd; /* socket file descriptors */
    char     buffer[1024];

    /* copy the message in "buffer" */
    strcpy(buffer, DATA);

    /* we intialize the port address with the parent process ID */
    port_server = getpid();

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*****
    /*****

    if ((child = fork()) == -1)
    {
        perror("fork");
        exit(0);
    }
}

```

INTERNET_socket_example2.c

```
else
{
if (child != 0)          /* this the parent, the return value is <> 0 */
{
parent = getpid();
printf("parent id = %d\n\n", parent);

/* Create a socket */
client_fd = create_socket();

/*****
/* The following instruction "sleep" is here to insure that the */
/* server (here the child) has had the time to create its own */
/* socket and is ready to accept connection BEFORE the client */
/* attempts to make a request. */
*****/
sleep(1); /* not necessary, allow a good print of the message result */

/* Read the message after requesting for connection to server socket */
client_receive_data(client_fd, port_server);

/* delete the socket */
delete_socket(client_fd);
}

if (child == 0)          /* this is the child; the return value is = 0 */
{
child = getpid();
printf("child id = %d\n\n", child);

/* Create a socket */
server_fd = create_socket();

/* Name socket using port_server */
bind_socket(server_fd, port_server);

/* Start accepting connection and write the message to the client */
server_send_data(server_fd, buffer);

/* delete the server socket */
delete_socket(server_fd);
}
}
}
```

INTERNET_socket_example3.c

```

/*****
/*****
/**
/** This is a example program using sockets within the INTERNET domain.
/** A child process sends a message to its parent process.
/** - The parent is the server: it creates a socket, binds a name to it,
/** listens for a connection and accepts it to receive the message.
/** - The child is the client: it creates a socket, asks to connect it
/** to the parent's one, and sends the message.
/**
/**
/*****
/*****

#include "E_socket_primitives.c"

/*****
/**
/** This message may be adjusted with the appliation requirements
/**
/*****

#define DATA "message sent to the server-parent" /* data exchanged */

int port_server = 0; /* port number */

main()
{
    int parent, child; /* processes ID (not necessary) */
    int server_fd, client_fd; /* socket file descriptors */
    char buffer[1024];

    /* copy the message in "buffer" */
    strcpy(buffer, DATA);

    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* we intialize the port address with the parent process ID */
    port_server = parent;

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using port_server */
    bind_socket(server_fd, port_server);

    /*****
    /*****
    /**
    /** fork creates a new process with almost exact copies of instruction,
    /** user-data, and system-data segments. After fork return, both parent
    /** and child processes receive the return. The return value is
    /** different, however, which is crucial, because it allows their
    /** subsequent actions to differ:
    /** - the child receives a 0 return value;
    /** - the parent receives the process-ID of the child.
    /**
    /**

```

INTERNET_socket_example3.c

```
/*  
*/  
if ((child = fork()) == -1)  
{  
    perror("fork");  
    exit(0);  
}  
else  
{  
    if (child != 0)          /* this the parent, the return value is <> 0 */  
    {  
        /* Start accepting connection and read the message from the client */  
        server_receive_data(server_fd);  
        sleep(1);  
  
        /* delete the server socket */  
        delete_socket(server_fd);  
    }  
  
    if (child == 0)          /* this is the child; the return value is = 0 */  
    {  
        child = getpid();  
        printf("child id = %d\n\n", child);  
  
        /* Create a socket */  
        client_fd = create_socket();  
  
        /* Write the message after requesting for connection to server socket */  
        client_send_data(client_fd, port_server, buffer);  
  
        /* delete the socket */  
        delete_socket(client_fd);  
    }  
}  
}
```

INTERNET_socket_example4.c

```

/*****
/*****
**
** This is an example program using sockets within the INTERNET domain. **
** A parent process sends a message to its child process. **
** - The child is the server: it creates a socket, binds a name to it, **
**   listens for a connection and accepts it to receive the message. **
** - The parent is the client: it creates a socket, asks to connect it **
**   to the child's one, and sends the message. **
**
**
/*****
/*****

#include "E_socket_primitives.c"

/*****
/*****
**
** This message may be adjusted with the appliation requirements **
**
**
/*****
/*****

#define DATA "message sent to the server_child" /* data exchanged */

int      port_server = 0; /* port number */

main()
{
    int      parent, child; /* process ID (not necessary) */
    int      server_fd, client_fd; /* socket file descriptors */
    char     buffer[1024];

    /* copy the message in "buffer" */
    strcpy(buffer, DATA);

    /* we intialize the port address with the parent process ID */
    port_server = getpid();

    /*****
    /*****
    /*
    /* fork creates a new process with almost exact copies of instruction,
    /* user-data, and system-data segments. After fork return, both parent
    /* and child processes receive the return. The return value is
    /* different, however, which is crucial, because it allows their
    /* subsequent actions to differ:
    /* - the child receives a 0 return value;
    /* - the parent receives the process-ID of the child.
    /*
    /*
    /*****
    /*****

    if ((child = fork()) == -1)
    {
        perror("fork");
        exit(0);
    }
}
```

INTERNET_socket_example4.c

```
else
{
  if (child != 0)          /* this the parent, the return value is <> 0 */
  {
    parent = getpid();
    printf("parent id = %d\n\n", parent);

    /* Create a socket */
    client_fd = create_socket();

    /******
    /* The following instruction "sleep" is here to insure that the  */
    /* server (here the child) has had the time to create its own  */
    /* socket and is ready to accept connection BEFORE the client  */
    /* attempts to make a request.  */
    /******
    sleep(1);

    /* Write the message after requesting for connection to server socket */
    client_send_data(client_fd, port_server, buffer);
    sleep(1); /* not necessary, allow a good print of the message result */

    /* delete the socket */
    delete_socket(client_fd);
  }

  if (child == 0)          /* this is the child; the return value is = 0 */
  {
    child = getpid();
    printf("child id = %d\n\n", child);

    /* Create a socket */
    server_fd = create_socket();

    /* Name socket using port_server */
    bind_socket(server_fd, port_server);

    /* Start accepting connection and read message sent by the client */
    server_receive_data(server_fd);

    /* delete the server socket */
    delete_socket(server_fd);
  }
}
}
```

REFERENCES

- [1] H-C Kim, J-L Gaudiot, R.E. Stumberger. *Data-Driven Execution of Logic Languages*. PhD Thesis (in preparation), Department of EE-Systems, University of Southern California.
- [2] Lindstrom, Gary. OR-Parallelism on Applicative Architectures. In *Second International Logic Programming Conference*, pages 159-170, 1984.
- [3] J.S. Conery. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. In *1987 Symposium on Logic Programming*, pages 159-170. IEEE Computer Society Press, August 1984.
- [4] Peter M. Kogge. The Warren Abstract Machine. In *The Architecture of Symbolic Computers*, pages 486-525. McGraw-Hill, 1991.
- [5] Peter M. Kogge. Parallel Inference Engines. In *The Architecture of Symbolic Computers*, pages 657-706. McGraw-Hill, 1991.
- [6] Andrew W. Appel. A Runtime System. In *Lisp and Symbolic Computation: An International Journal*. 1990 Kluwer Academic Publishers.
- [7] L.V. Kale. The Chare Kernel Parallel Programming Language and System. In *1990 International Conference on Parallel Processing*.
- [8] Chien Chen. *Scheduling Heuristics and Runtime Data Structures for the Parallel Execution of Prolog Programs*. PhD Dissertation, University of California Berkeley.
- [9] H-C Kim, J-L Gaudiot, R.E. Stumberger. *Intermediate Representation and Translation Scheme for the Non-deterministic Data-Flow Parallel Execution Model*. PhD Thesis (in preparation), Department of EE-Systems, University of Southern California.
- [10] H-C Kim, J-L Gaudiot, R.E. Stumberger. *A Novel Binding Environment for Non-single Address Space*. PhD Thesis (in preparation), Department of EE-Systems, University of Southern California.
- [11] Marc J. Rochkind. *Advanced UNIX Programming*, Prentice-Hall Software Series.
- [12] *A Socket-Based Interprocess Communications Tutorial*. Department of EE-Systems, University of Southern California.