273721

Distributed Prosoft: Management of Tools and Memory

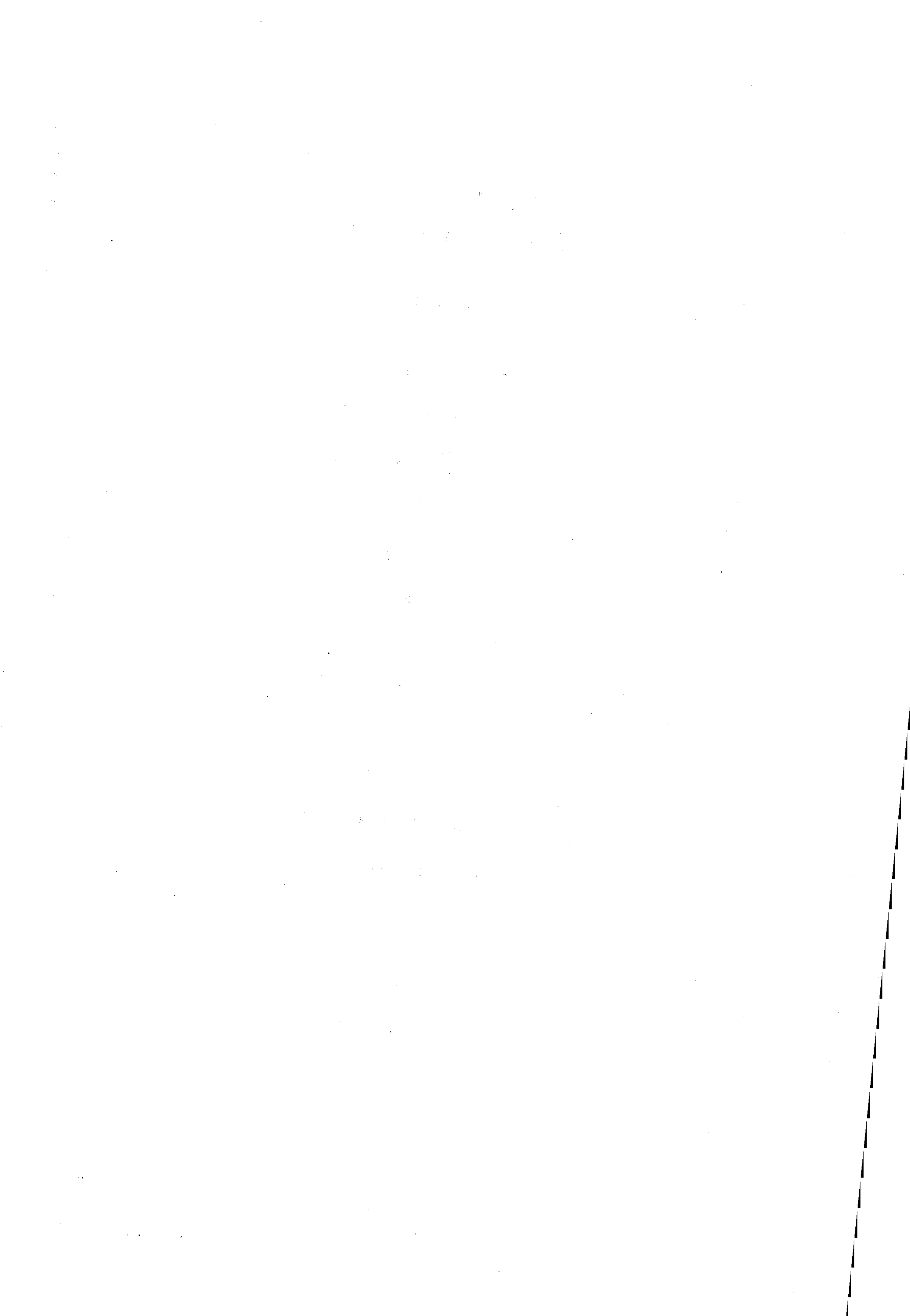
Report on a work developed at the
Institut für Informatik
Universität Stuttgart / Germany
from December 1995 to March 1996

Lisandro Zambenedetti Granville
Luciano Paschoal Gaspary

UFRGS
SABi

05230594

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA



Distributed Prosoft: Management of Tools and Memory

Report on a work developed at the
Institut für Informatik
Universität Stuttgart / Germany
from December 1995 to March 1996

Lisandro Zambenedetti Granville
Luciano Paschoal Gaspary

Engenharia de Software: SBU
Engenharia: Software
PROSOFT
Desenvolvimento: Software
CNPq 1.03.03.00-6
INSTITUTO DE INFORMÁTICA
BIBLIOTECA
N.º CHAMADA FL 129
N.º REG.: 37736
05/09/00
ORIGEM: 1 088 00 R$ 10,00
FUNDO: II
FORN.: II / LISANDRO Z. GRANVILLE

The authors thank Prof. Daltro José Nunes for the support and incentive in the development of this environment. A special thank to Heribert Schlebbe for the great help in all the steps of the implementation of the system.

Stuttgart, March 1996

Contents

1 **Introduction** 1

2 **Conventional Prosoft** 1

3 **Transition to Distributed Prosoft** 2

4 **Distributed Prosoft** 4

4.1 Design Decisions 4

4.2 ICS and Communication between ATOs 5

4.2.1 Client Side of a Remote ICS Call 5

4.2.2 Server side of a Remote ICS call 6

4.3 Control and Management of the Prosoft Server Processes over the Network 7

4.3.1 Configuration of the Servers 7

4.3.2 Management of the Servers 8

4.3.3 User Session 8

4.4 ICS Call and Distributed Memory Management 10

4.4.1 Parameters of the ICS Remote Call 10

4.4.2 Distributed Memory Management 12

4.4.3 Minimization of Data Transport 13

4.4.4 Garbage Collection 17

5 **Suggestions** 18

5.1 Communication between Users 19

5.2 Reliability 19

6 **Conclusions** 20

Bibliography 21

Appendix 22

1 Introduction

PROSOFT – a Software Development Environment – has been developed at the Instituto de Informática, UFRGS (Universidade Federal do Rio Grande do Sul), under the direction of Prof. Dr. Daltro José Nunes, supported by the Institut für Informatik, Universität Stuttgart.[1]

The distributed version of the system – Distributed PROSOFT – is conceived and implemented to satisfy new requirements of the environment and to provide more flexibility. Its development is based on techniques which will be elucidated in the several sections of this report.

Section 2 covers the concepts of the conventional PROSOFT showing the entities that constitute the original system. In section 3 the new features are shown required in the system: distribution of ATOs on the network, better usage of resources and more efficient configurability mechanisms. The main concepts of the distributed version of the system, the process and memory management of the ATO servers, are explained in section 4. Suggestions of future implementations based on the new environment are discussed in section 5.

2 Conventional Prosoft

The main goal of the mentioned PROSOFT project is to create a prototype for a generic computational environment to develop software. The conventional PROSOFT is an extensible software system where new components are added forming bases for the creation of other tools. A tool in PROSOFT is called ATO – Ambiente de Tratamento de Objetos – which consists of a class representing its data structure and a set of methods that manipulate instantiations of this class. Each instance of an ATO class is called object.

Each object interacts with others by a common communication interface ICS – Interface de Comunicação do Sistema. It takes a set of objects as parameters and proceeds the call of an operation of another ATO. The result is returned to the calling object.

In the conventional PROSOFT system all ATOs are bound in only one executable program. They therefore always belong to the same execution process running on the same machine. Hence, the ICS always executes local calls, which have access

[1]The current work was developed within the scope of a person-to-person cooperation between the PROSOFT working group of the Instituto de Informática,UFRGS, Porto Alegre (Prof. Dr. Daltro José Nunes) and the Institut für Informatik, Universität Stuttgart, sections Dialogsysteme (Prof. Dr. Rul Gunzenhäuser) and Betriebssoftware (Prof. Dr. Klaus Lagally). Expenses for traveling and accommodation in Germany were payed by the Brazilian Research Council (CNPq).

to a set of global data of the PROSOFT system. Furthermore, a conventional PROSOFT session does not possess any information concerning the active sessions of other users. Therefore communication among them is impossible. Figure 1 shows the localization of ATOs in the address space of a single executable program and their interaction via ICS.

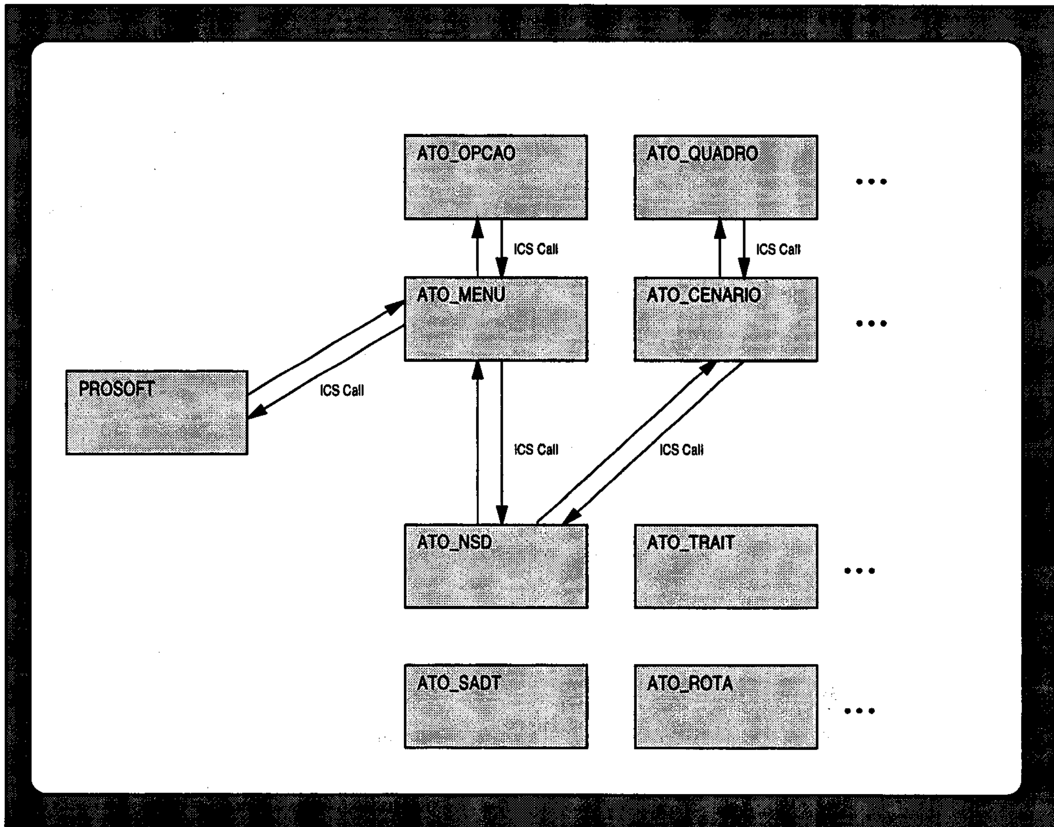


Figure 1: Conventional PROSOFT System

A more detailed description of conventional PROSOFT system, its development environment and paradigm can be found in [Nun92] [Nun93].

3 Transition to Distributed Prosoft

With the growing number of developed ATOs the conventional PROSOFT system started to create some problems taking directly effect in the development of new tools. The creation and maintenance of ATOs require a lot of time to be linked, since all ATOs are bound together in only one executable program.

Furthermore, each PROSOFT user, even if interested only in a subset of ATOs is forced to load and run the complete set. Evidently memory resources were wasted, once the tools could not be shared between several users.

On the other hand, new ideas and concepts were developed and new necessities appeared. For instance, the possibility of having cooperative work in the future also has contributed to a new conception of the system.

Hence, the concept of Distributed PROSOFT was created. The first work was made by Heribert Schlebbe during his working stay in Brazil in 1994 [Sch94]. The new system should allow several tools to be autonomously distributed over the network and capable to be attended to different users simultaneously. This model covers the new requirements and is supposed to be a base for future implementations.

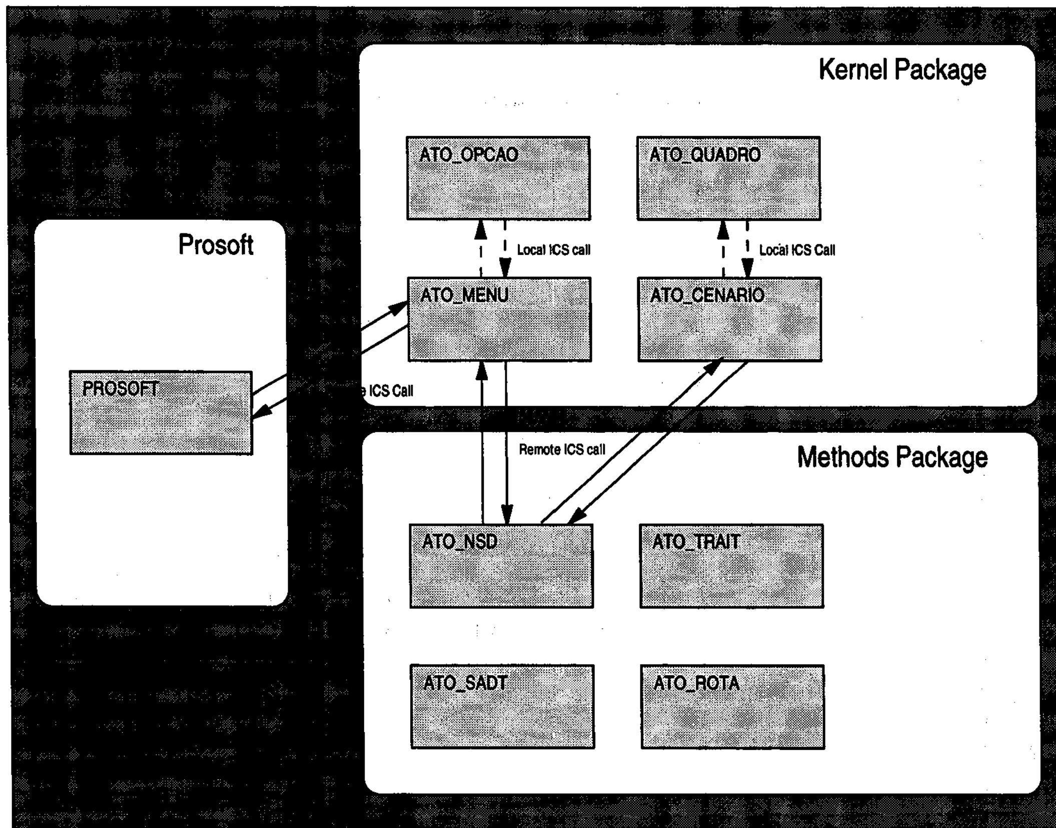


Figure 2: Distributed PROSOFT

The environment consists of ATOs spread over the network and joined in packages – so-called ATO packages. Each package is conceived to be a server process which is capable, hence, to dispatch requests of users or other packages, i.e. to execute the requested operation of a respective ATO belonging to the package. If the requested service resides in the same package of the calling ATO it is

meant to be a local call, otherwise it is supposed to be a remote one. (see figure 2 on page 3). The processing of these ICS calls is completely transparent to the implementor of an ATO (so-called atista), since it is implemented at a lower level.

4 Distributed Prosoft

4.1 Design Decisions

The Distributed PROSOFT system was implemented according to a client/server architecture. This model is very appropriate to the development of this environment, since the ATOs are completely encapsulated and their communication is done exclusively by means of a common interface – the ICS. As a result of that, the ATOs behave like servers, receiving requests from client processes via ICS. Furthermore, the classes of the respective ATOs represent data structures which can be allocated at the start-up time of the service, i.e. before serving any requests of the clients. The access to these classes does not require any synchronization when serving asynchronous requests of clients, because they will be accessed just read-only. An instance of the communication of ATOs via ICS can be seen in figure 3.

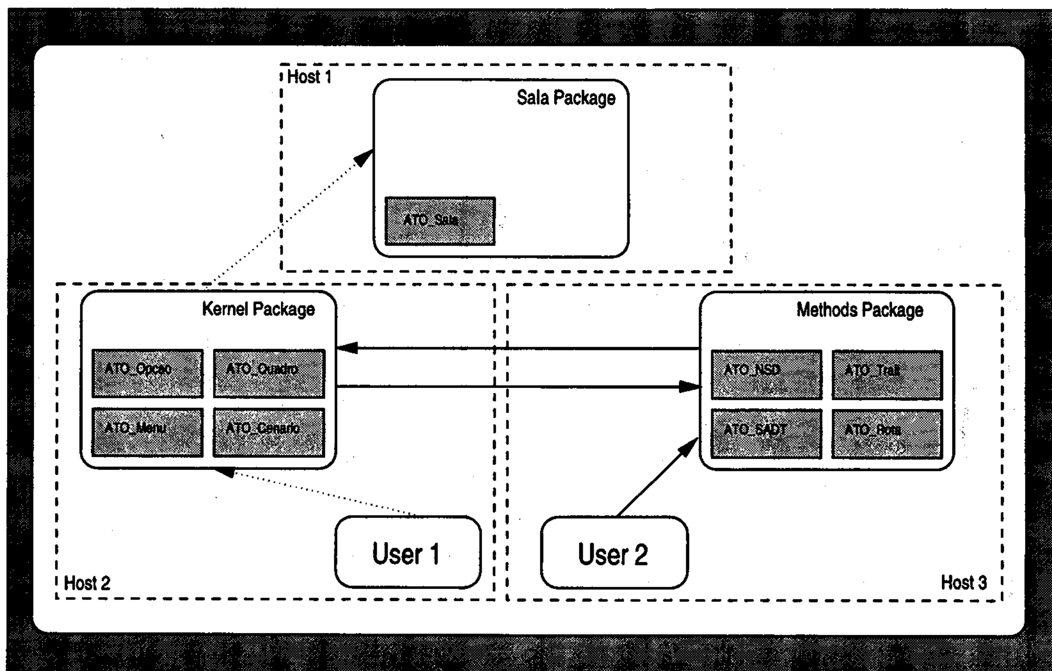


Figure 3: Communication between ATOs via Remote ICS

To provide this functionality of ICS the environment has to be remodeled by using the following techniques:

- **Remote Procedure Call** (RPC): The distribution of ATOs over the network requires the use of efficient mechanisms which allow communication between remote processes to exchange objects between packages of ATOs. More details about the RPC mechanism can be found in [Blo92].
- **Multi-threaded Servers**: Since the ATOs are supposed to be sharable among several users, an implementation of servers is necessary to handle asynchronous incoming calls from different clients simultaneously. Hence, multi-threaded servers are developed which are capable to execute each incoming ICS request by its own thread of control. More information concerning multi-thread programming can be found in [Pow91] and [Sun95].

4.2 ICS and Communication between ATOs

An ICS call in Distributed PROSOFT has the same semantic behaviour like a conventional call. The parameters of an operation to be executed are sent to the server which dispatches the incoming request. The calling process blocks, until the server returns the answer of the required service. An ICS call in Distributed PROSOFT, hence, is executed as a co-routine. For this reason there is no need of any syncronization mechanisms between the processes related to these calls (see figure 4). Indirect recursive calls to the same server are also permitted. More details about the client/server specifications of the ICS can be found in [Sch94] and in the Appendix.

4.2.1 Client Side of a Remote ICS Call

All ATO clients – i.e. the PROSOFT main program as well as most of the ATO servers (they behave like clients when requesting services of other ATO servers) – have an individual ICS implementation distinguishing between ATOs that are called locally and those that have to be called remote. Local ATO calls are executed in the same way as in the conventional PROSOFT, while the remote ATO calls need to proceed the following steps:

- discover the network location (RPC address and path of server program) where the ATO server of the needed operation resides;
- create a TCP connection to the server of the requested ATO package;
- prepare the parameter objects of the ICS call by encoding them in an appropriate way for transport across the network to the server package;

- call the routine using the transport descriptor;
- receive the resulting object and decode it for the client address space;
- close the TCP connection to the server.

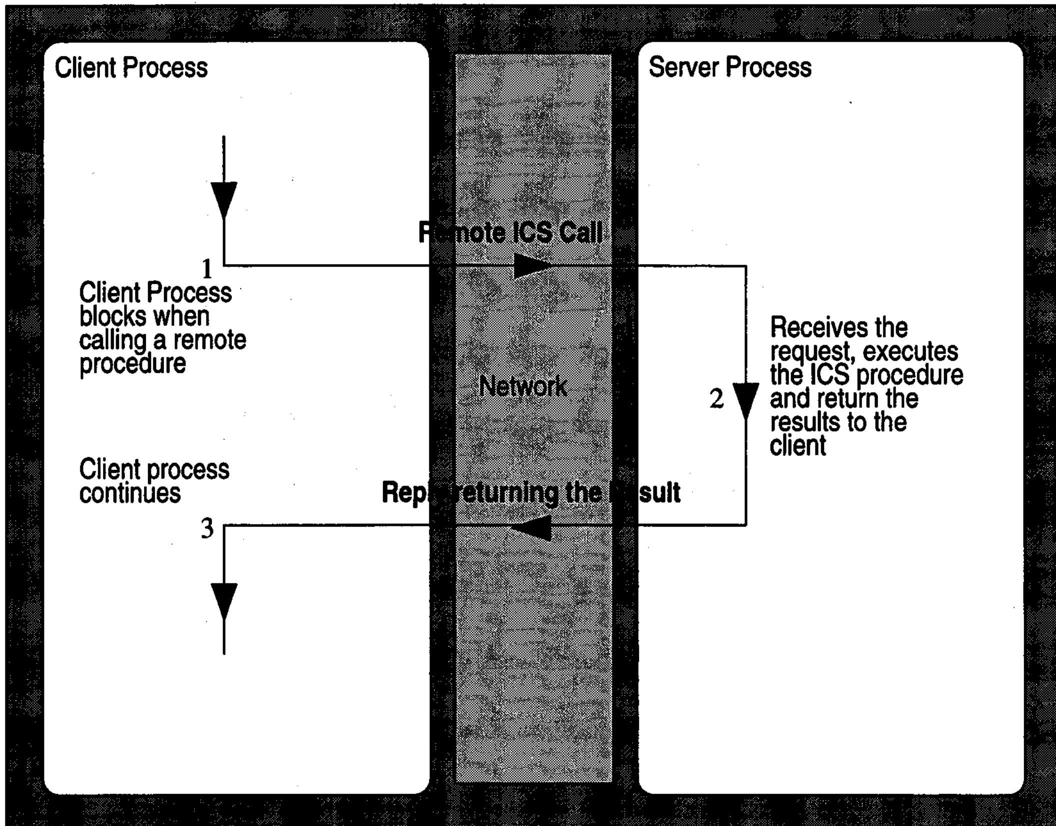


Figure 4: Remote ICS call

4.2.2 Server side of a Remote ICS call

When activated, the main thread of the server creates a handle for TCP connection to receive requests and then enters the server loop. The current implementation uses the multi-threaded server loop provided by the Solaris operating system of Sun Microsystems. This routine is reliable, since it has been totally implemented using multi-thread safe libraries.

After the connection with the server is established, a dispatching routine is executed to serve the request by executing the following steps:

- receive the parameters of the service;

- rebuild the PROSOFT objects by using the unique identification existing in each node;
- call the local ICS function;
- prepare the result object for transport across the network;
- send the result object to the client.

4.3 Control and Management of the Prosoft Server Processes over the Network

Since the ATOs – due to their distribution over the network – will execute autonomously, there is a need of providing a manager process which is able to control these servers and allows the configuration of them. The PROSOFT Manager was created for this purpose.

4.3.1 Configuration of the Servers

The main point in configuring the servers on the network consists in the task to inform the PROSOFT Manager about the ATOs incorporated in each package as well as their localization on the network. For this purpose, a configuration file was introduced.

The correct configuration of ATOs is fundamental for an appropriate and efficient performance of Distributed PROSOFT. It must be considered that some ATOs are intrinsically related to each other. It is recommended to locate them, for this reason, in the same package, otherwise there would be an intense network traffic.

For instance, the ATOs `cenário` and `quadro` have a strong relation to each other. If these ATOs were located in different packages, their interaction would provoke a heavy network traffic. Hence, the efficiency of the environment depends on a good customization of these ATOs and the PROSOFT administrator should be aware of this.

The configuration file also acts as a base of information for a utility which automatically generates, according to this file, the `makefiles` and – by the RPC protocol compiler `rpcgen` – the source code of the ICS routines. The source code includes the interface of the client side for local and remote ICS calls, the transport functions for passing argument/result objects to/from a remote ICS procedure and the server routine.

4.3.2 Management of the Servers

The PROSOFT Manager is a server process which provides the management of the distributed system in total, controlling the users who are interacting with the system as well as the ATO server processes that are running distributed on the network (see figure 5). It is able to serve the following requests:

- starts and shuts down a PROSOFT server according to the customization of the server type (multi-client, single-client or local-client) on a specified host. During the start-up time of the process, the classes belonging to the ATOs of that servers are created, so that they later can be accessed read-only by the client processes;
- initializes user specific information on the server side:
  - creates a memory management structure of the user;
  - allocates a package identifier which is needed to control the optimized memory update strategy when calling other servers via Remote ICS (see section 4.4.3);
  - establishes an RPC connection to the graphical display of the user (see section 4.3.3);
- frees user specific data on the server side when a user has finished his session:
  - destroys the memory storage associated to that user;
  - closes the RPC connection to the user display;
- provides the RPC address (program number, version number and host name) of an ATO server to a client process. For optimization the RPC addresses of active servers are cached on the client side;
- keeps information about the users which are currently active (e.g. user identification, user name, current working directory, etc);
- keeps information about the ATO servers which are active over the network.

4.3.3 User Session

Every user of the Distributed PROSOFT activates his own set of processes, as described below:

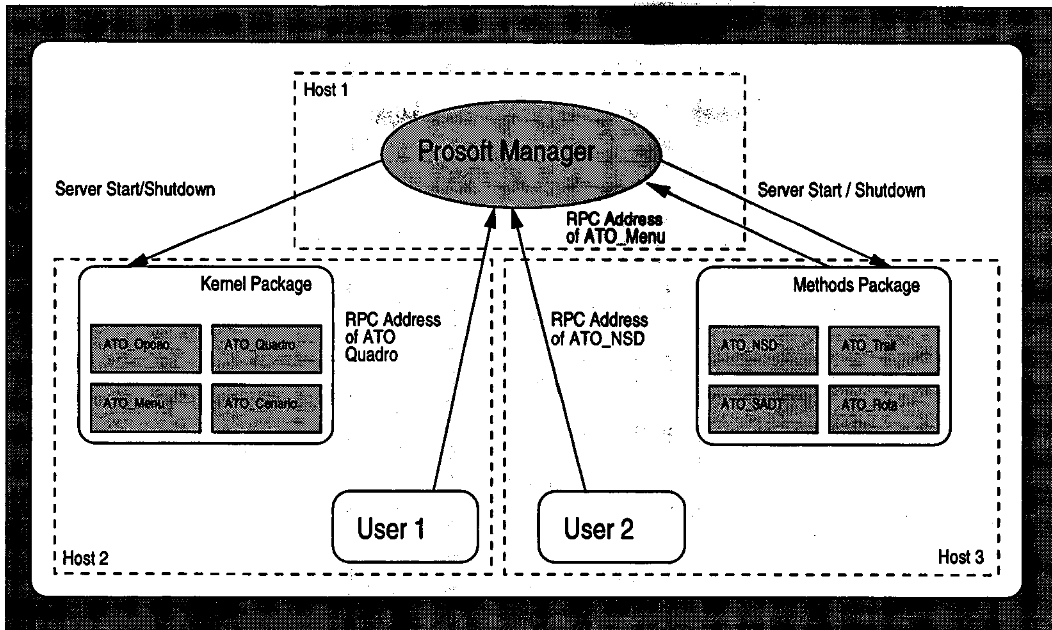


Figure 5: PROSOFT Manager and the requests

The User Main Process

A user activates the Distributed PROSOFT by starting a program named `prosoft`. This process is responsable for registering the user with the PROSOFT Manager – i.e. getting a unique user identification for his session – and for activating other server processes on demand.

The Graphic Server Process

Once a user has opened a PROSOFT session and an identification has been assigned to him by the PROSOFT Manager, a server is started which is responsible for receiving and displaying graphical commands from all ATOs the user will activate during his session. This process is called Display Manager.

Thus, the Display Manager implements an RPC server with an RPC address which is built uniquely by the user identification. This unique address assignment warrants that all outputs of graphical procedure calls (made be the ATO servers during the session) will be directed to the display of the user. The Display Manager needs to be single-threaded due to the multi-thread unsafeness of the actual X-Windows system. Since ICS requests are executed like co-routines, all graphical output sent to the Display Manager will be also processed sequentially. Therefore no synchronization mechanism is necessary.

To allow server processes to direct the graphical output to the right Display Manager, the user identification must be sent to the server as an additional

DE INFORMÁTICA
BIBLIOTECA

argument of the ICS mechanism implemented at lower layer. This information is saved in a thread-specific variable of the ICS dispatching thread, so that every graphical RPC call can get access to the right Display Manager.

The Directory Server Process

The ATO Directory does not follow the PROSOFT paradigm. Moreover, this ATO has a set of global variables that do not allow the standard incorporation of the ATO Directory in the Distributed PROSOFT system.

A main problem lies in the fact that the ATO Directory reads/writes PROSOFT objects from/to local disks which are mounted on the workstation of the user. A remote running ATO Directory, however, would have no access to these data, because the directory structure visible to that ATO would be that of the PROSOFT Manager.

To solve this problem the ATO Directory should be customized, for each user session, as an individual server process running on the local host of the user. In this way, the ATO works properly, because it now reflects the directory informations of the user who has activated the ATO process.

4.4 ICS Call and Distributed Memory Management

As stated before, the implementation of Distributed PROSOFT is mainly based on turning the ICS interface into a communication interface between ATOs which are interacting across the network. By providing an additional software layer in the ICS interface which is capable to handle client/server communication, the PROSOFT system can be transformed into the distributed system without changes of the source code of the ATOs and the higher level components of the conventional version.

4.4.1 Parameters of the ICS Remote Call

The standard ICS routine has a compound of three parameters: the ATO name, the function to be executed and a list of PROSOFT objects to be passed to the function.

In the lower level implementation of the remote ICS routine, however, two additional parameters are required: the user identifier and the return package identifier.

The main problem here is the fact that an object when sent to the address space of the remote side might be a sub-object of another one in the client address space. Since the RPC mechanism does not provide any shared memory facilities – and therefore inclusions of objects are not recognized – the use of RPC might

provoke undesired copies of objects. Therefore additional mechanisms to treat the parameter and result objects are necessary.

For example, when a remote server procedure is called with a parameter representing a **cenário** and another one representing a **quadro** object – being the **quadro** object a sub-object of the first one – the server will rebuild both objects – the **cenário** and just so the **quadro** object – independently ignoring the object inclusion. In this case there exist two copies of the same **quadro** object on the server side, which apparently do not work properly (see figure 6 for illustration).

First proposal: Identification of objects in different address spaces

In a first approach to solve this problem all possible inclusions of the parameter objects were analyzed, before executing an RPC, to find out which object was sub-object of another one. Instead of sending the included object just the path to reach this sub-object from the parent one is sent. By this method an included object can be identified within another object in the address space on the remote side, so that improper copies are avoided. The same happens with the result object: Because a result object of an ICS call either is a new object or a modified parameter object, only the modified parts of a certain object – inclusive the paths to reach them – were returned to the client address space, in order to update the original object.

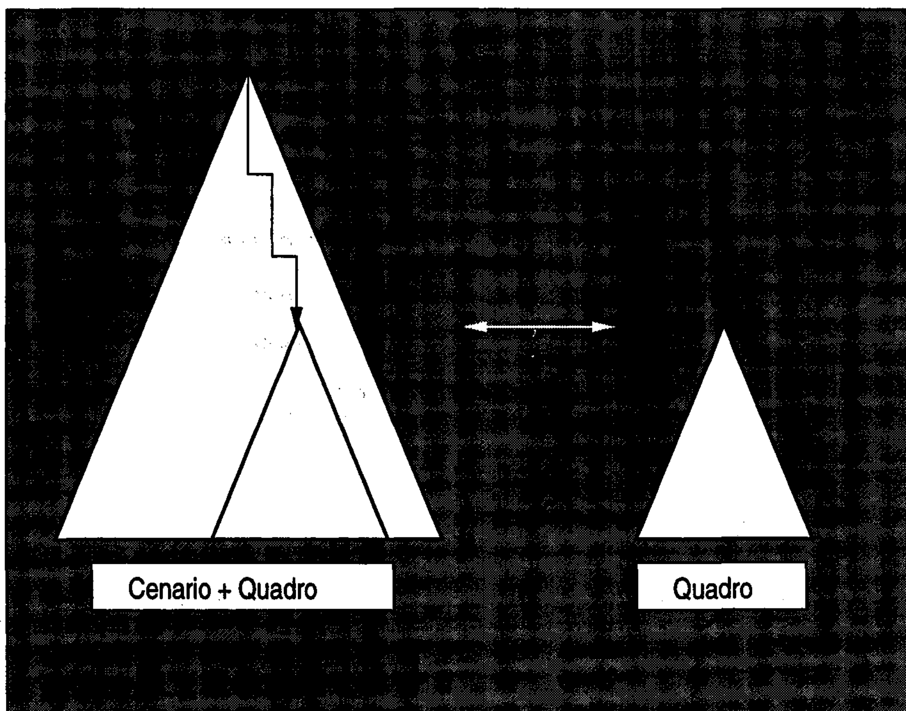


Figure 6: Reaching a sub-object

However, this solution turned out to be very inefficient, because in some cases it is hard to analyze all modifications done in the result object. A critical case, for instance, is the exchange of the places of two sub-objects within an object – a case which requires a great analytical effort to recognize it; this proposal was discarded.

Second proposal: Identification of nodes in different address spaces

Instead of identifying parameters and possible inclosures in a remote address space on the base of PROSOFT objects, the nodes themselves – which every object is constructed of – are regarded to find out a more effective solution.

When created, every node is assigned a unique identification which depends on the ATO package identifier. This node identifier can be used to recognize the same node in different address spaces. Furthermore, in order to form a PROSOFT object which should be independent of address spaces, the memory pointers – which are linking nodes together to build an object – should be replaced by their corresponding node identifiers. This object representation can be used to rebuilt a homologous object in any desired address space.

If the remote ICS function would transport only the root identifications and a unique sequence of address space independent nodes of the parameters, the objects could easily be rebuilt – with regard to all their object inclusions – in the address space of the remote side.

The transport of nodes via net can be minimized, if both sides (client and server) maintain a storage to cache all nodes which ever have been exchanged, and further, if there exists a mechanism to control which nodes are in a consistent state on both sides.

This concept of a minimum data exchange requires the implementation of a distributed memory management mechanism which is described in the following section.

4.4.2 Distributed Memory Management

There are several options to implement a distributed memory.

- *Centralized memory*: The memory resides at exactly one location on the network; a manager process administers the read/write requests of remote processes to that memory.

- *Floating memory*: The memory may be distributed over different locations (processes) on the network; a mechanism is required to control which process is actually the owner of the memory object to be accessed.

- *Replicated memory*: Every process caches parts of the common memory which must not necessarily be in a consistent state; a mechanism is needed to control which parts of the remote memory have to be updated when doing a request to a remote process, and which are the parts of its own memory to be updated upon receipt of the result.

The solution implemented uses the last method. The memory replication method seems to be the ideal option to minimize the network data flow.

To allow object nodes to be efficiently accessed in the memory, when objects of the ICS interface have to be codified or decodified in the actual address space, a *hash table* is implemented. To keep insertions of nodes into and deletions from the table transparent to the implementor of ATOs, all primitive functions of the PROSOFT system that create, destroy and change nodes were modified in order to treat these operations automatically.

To implement a distributed memory manager, two fields have to be added to the structure of a conventional PROSOFT node:

- *Node identifier*: On the one hand this field is used – as already mentionend above – as a substitution for memory pointers when building object representations which must not be dependent on an address space, on the other hand it serves as a hash index to the memory table;
- *Set of package identifiers*: This field contains the identifiers of those server packages that already have stored the most recent version of this node in their memory. Section 4.4.3 discusses this field in detail.

When a user – identified by his identification – for the first time requests a service of a certain ATO server, a new hash table will be allocated to store all user specific nodes. A user specific storage is necessary to avoid synchronization between the asynchronous memory accesses made by other user threads working in parallel.

To provide unique identifications for all nodes generated by a user, the counter for node identifiers is initialized to an appropriate value depending on the package identifier. As the node identifier is stored in an integer word, every user is able to active 2^8 packages simultaneously, each package with up to 2^{24} memory nodes.

4.4.3 Minimization of Data Transport

As mentioned above, a special field containing a set of package identifiers was added to the descriptor of a PROSOFT node to maintain information which ATO packages have already stored the most recent version of the node in their memory.

Hence, when calling remotely the ICS only those nodes of parameter or result objects (which are not uptodate in the user memory of the corresponding package side) have to be transported via network. These nodes have to be sent across the network in their address space independent representation, as described above in section 4.4.1.

This mechanism of memory control reduces the flow of data on the network.

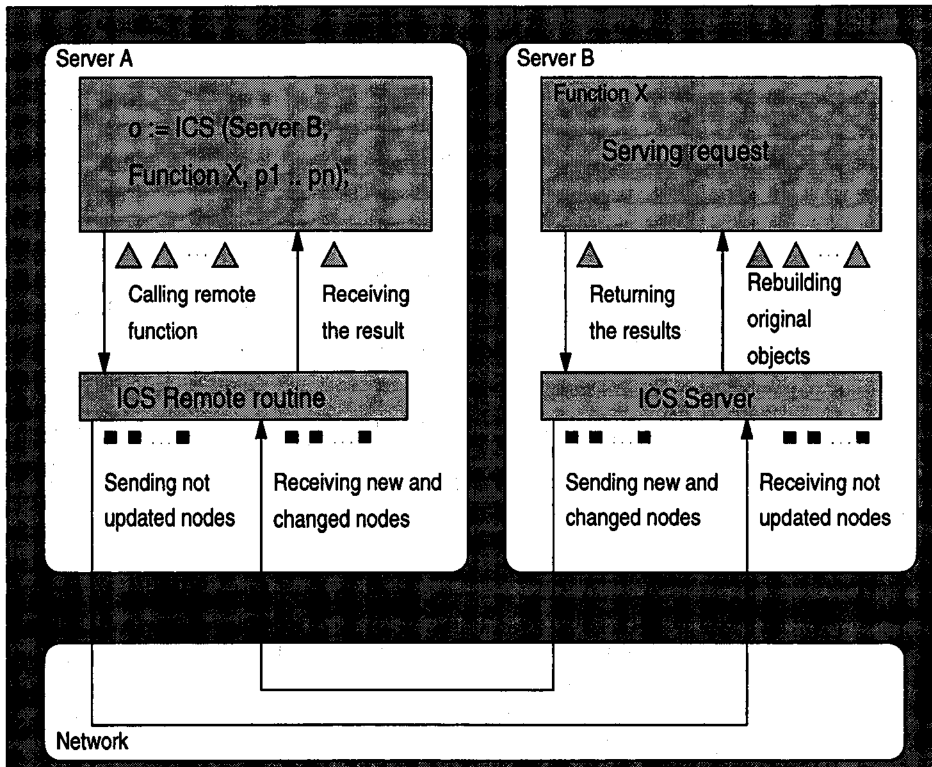


Figure 7: Flow of Data in a ICS Remote Call

The minimized data flow of a remote ICS call is illustrated schematicly in figure 7:

From client to server:

- The identifier of the root nodes of the parameters objects: This information allows the server to recognize in its memory the roots of the objects received as parameters and to rebuild the objects.
- A list of those nodes of the parameter objects which are not uptodate in the server memory.

From server to client:

- The root node identifier of the resulting object. This information is needed to allow the client – on successful completion of the service – to localize and rebuild the result object in its own memory.
- A list of created and modified nodes of the result object which are not uptodate in the client memory.

Example

Figure 8 shows a more detailed example of an ICS request of a client package identified by package id 1 to an ATO server package identified by package id 2. The result object is assumed to be the same one as the parameter object identified by node id 7.

Left hand above:
The parameter object to be sent to package 2 consists of five nodes; three of them (7, 9, 10) are already uptodate in package 2 and two nodes (8, 11) are marked as uptodate only in package 1, since they have been recently modified or created there. Thus, only nodes 8 and 11 have to be sent.

Right hand above:
On the server package 2 the present node 8 has to be updated while node 11 has to be inserted into the object. Both are marked uptodate in packages 1 and 2. On both sides there exists now an identical instance of the object.

Right hand below:
It is assumed that the operation to be executed by the local ICS call on the server side has made some changes to the node 9. Consequently only this node has to be returned as result in order to update node 9 of the client memory.

Left hand below:
Upon successful completion of the ICS call node 9 of package 1 is changed correspondingly and marked as consistent in both packages on both sides.

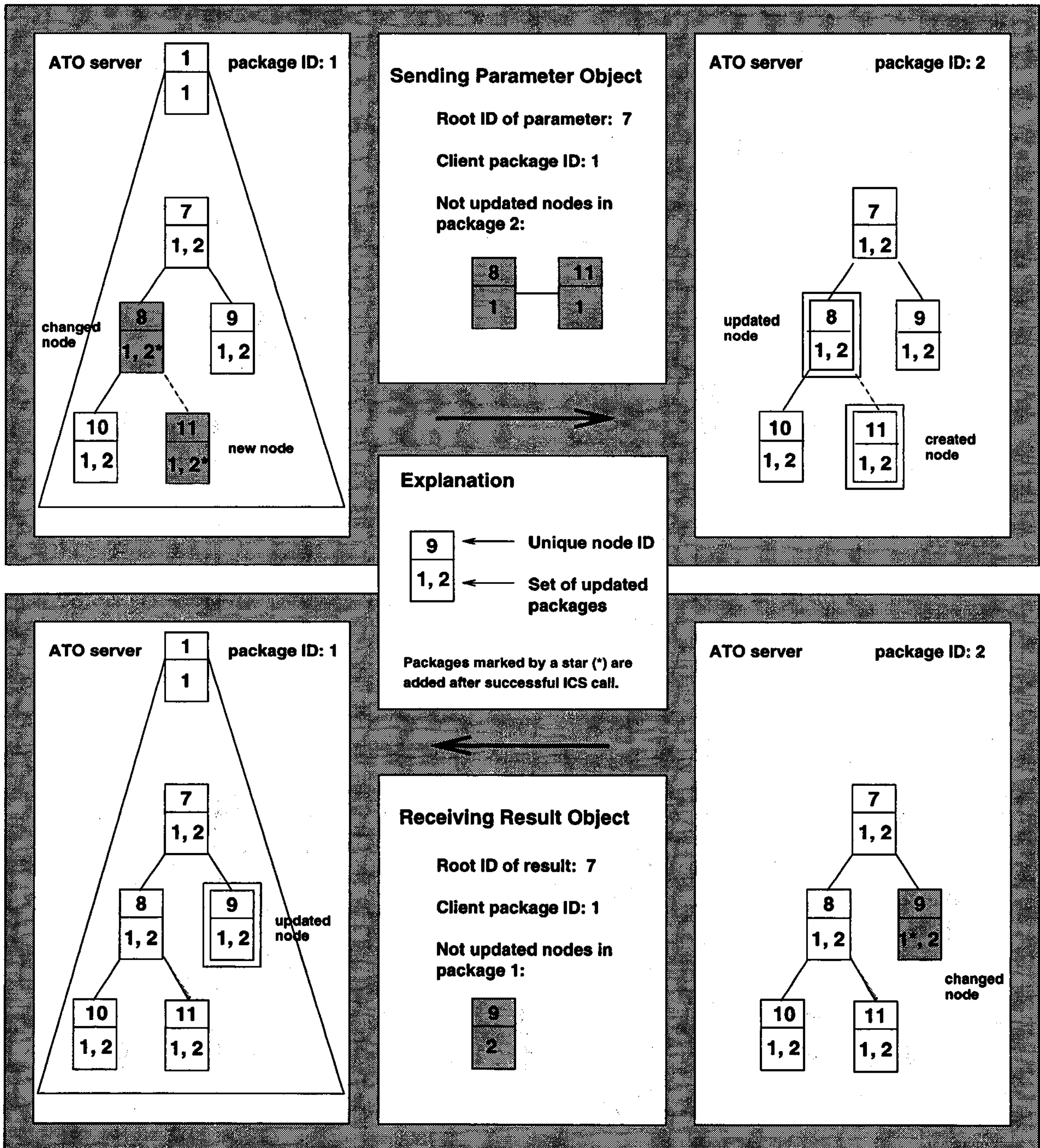


Figure 8: Minimization of data exchange due to memory caching of servers

4.4.4 Garbage Collection

The Garbage Collection is a procedure that frees PROSOFT objects which were not bound to the user data (cenario) during an interaction cycle of the PROSOFT user. The Garbage Collector works in the following way: Every object (when created) is inserted into a so-called Garbage List and taken off from it when it is accessed by a binding operation. This garbage processing is integrated in the primitive functions of the PROSOFT system, and remains therefore completely transparent to an ATO implementer who is using exclusively these primitive operations to construct and manipulate his PROSOFT objects.

The Necessity of Garbage Collection

The Garbage Collection is needed due to the fact that ATO implementers should not be responsable for destroying objects which are not used anymore. Thus, these objects should be automatically destroyed. Without a Garbage Collector some temporary unbound PROSOFT objects would be created and never more destroyed during a PROSOFT session, filling up the available memory with useless data.

The following ICS call may illustrate this:

```
ROTA := ICS (ATO_SADT, DEVOLVE_ROTA, sadt,
        cria_objeto_coordenada(3,2), cria_objeto_real(cont),
        nil,nil);
```

Here, two objects are created: one of type `coordenada` and another one of type `real`. Both objects will not be bound to the user's `cenario`, because they are temporary objects. Without Garbage Collection these objects would stay in the memory until the end of the user session. However, the primitive operations `cria_objeto_coordenada` and `cria_objeto_real` are implemented in a way that all new created objects are put automatically into the Garbage List and, hence, whilst staying unbound can be removed at an appropriate time.

New Concept and Functionality of the Garbage Collection

The cycle of Garbage Collection of the conventional PROSOFT system actually depends on the main loop of the system, i.e. the interaction cycle of the PROSOFT user. Whenever an interaction loop ends, all unbound temporary objects are eliminated by the Garbage Collector.

Moreover, the actual Garbage Collection – which is done by the primitive operations of the system – is using a global variable to maintain the garbage list. This implementation concept conflicts with the implementational necessities of the Distributed PROSOFT. Furthermore, ATOs that are producing garbage objects may be located at different servers. These facts require the necessity to conceive a Distributed Garbage Collection.

The collection cycle of the Distributed Garbage Collector should not be related to the PROSOFT main loop anymore, but must be oriented at the server dispatching routine that serves a remote ICS request. Whenever a client process asks a remote server for a service, its dispatcher routine creates its own garbage list, in a way to keep it consistent with that one on the client side. At the end of the service the dispatcher calls a procedure to destroy all those objects which still have remained in the garbage list. Due to the fact that the dispatcher routine is executed by its own thread of control, the garbage list is bound to a thread-specific variable which can be accessed only by the thread the variable belongs to.

However, both sides involved in a service request must have knowledge, whether an object is garbage or not on the other side. To convey this recognition it is necessary to store this information in the objects themselves, since objects are the entities to be sent to the remote side. Thus, whenever an object (parameter or result) is verified to be garbage on the remote side, it is inserted into the local garbage list too. This procedure warrants the garbage lists on both sides to be consistent.

In summary, the main differences between the conventional and distributed garbage collection are the following:

- The conventional collector is related to the interaction cycle of a PROSOFT user, while the distributed one is bound to the thread of an ICS dispatching routine of an ATO server.
- In contrast to the conventional implementation the distributed one has to take into account, that garbage objects passed as parameters or received as result by a remote ICS call have to be treated as garbage on the server as well as on the client side.

5 Suggestions

As stated previously, the Distributed PROSOFT system provides different mechanisms to allow communication of processes over the network.

Since the distributed programming is more complex than conventional programming, failures of the components of the working environment (servers, network) might cause severe recovery problems. Hence, some suggestions are made which should be taken into consideration.

5.1 Communication between Users

The current communication in Distributed PROSOFT works the way that a user or a client process makes a request to an existing server process. However, communication between different users is not yet possible. Once a base of communication has been implemented, there would exist possibilities to provide communication between the users as well.

The exchange of objects belonging to different users is possible, because the mechanism of remote ICS already provides this functionality. The Distributed PROSOFT could, hence, become a cooperative environment, if an object manager would be provided. Following the PROSOFT paradigms, this manager should be an ATO taking care of the data exchange between users.

5.2 Reliability

Reactivation of Server Processes

A crash of an ATO server may occur. This creates the need to restart the server in order to continue serving the incoming requests. However, the new server process does not have the same information buffered as the old one does; some unexpected effects might occur.

The main problem is in our eyes that some activated servers may have nodes which are marked as uptodate in the server which has just crashed. By this way these nodes are not sent to that process, when a remote ICS call occurs, but they are needed in this new server, so that objects can be rebuilt there.

A solution for this problem is to force the PROSOFT Manager to inform all existing servers, that a specific server has been restarted after a crash, so that they can actualize their package set information.

Change of the Localization of the Prosoft Main Loop

In the current PROSOFT implementation the main loop, i.e. the interaction cycle, is located in the ATO Menu which is incorporated with other frequently used ATOs in a server package. The main data of every user (the so-called `cenario`) are stored in the memory of this package and there is no replication of these data available in the memory of the user main process. Hence, a crash of this package would provoke a total loss of data of all active users.

A solution of this problem would be to move the localization of this central loop from the ATO Menu to the main program of each user, so that the user would be able to recover the data of his cenario.

6 Conclusions

The implementation of Distributed PROSOFT offers many advances reflecting more productivity, better resource usage and an easier configuration of the system. Besides, this environment will be a base to future implementations allowing cooperative work among different PROSOFT users.

The performance of the distributed environment is very satisfactory. Even though the ATOs are distributed over the network, their response time is very similar to the previous environment. The efficiency depends, however, on an appropriate configuration of the ATOs (see section 4.3.1).

The economy of memory resources must also be taken into consideration. In the conventional PROSOFT, every user had the complete set of ATOs available to his private use; a lot of memory resources was wasted. In the new implementation the usage of memory is optimized, once the ATOs are shared among the users.

The new environment also provides easy configurability. The addition and removal of ATOs in the PROSOFT environment was not an ordinary task, since many source codes were supposed to be updated and recompiled. The new environment allows an easy customization of the ATOs, and their servers can be generated automatically by means of a utility using the configuration file.

The goals and aims of the project have been achieved. Now PROSOFT is a remodeled system ajusted to the new requirements. Advanced architectures and mechanisms have been used to implement and to test the new version successfully on multiprocessor machines.

Bibliography

[Blo 92] John Bloomer. *Power programming with RPC.* O'Reilly & Associates, Sebastopol, Calif., 1992. XXXII, 459 p.

[Nun 92] Daltro J. Nunes. Estrategia data-driven no desenvolvimento de software. In *VI Simposio Brasileiro de Engenheria de Software.* Sociedade Brasileira de Computação, Proceedings 1992.

[Nun 93] Daltro J. Nunes. Verbesserung der Software-Qualität durch Verifikation der Korrektheit der Implementierung im Projekt Prosoft. Fakultät Informatik, Universität Stuttgart, Report No. 1993/2.

[Pow 91] M. L. Powell et. al.. SunOS Multi-Thread Architecture. In *Proceedings of the Winter 1991 USENIX Conference, Texas.* USENIX Assoc., 1991. pp. 65-79.

[Sch 94] Heribert Schlebbe. Distributed Prosoft. *Report on a working stay at the Institute of Computer Science of the State University of Rio Grande do Sul (UFRGS) at Porto Alegre / Brazil, from May 1 to June 15, 1994,* Institut für Informatik, Universität Stuttgart, 1994.

[Sun 95] Multithreaded Programming Guide. SunSoft Press, USA, 1995.

Appendix

Syntax of the Configuration File

In the following is given an example of the configuration file of the distribution of ATOs over the network and the rules to create it.

The syntax of the file is very simple:

<prosoft_configuration> := *<server_configuration>*∗ .
<server_configuration> := *<package_specification>* *<ATO_specification>*∗ .
<package_specification> := '@' *<package_name>* ':' *<server_type>* ':' *<server_host>* ':' *<server_program>* ':' .
<package_name> := **string** .
<server_type> := 'M' | 'S' | 'L' .
<server_host> := **string** .
<server_program> := **string** .
<ATO_specification> := '&' *<ATO_identifier>* ':' *<ATO_name>* ':' .
<ATO_identifier> := **integer** .
<ATO_name> := **string** .

Comments: Comment lines may be inserted to the configuration file by using the character % at the beginning of a line.

Specification of ATO packages: A package is defined by indicating the following items:

1. name of the package;
2. server type, i.e. how it is going to be executed (L – Local, S – Single-user or M – Multi-user);
3. name of the host where the server will be executed, and finally
4. directory where the server executable resides (if no host name is given, the server will be started for exclusive use of the user).

Inclusion of ATOs in a package: Designates an ATO to be incorporated into the last defined ATO package (ATO identifiers and their names are defined in a file of the PROSOFT system called `nomes.h`).

The following configuration file was used to distribute some ATO servers on different hosts of the workstation pool at the Institut für Informatik, Universität Stuttgart. The Distributed PROSOFT implementation was tested under these conditions:

```
% Pacote gráfico
@grafico:S:tanne:/user/granvile/prosoft/servers/grafico_svc:
%
% -----------------------------------------------------
% Pacote Kernel
@kernel:M:tanne:/user/granvile/prosoft/servers/kernel_svc:
&50:ATO_CENARIO:
&51:ATO_QUADRO:
&52:ATO_MENU:
&53:ATO_OPCAO:
&54:ATO_SERVICE:
&66:ATO_LISTA:
%
% -----------------------------------------------------
% Pacote Local - Diretório
@directory:L::directory_svc:
&58:ATO_DIRETORIO:
&66:ATO_LISTA:
%
% -----------------------------------------------------
% Methods
@methods:M:trick:/user/granvile/prosoft/servers/methods_svc:
&60:ATO_NSD:
&61:ATO_ROTA:
&62:ATO_SADT:
&56:ATO_JACK:
&66:ATO_LISTA:
%
% -----------------------------------------------------
% Sala
@sala:M:tanne:/user/granvile/prosoft/servers/sala_svc:
&96:ATO_SALA:
&66:ATO_LISTA:
&57:ATO_TEXTO:
% -----------------------------------------------------
```

```
#define ICS_PROG    60    /* Package program number */
#define ICS_VERS    1     /* Package version number */
#define ICS_PROC    1     /* ICS procedure number   */
. . . . . .
```

Description of a Prosoft object

```
struct obj_struct {
    objeto          antecessor;
    objeto          posterior;
    objeto          sucessor;
    integer         NodeId;     <----- Unique node identifier
    integer         PkgSet;     <----- Set of updated packages
    tipo_nodo       tipo;
    tipo_valores    valores;
};
typedef struct obj_struct *objeto;
```

Description of ICS parameters

```
struct ics_params {
    integer         UserId;     <----- User identifier
    integer         PkgId;      <----- Identifier of return package
    int             ato;
    int             oper;
    objeto          p[5];
};
```

Definition of the server's program and version and the ICS procedure

```
program ics_server_program {
     version ics_server_version {
          objeto ICS (ics_params) = ICS_PROC;
     } = ICS_VERS;
} = ICS_PROG;
```

Figure 9: Node structure of Distributed PROSOFT

```
#define ICS_PROC  1
objeto remote_ics (int ato, oper; objeto p1, p2 ...)
{
    CLIENT     *cl;              ics_params params;
    info_type  *info;            objeto     result;
```

Get information from the Prosoft manager about the host, path of the server executable, program & version number of the ATO ato.

```
    info = consult_prosoft_manager (ato);
```

Create an RPC client with TCP connection to the host using the RPC program number of the ATO-specific ICS call. If this fails, the ATO-server is started automatically.

```
    do {
       cl = clnt_create (info->host, info->prog, info->vers, "tcp");
       if (cl == NULL)
          start_ato_server (info->host, info->path);
    } while (cl == NULL);
```

Prepare the Prosoft objects of the ICS call for transport into the server address space

```
    params = encode_ics_params (ato, oper, p1, p2, ...);
```

Send the arguments to the server using the xdr transport functions xdr_ics_params receive the result from the server using the xdr transport functions xdr_objeto.

```
    clnt_call (cl, ICS_PROC, xdr_ics_params, params, xdr_objeto, &result,
               CALL_TIMEOUT);
```

Decode the resulting ICS object for the client's address space and return it

```
    result = decode_ics_result (result, ato, oper, p1, p2, ...);
}   return (result);
```

Figure 10: Remote ICS (client side)

Server's RPC address

```
#define PACKAGE_PROG  60  /* Server program number of the package */
#define PACKAGE_VERS  1   /* Server version number of the package */
#define ICS_PROC  1       /* Server procedure number of ICS (the only procedure defined) */
```

Server dispatch function

```
void package_prog (rqstp, transp) struct svc_req *rqstp; SVCXPRT *transp;
{   ics_params params;  objeto result;
    if (rpstp->rq_proc == ICS_PROC) { /* Server procedure ICS_PROC called */
        Get the ICS arguments using the xdr data representation for transport of ICS parameters
        if (! svc_getargs (transp, xdr_ics_params, &params)) {
            fprintf (stderr, Cannot decode arguments"); return;
        }
        Build and identify the Prosoft parameter objects for the server's address space
         params = decode_ics_params (params);
}
        Call the ICS function of the ATO-package
        result = ICS (params->ato, params->oper, params->p1, params->p2, ...);
        Prepare the Prosoft result object for transport back into the client's address space
         result = encode_ics_result (result, params);
        Send the result object to the client by using the xdr transport function for objects
        if (! svc_sendreply (transp, xdr_objeto, &result) {
}   }   }    printf (stderr, "Server system error: cannot send result"); return;
```

Figure 11: Remote ICS (server side)