On Remote Procedure Call*

Patrícia Gomes Soares†
Distributed Computing and Communications (DCC) Lab.
Computer Science Dept., Columbia University
New York City, NY 10027

Abstract

The Remote Procedure Call (RPC) paradigm is reviewed. The concept is described, along with the backbone structure of the mechanisms that support it. An overview of works in supporting these mechanisms is discussed. Extensions to the paradigm that have been proposed to enlarge its suitability, are studied. The main contributions of this paper are a standard view and classification of RPC mechanisms according to different perspectives, and a snapshot of the paradigm in use today and of goals for the future of RPC.

1 Introduction

Over time, operating-system support as well as language-level support have been designed to ease the development of *distributed applications*. A *distributed application* is a set of related programs that execute concurrently or in parallel, on the same or distinct autonomous processing *nodes* of a *distributed system*, interchanging information.

The decentralized configuration provided by a distributed system brought many benefits. The interconnection of these processing units makes resources available to a wider community. Both data and processes[1] can be replicated thereby achieving greater performance through parallelism, increased reliability and availability, and higher fault tolerance. The remote accessibility of specialized units provide better performance for applications with specific requirements.

Initially, only operating-system-level primitives supported interprocess communication. Unfortunately, the details of communication were not transparent to application programmers. To transmit complex data structures, it was necessary to explicitly convert them into or from byte streams. The programmer needed to be aware of heterogeneity at the machine level, as well as at the language level, between the sending and the receiving processes. No compile-time support was offered by the languages, and debugging became an extremely difficult task. A higher level of abstraction was needed to model interprocess communication. A variety of language-level constructors and operators were then proposed to address this problem. They improved readability, portability, and supported static type checking of distributed applications. A major achievement in this area was the Remote Procedure Call (RPC).

The procedure mechanism, which is provided by most imperative nondistributed languages, was initially proposed to support code mod-

*The IBM contact for this paper is Jan Pachl, Centre for Advanced Studies, IBM Canada Ltd., Dept. B2/894, 895 Don Mills Road, North York, Ontario, M3C 1W3.

†This work was partly supported by Brazilian Research Council (CNPq) under Grant 204543/89.4.

[1] A running program unit is a process.

ularization, and to be an abstraction facility. RPC extends this mechanism to a distributed environment, whereby a process can call a procedure that belongs to another process. Interprocess communication is then given the syntax and semantics of a well-accepted strongly typed language abstraction. Significant results have been achieved in the effort to better accommodate in the model problems related not only with interprocess communication issues, but also with distributed environment configurations.

Computing systems continue evolving at an astonishing speed, and parameters change quickly. A high-speed network is capable of continuously broadcasting a database, thus providing for an almost zero access time to data. An operating system today plays the role of a decentralized manager that searches for facilities to provide applications with necessary resources. Users are close to a declarative environment where they specify *what* their needs are, and the distributed system is supposed to select, based on the instantaneous configuration of the system, *how* to provide it.

Much demand is placed on mechanisms that isolate architectural dependent characteristics from the user. The RPC is an abstraction which can make these dependencies transparent to the user. A system supporting RPCs may provide for *location transparency*, where the user is unaware of *where* the call is executed.

The RPC constitutes, also, a sound basis for moving existing applications to the distributed system environment. It supports software reusability, which can be one of the most important medicines in attacking the software crisis [69].

In this paper, the whole RPC concept is explored, and surveyed. In Section 2, the underlying architecture is described, and in Section 2.1 the concept of RPC is introduced. Section 3 presents the major challenges in supporting the paradigm. The standard design of an RPC system is described in Section 4, as well as its major components, their function, and how they interact. Sections 5, 6, and 7 survey each of those major components, analyzing the problems they are supposed to deal with, and describing several systems and their distinct approaches. In Section 8, extensions and software support added to the RPC paradigm are discussed. Finally, Section 9 describes measurements that can be used to evaluate the performance of RPC systems.

2 General Concepts

Before exploring the concept of RPC, the underlying environment is described here. The major environment requirements that RPC is aimed to support are introduced. Other approachs that address these features are surveyed.

A *distributed computing system* consists of a collection of autonomous processing nodes interconnected through a communication network. An *autonomous processing node* comprises one or more processors, one or more levels of memory, and any number of external devices [8, 47]. No homogeneity is assumed among the nodes, neither in the processor, memory, nor device levels. Autonomous nodes cooperate by sending *messages* over the network. The *communication network* may be *local area* (*short haul*), *wide area* (*long haul*), or any combination local ones and wide ones connected by gateways. Neither the network nor any node need be reliable. In general, it is assumed that the communication delay is long enough to make the access to nonlocal data significantly more expensive than the access to local primary memory. It is arguable, though, as to whether or not the nonlocal access is not significantly more expensive than the local access when secondary storage is involved [59].

A *distributed program* consists of multiple *program units* that may execute concurrently on the nodes of a distributed system, and interact by exchanging messages[2]. A running program unit is a *process*. The *distributed programming* consists of implementing *distributed applications*. The *distributed languages* consist of programming languages that support the development of distributed applications. There are three important features that distributed

[2]In shared memory distributed systems, program units can also interact through shared memory. These systems will not be considered in this paper, and some assumptions and assertions made may not be valid for those environments.

languages must deal with that are not present in sequential languages [8]:

- Multiple program units
- Communication among the program units
- Partial failure.

A distributed language must provide support for the specification of a program comprising multiple program units. The programmer should also be able to perform *process management*. That is, to specify the location in which each program unit is to be executed; the mode of instantiation, heavy weighted or light weighted; and when the instantiation should start. The operating system allocates a complete address space for the execution of heavy weighted processes. The address space contains the code portion of the process, the data portion with the program data, the heap for dynamic allocation of memory, and the stack, which contains the return address linkage for each function call and also the data elements required by a function. The creation of a light-weighted process causes the operating system to accommodate the process into the address space of another one. Both processes have the same code and data portions and share a common heap and stack spaces. Process management can be used to minimize the execution time of a distributed program, where the parallel units cooperate rather than compete. A full analysis of the limitations of synchronous communication in languages that do not support process management is found in [46]. Processes can be created either implicitly by their declaration, or explicitly by a create construct. The total number of processes may have to be fixed at compile time, or may increase dynamically.

Language support is also needed to specify the communication and synchronization of these processes. Communication refers to the exchange of information, and synchronization refers to the point at which to communicate.

For applications being executed in a single node, the node failure causes the shutdown of the whole application. Distributed systems are potentially more reliable. The nodes being autonomous, a failure in one does not affect the correct functioning of the others. The failure of only a subset of the nodes in an application's execution, *partial failure*, leaves still the opportunity for the detection and recovery of the failure, avoiding the application's failure. Considerable increase in reliability can also be achieved by simply replicating the functions or data of the application on several nodes.

Distributed languages differ in modeling program units, communication and synchronization, and recoverability. Some languages model program units as objects and are *distributed object-based*. An extensive survey of distributed object-based programming systems is described in [16]. Others model the program units as processes units and are named *process-oriented* (Hermes [75], and NIL [76]). Yet others model program units into *tasks*, as well as processes (Ada [20]), where a task is a less expensive instantiation of program units than the other two methods.

Many languages model communication as the sending and reception of messages, referred to as *message-based languages*, simulating datagram mailing (CSP [35], NIL [74]). A process P willing to communicate with a process Q, sends a message to Q and can continue executing. Process Q, on the other hand, decides when it is willing to receive messages and then checks its mail box. Process Q can, if necessary, send messages back to P in response to its message. P and Q are *peer processes*, that is, processes that interact to provide a service or compute a result. The application user is responsible for the correct pairing and sequencing of messages.

Several languages require the processes to synchronize in order to exchange information. This synchronization, or meeting, is *rendezvous*, and is supported by languages such as Ada [20], Concurrent C [27], and XMS [26]. The rendezvous concept unifies the concepts of synchronization and communication between processes. A *simple rendezvous* allows only unidirectional information transfer, while an *extended rendezvous*, also referred to as *transaction*, allows bidirectional information transfer.

The *Remote Procedure Call* (*RPC*) mechanism merges the rendezvous communication and synchronization model with the procedure model. The communication details are handled by the mechanisms that support such abstrac-

tion, and are transparent to the programmer. Languages that support RPC are referred to as *RPC-based languages*[3] (Mesa [13], HRPC [55], Concert C [90, 7]). For the synchronization issue, RPC-based languages are classified as the ones that block during a call until a reply is received, while message-based languages unblock after sending a message or when the message has been received.

A distributed language may provide exception handling mechanisms that allows the user to treat failure situations. The underlying operating system may also provide some fault tolerance. Projects such as Argus [44] concentrate on techniques for taking advantage of the potential increase in reliability and availability.

A survey of programming languages paradigms for distributed computing is found in [8], and an analysis of several paradigms for process interaction in distributed programs is in [5] .

2.1 RPC: The Concept

This paper assumes familiarity with the operational semantics of the procedure call binding. Section A of the Appendix presents a quick review of the concept.

RPC extends the procedure call concept to express data and control scope transfer across threads of execution in a distributed environment. The idea is that a thread (*caller thread*) can call a procedure in another thread (*callee thread*), possibly being executed on another machine. In the conventional procedure call mechanism, it is assumed that the caller and the callee belong to the same program unit, that is, thread of execution, even if the language supports several threads of execution in the same operating-system address space as shown in Figure 1. In RPC-based distributed applications, this assumption is relaxed, and interthread calls are allowed as shown in Figure 2. Throughout this paper, *local* or *conventional call* refers to calls in which the caller and callee are inside the same thread, and *remote calls* or *RPC* refers to calls in which the caller and callee are inside distinct threads. Section B of the Appendix presents an extension of operational semantics of the procedure call presented in Section A, to handle RPCs.

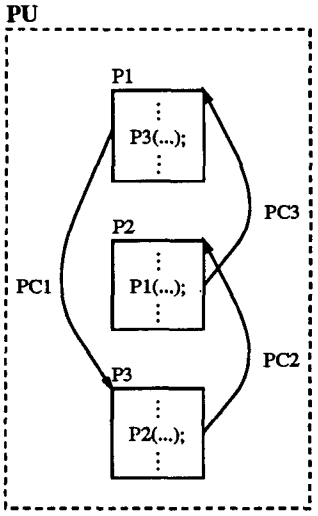

Figure 1: Conventional Procedure Call Scenario

Distributed applications using RPC are then conceptually simplified, communicating through a synchronized, type-safe, and well-defined concept. A standard RPC works as follows. Thread P that wants to communicate with a thread Q, calls one of Q's procedures, passing the necessary parameters for the call. As in the conventional case, P blocks until the called procedure returns with the results, if any. At some point of its execution, thread Q executes the called procedure and returns the results. Thread P then unblocks and continues its execution.

The communication mechanism has a clean, general, and comprehensible semantics. At the application level, a programmer designs a distributed program using the same abstraction as well-engineered software in a nondistributed application. A program is divided into modules that perform a well-defined set of operations (procedures), expressed through the interface of the module, and modules interact through

[3] There exists a common agreement in the research community on the viability of the use of RPC for distributed systems, especially in a heterogeneous environment [56].

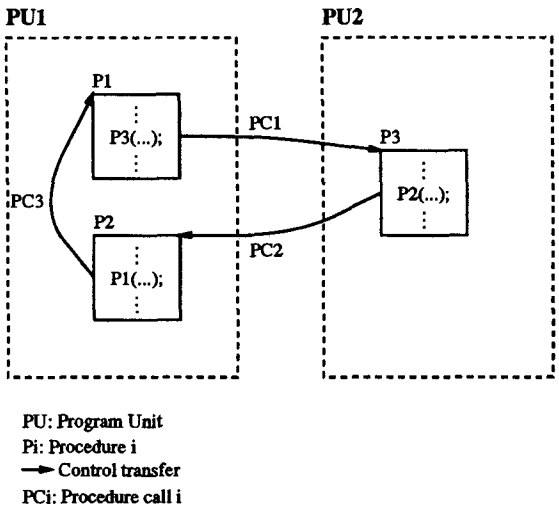

Figure 2: Remote Procedure Call Scenario

procedure calls. Information hiding, where as much information as possible is hidden within design components, and largely decoupled components are provided. For the application user, it is transparent where these modules are executed, and all the communication details are hidden. The implementation framework that supports the concept is an *RPC system* or *RPC mechanism*.

The application programmer or the *RPC system designer* are responsible for mapping the application modules into the underlying architecture where the program is to be executed. The RPC system designer must be prepared to deal with three possible configurations of an RPC from a system-level point of view as shown in Figure 3, where threads are abbreviated by the letter T, operating-system processes by P and machines by M. An RPC can be performed between two threads executing inside a process (RPC1 in the Figure), between two processes running in the same machine (RPC2), or between two processes running in different machines (RPC3). For each situation, the communication media used may vary, according to RPC system designers, environment availability, and possibly the wants of the application programmer. RPC1 could use shared memory, RPC2 could use sockets [72], and RPC3 could use a high-speed network. Communication traffic on the same machine (such as for RPC1 and RPC2) is *cross-domain* traffic [9], and between domains on separate machines is *cross-machine* traffic (as for RPC3).

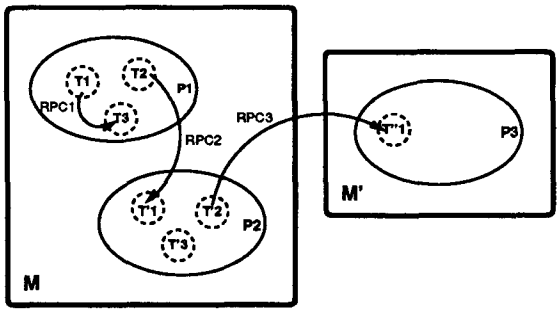

Figure 3: Possible configurations of RPCs

3 Challenges of an RPC Mechanism

The ideal RPC mechanism is the one that provides the application user with the same *syntax* and *semantics* for all procedure calls independently of being a local call or a remote one in any of the possible configurations. The underlying system supporting the concept proceeds differently for each type of call, and it will be in charge of hiding all the details from the application user. The system must be capable to deal with the following differences in the source of the calls, that require development and execution-time support:

- Artificialities may occur in extending the conventional procedure call model, independent of the RPC configuration. Some aspects to be considered are:
  - With the relaxation of the single-thread environment for the caller and the callee, thread identifications and addressing must be incorporated into the model.
  - In the conventional model, the failure of the callee thread means the failure of the caller thread. In the RPC model, this is not the situation and new types of exceptions and recovery are required.
  - Caller and callee threads share the same address space, in the conventional case, and therefore have the

same access rights to the system resources. The implementation of the information exchange protocol is simple. Global variables are freely accessed, and parameters, return values, and their pointers are manipulated. In the RPC model, the RPC mechanism has to move data across address spaces and pointer-based accesses require special handling. Addresses need to be mapped across address spaces and memory need to be allocated to store a copy of the contents of positions aliased by pointers. By using pointers, cyclic structures may be created, and the mechanism must overcome infinite loops during the copying of data.

- The presence of a communication network between caller and callee can cause differences. The RPC mechanism has to handle transport-level details, such as, flow control, connection establishment, loss of packets, connection termination, and so on.

- The distributed environment configuration can cause differences. Issues such as availability, recovery, locality transparency of processes, and so on, can arise. Some of the problems of distributed environments result from heterogeneity at the machine level, operating-system level, implementation-language level, or communication-media level. Distinct RPC mechanisms support heterogeneity at distinct levels, the ultimate goal being to support heterogeneity at *all* levels in a way completely transparent to the application programmer.

Some mechanisms accommodate these requirements by exposing the application programmer to different syntaxes, or semantics, depending on the configuration of the call being issued. Nelson [53] describes and analyzes the major concerns of an RPC mechanism (and their complexity), which can be summarized as follows:

- **Essential properties of an RPC mechanism:**
  - Uniform call semantics
  - Powerful binding and configuration
  - Strong type checking
  - Excellent parameter functionality
  - Standard concurrency and exception handling.

- **Pleasant properties of an RPC mechanism:**
  - Good performance of remote calls
  - Sound remote interface design
  - Atomic transactions, respect for autonomy
  - Type translation
  - Remote debugging.

In Section 8 these issues are considered.

If a remote call preserves exactly the same syntax as a local one, the RPC mechanism is *syntactically transparent*. If a remote call preserves exactly the same semantics of a local call, the mechanism is *semantically transparent*. The term *remote/local transparency* refers to both syntax or semantics transparency, or either of them (when there is no ambiguity).

To support the syntax and semantics transparency, even in the presence of machine or communication failures, the RPC mechanism must be very robust. Nevertheless, an RPC must be executed efficiently, otherwise (experienced) programmers will avoid using it, losing transparency.

RPC mechanisms should provide for the easy integration of software systems, thereby facilitating the porting of nondistributed applications to the distributed environment. The procedure model can limit the ratio of dependency between components of a program, by *coupling*, increasing the potential for software reusability [69]. In languages supporting RPCs, the interaction between modules is specified through a collection of typed procedures, the *interface*, which can be statically checked [29]. Interfaces, being well-defined and documented, give an object-oriented view of the application.

An RPC mechanism must not allow cross-domain or cross-machines access by unauthorized RPC calls. This provides for a *large*

grained protection model [9], because the protection must be divided by thread domains.

Criticism of the paradigm exists [82]. Issues such as the following are still undergoing analysis:

- Program structuring in RPC, mainly difficulties in expressing pipelines and multicasting.
- Major sources of transparency loss, such as, pointers and reference parameters, global variables, and so on.
- Problems introduced by broken networks and process failures.
- Performance issues around schemes to, transparently or gracefully set up distributed applications based on RPC in such a way that the performance is not degraded.

A survey of some features provided by a set of major RPC implementations (such as Cedar RPC, Xerox Courier RPC, SUN[4] ONC/RPC, Apollo NCA/RPC, and so on) can be found in [83]. The implementations are mainly compared considering issues such as design objectives, call semantics, orphan treatment, binding, transport protocols supported, security, and authentication, data representation, and application programmer interface.

4 Design of an RPC System

The standard architecture of an RPC system, which is based on the concept of *stubs* is presented here. It was first proposed in [53] based on communication among processes separated by a network, and it has been extended and used since then.

Stubs play the role of the standard procedure prologue and epilogue normally produced by a compiler for a local call. The prologue puts the arguments, or a reference to them, as well as the return address, in the stack structure of the process in memory, or in registers, and jumps to the procedure body. By the end of the procedure, the epilogue puts the values returned, if any, either in the stack of the process in memory or in registers, and jumps back to execute the instruction pointed to by the return address. Depending on the RPC configuration, stubs may simulate this behavior in a multithread application.

Suppose two threads that want to communicate as shown in Figure 4: the *caller thread*, issues the call to the remote procedure, and the *callee thread* executes the call. When the caller thread calls a remote procedure, for example, procedure `remote`, it is making a local call as shown in (1) in Figure 4, into a stub procedure or *caller stub*. The caller stub has the same name and signature as the remote procedure, and is linked with the caller process code. In contrast with the usual prologue, the stub packs the arguments of the call into a *call message* format (the process is *marshaling*), and passes (2) this call message to the *RPC runtime support* which sends (3) it to the *callee thread*. The *RPC runtime support* is a library designed by the RPC system to handle the communication details during runtime. After sending the call message to the callee thread, the runtime support waits for a *result message* (10), that brings the results of the execution of the call. On the callee side, the runtime support is prepared to receive (4) call messages to that particular procedure, and then to call (5) a *callee stub*. Similar to the caller stub, the *callee stub* unpacks the arguments of the call in a process referred to as *demarshaling*. After that, the callee stub makes a local call (6) to the real procedure intended to be called in the first place (procedure `remote`, in our example). When this local call ends, the control returns (7) to the callee stub, which then packs (that is, marshals) the result into a *result message*, and passes (8) this message to the runtime support that sends (9) it back to the caller thread. On the caller side, the runtime support that is waiting for the result message receives (10) it and passes (11) it to the caller stub. The caller stub then unpacks (that is, demarshals) the result and returns (12) it as the result to the first local call made by the caller thread. In the majority of the RPC mechanisms, the stubs are generated almost entirely by an automatic compiler program, the *stub generator*.

[4] SUN is a trademark of Sun Microsystems, Inc.

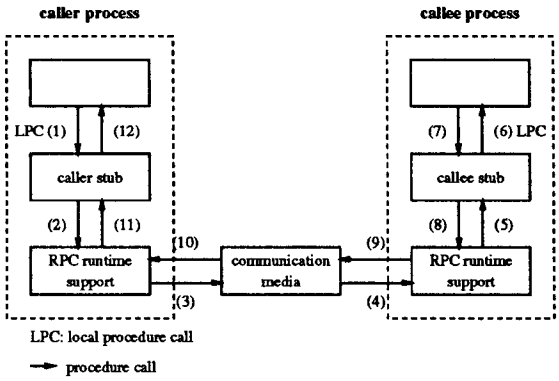

Figure 4: Basic RPC components and interactions

4.1 RPC System Components

RPC systems must interact with a variety of computer system components, as well as support a wide range of functionality, exemplified as follows. The system or the source language must provide a minimum *process management*, that allows the design and execution of threads (heavy-weight or light-weight), that constitute the distributed application. To issue an RPC, a process should acquire an identifier to the respective procedure (name *binding*) so that it can refer to it. RPC systems must provide means for processes to exchange bindings, using a *binding discipline*. The runtime support module must interact with the transport layer to transport the data among the communicating processes should a cross-machine communication occur.

The major components of an RPC mechanism can be divided into three categories:

- *Application-development time components:* components used by the application user during development time.
- *Process modules:* components that are mainly dependent on the specific RPC mechanism. Depending on the particular mechanism, they can be application-development components, execution-time components, or even components predefined and available in the operating system.
- *Execution-time components:* components used only during run time.

RPC facilities can differ in any of these components and in several dimensions. For instance, the binding discipline (a process module component) may differ from one RPC system to another on the interface that they offer to the application user, on the time one process should perform the binding (compilation or run time), on the implementation details of how this binding is performed, on whether the binding deals with heterogeneity and how, and so forth. While these choices should be orthogonal, they are usually hard coded and interwined. As a result, RPC mechanisms usually cannot interact and are extremely difficult to modify to make such interaction possible.

In the following sections, the functionality of each of the components is defined precisely, the dimensions of differences in RPC implementations are analyzed, and real systems and their approaches are explored. A brief historical survey of the systems referred to is presented in Appendix C.

5 Application-Development Time Components

The *interface description language* (*IDL*) and *stub generator* are two major application-development time components. In designing a program unit, the user specifies the procedures that the process can call remotely (*imported procedures*), and the procedures that other processes can call (*exported procedures*). The set of *imported* and *exported* procedures of a process is its *interface* and is expressed through the interface description language (defined in Section 5.1) of an RPC facility.

For the imported and exported procedures of a process, the *stub generator* creates the caller and callee stubs, respectively. These stubs are responsible for the marshaling of arguments, the interaction with the communication media, and the demarshaling of result arguments, if any during run time.

5.1 Interface Description Language (IDL)

An *interface description* (*ID*), or *module specification*, consists of an abstract specification of

a software component (program unit) that describes the interfaces (functions and supported data types) defined by a component. It can be interpreted as a contract among modules. An *interface description language* (*IDL*) is a notation for describing interfaces. It may consist of an entirely new special-purpose language, or may be an extension of any general-purpose language, such as, C.

The concept of an ID is orthogonal to the concept of RPC. Its use motivates the information-hiding concept, because modules interact based on their interfaces and the implementation details are nonapparent. Nevertheless, IDs are of major significance for RPC mechanisms, which use them for generating caller and callee stubs. In general, RPC protocols extend IDLs to support extra features inherited solely from the RPC mechanism.

In specifying a function, an ID contains its signature, that is the number and type of its arguments, as well as the type of its return value, if any. More elaborate IDLs also allow for the specification of the other important options:

- Parameter passing semantics
- Call semantics
- Interface identification.

It is not always possible to find out the *directionality* of the arguments in a function call based only on its ID, because of the diversity of *parameter passing semantics* (*PPS*) supported by high-level languages. By *directionality* is meant the direction of the information flow, that is, whether the caller is passing values to the callee, or vice versa, or both. The PPS specification explicitly determines how the arguments of a function are affected by a call. Concert [90, 7] defines a complete set of attributes. As directional attributes, Concert contains **in**, **in (shape)**, **out**, and **out (value)**. It makes a distinction between the *value* and the *shape* of an argument. The *value* refers to the contents of the storage, while *shape* refers to the amount of storage allocated, as well as the data format. The **in** attribute means that the information is passed from the caller to the callee, and because of that must be initialized before the call. The **in (shape)** attribute means that only the *shape information* of the data is passed from the caller to the callee. The **out** attribute means that the information is passed from the callee to the caller, and the callee can change the value of the parameter, as well as its shape information. The **out (value)** attribute is similar to **out**, but the shape information does not change. Only the value of the parameter changes.

Concert also defines the following *value-retention* attributes: **discard**, **discard (callee)**, **discard (caller)**, **keep**, **keep (callee)**, and **keep (caller)**. These attributes specify whether copies of storage generated by the runtime system during the procedure's execution should remain or be deallocated (discarded). These allocations are generated as part of the interprocess operation. The **keep** attribute and the **discard** attribute specify that both caller and callee can keep or discard the storage allocated.

To deal with pointers across operating-system address spaces or across machine boundaries, Concert defines the following *pointer attributes*: **optional**, **required**, **aliased**, and **unique**. A pointer to a **required** type is pointer that cannot be NULL, whereas a pointer to an **optional** type can be NULL, simply indicating the absence of data. A pointer to a **unique** type is a pointer to data that cannot be pointed to by any other pointer used in the same call. In other words, it does not constitute an alias to any data being passed during the call. Pointers to **aliased** types, on the other hand, are pointers to data that may also be pointed to by other pointers in the same call. This property is expressed in the transmission format, and reproduced on the other side, so that complex data structures, such as graphs, can be transmitted.

The *call semantics*, on the other hand, is an IDL feature inherited from the RPC model. In conventional procedure call mechanisms, a thread failure causes the caller and callee procedures to fail. A failure is a processor failure, or abnormal process termination. If the thread does not fail, the called procedure was executed just once in what is called *exactly-once semantics*. If the thread fails, the runtime system, in general, restarts the thread, the call is eventually repeated, and the process contin-

ues until it succeeds. Just as procedures may cause side-effects on permanent storage, earlier (abandoned) calls may have caused side-effects that survived the failure. The results of the last executed call are the results of the call. These semantics are called *last-one semantics*, *last-of-many semantics*, or *at-least-once semantics*.

In the RPC model, caller and callee may be in different operating system address spaces, as well as in different machines. If one thread is terminated, the other may continue execution. A failed caller thread may still have outstanding calls to other nonterminated callee threads. A callee, on the other hand, may fail before receiving the call, after having participated in the call, or even after finishing the call. For callers and callees on different machines, the network can also cause problems. A call message can be lost or duplicated, causing the RPC to be issued none or several times. If the result message is lost, the RPC mechanism may trigger the duplication of the call.

The RPC mechanism must be powerful enough to detect all, or most of, the abnormal situations, and provide the application user with a *call semantics* as close as possible to the conventional procedure call semantics. This is only a small part of *semantics transparency* of an RPC mechanism, which is discussed thoroughly in Section 8. Nevertheless, it is very expensive for an RPC system, to support exactly-once semantics. The run time would recover from the failure without the user being aware. Weaker and stronger semantics for the RPC model were, therefore, proposed, and some systems extended the IDL with notations that allow the user to choose the call semantics that the user is willing to pay for a particular procedure invocation. The *possibly*, or *maybe* semantics do not guarantee that the call is performed. The *at-most-once* semantics ensures that the call is not executed more than once. In addition, calls that do not terminate normally should not produce any side-effects, which requires undoing of any side-effect that may have been produced so far.

On the other hand, a procedure that does not produce any side-effect can be executed several times and yet preclude the same results without causing any harm to the system. Such procedures are *idempotent* procedures, and, for them, an at-least-once semantics are equivalent to an exactly-once semantics. Another way of approaching the problem is to extend the IDL so that the user can express such properties as idempotency of a procedure, and let the system do the work.

Some RPC mechanisms extend IDLs to support the declaration of an *interface identifier*, which may be just a name, or number, or both, or it may include such additional information as the version number of the module that implements it. This identifier can be particularly useful for RPC mechanisms that support the binding of sets of procedures, rather than just one procedure at a time. The binding can be done through this identifier.

The Polygen system [14], within the UNIX[5] environment, extends the functionality of interfaces a little further. In addition to a module's interface, the interface describes properties that the module should pursue, and rules to compose other modules into an application. A site administrator provides a characterization of the interconnection compatibilities of target execution environments, and an abstract description of the compatibility of several programming languages with these environments. This characterization is in terms of rules. Applying these rules, Polygen generates, for a particular interface, an *interface software* that can be used to compose components in a particular environment. Besides the usual stubs, the stub generator creates commands to compose and create the executables, including commands to compile, link, invoke, and interconnect them. The source programs, the generated stubs, and the generated commands are brought together in a *package*. The activity of analyzing program interfaces and generating packages is *packing*. This scheme allows for components to be composed and reused in different applications and environments without being modified explicitly by the software developer.

In Cedar [13], Mesa [51] is the language used as IDL and as the major host language for the programs. No extensions were added. The programmer is not allowed to specify arguments or results that are incompatible with the lack of

[5]UNIX is a trademark of UNIX System Laboratories, Inc.

shared memory.

NIDL [22], the IDL of NCA/RPC, is a *network interface description language*, because it is used not only to define constants, types, and operations of an interface, but also to enforce constraints and restrictions inherent in a distributed computation. It is a pure declarative language with no executable constructs. An interface in NIDL has a header, constant and type definitions, and function descriptions. The header comprises a *UUID* (Universal Unique Identifier), a name, and a version number that together identify an interface implementation. NIDL allows scalar types and type constructors (structures, unions, top-level pointers, and arrays) to be used as arguments and results. All functions declared in NIDL are strongly typed and have a required *handle* as their first argument. The *handle* specifies the object (process) and machine that receives the remote call. A single global variable can be assigned to be the handle argument for all functions in an interface, making the handle argument implicit and permitting the binding of all functions of an interface to a remote process. This scheme disturbs the transparency and semantic issues of RPCs somewhat. The required argument reminds the user that the procedure that is being called is a remote one, losing transparency. Yet, the flexibility of making it implicit, increases the chance of confusing the user who is used to local call interface and who may not expect to be calling a remote function. Consequently, the user may not be prepared to handle exception cases that can arise. In the implicit situation, the application user is responsible for implementing the import function that is automatically called from the stubs when the remote call is made.

Procedures declared in an interface can be optionally declared as `idempotent`. The NIDL compiler generates the caller and callee stubs written in C, and translates NIDL definitions into declarations in implementation languages (C and Pascal are the supported languages so far). This scheme makes an interface in IDL general enough to be used by any programming language if the NIDL compiler can do the conversion. These files must be included in the caller and callee programs that use the interface. In this interface, the user has the flexibility of declaring *transmissible types*, which the NIDL compiler cannot marshal. The user is responsible for the design of a set of procedures that convert the data to or from a *transmissible type* to or from types that NIDL can marshal. This allows, for instance, the user to pass nested pointers as arguments to remote calls.

SUN RPC [79] defines a new interface description language, *RPC Language* (*RPCL*). In the specification file, each program has a number, a version, and the declaration of its procedures associated with it. Each procedure has a number, an argument, and a result. Multiple arguments or multiple result values must be packed by the application user into a structure type. The compiler converts the names of the procedures into remote procedure names by converting the letters in the names to lower case and appending an underscore and the version number. Every remote program must define a procedure numbered 0, a null procedure, which does not accept any arguments, and does not return any value. Its function is to allow a caller who, by calling it, can check whether the particular program and version exist. It also allows the client to calculate the round-trip time.

In the Modula-2 extension [1] to support RPC, the old Modula-2 external module definitions were used as the interface description with only a few restrictions and extensions added. The idea was to make as few modifications as possible, allowing any external modules to be executed remotely. For restrictions, remote external modules cannot export variables and formal parameters of pointer types do not have the same semantics as when called locally. For extensions to the syntax, a procedure can be defined as being idempotent, a variable can be defined as of type result (out parameter) or segment. The attribute segment allows the user to directly control the Modula/V system to move exactly one parameter as a large block, and to move the other data needed in the call as a small message.

5.2 Stub Generator

During runtime, the *stubs* are the RPC mechanism generated components that give the user the illusion of making an RPC like a local call. The main function of stubs is to hide the imple-

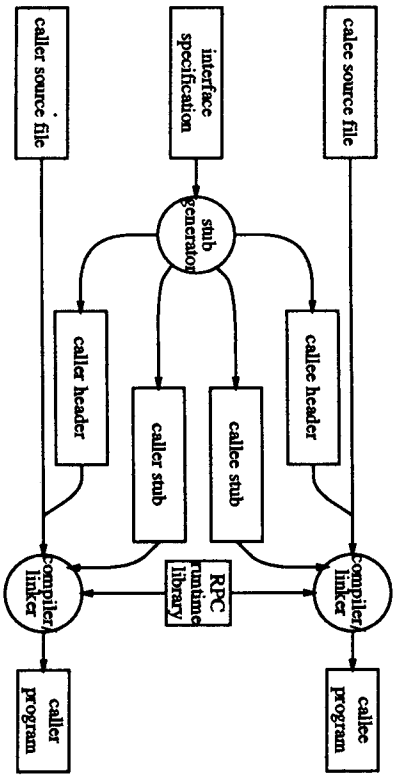

Figure 5: Stub Generator Global Overview

mentation details from the application designer and the user. They are usually generated at application development time by the RPC mechanism component, the *stub generator*. From the interface description of an interface, the *stub generator* generates a callee header file, a caller header file, a caller stub, and a callee stub for each of the functions declared in the interface as shown in Figure 5. If the caller and callee are written in the same language, the header file can be the same for both the caller and callee. Some stub generators [29] also generate *import* and *export* routines. Such routines perform the binding of the respective interface or procedure.

Stubs are the RPC mechanism components responsible for the transmission of abstract data types expressed in (high-level) programming languages over a communication media. A *data representation protocol* defines the mapping between typed high-level values and byte streams. The *marshaling* is the process of converting typed data into *transmitted representation* (*wire representation* or *standardized representation*). The *demarshaling* is the reverse process.

The data representation protocol is responsible for defining the mapping at two levels [73]: the *language level* and the *machine level*. Caller and callee threads can be implemented in the same language or different ones, which is *homogeneity or heterogeneity at the language level* or *homogeneity or heterogeneity at the abstract data type level*. Different languages, such as, FORTRAN, LISP, COBOL, and C, have different abstract data types, and data type conversion may be needed for the communication. Existing compilers for nondistributed languages already provide transformations in the same machine, in the same or different languages, and in the configuration of different data types. These conversions are *implicit conversion*, *coercion*, *promotion*, *widening*, *cast*, or *explicit conversion*. An example of implicit conversion is a function that accepts only arguments of float type, and is called with integer arguments.

Stubs are also responsible for giving semantics to parameter passing by reference, or pointer-based data in general, across a communication media. Several implementations have been proposed. One possible solution would be to pass the pointer (address) only. Any future data access through the pointer sends a message back to the caller to read or write the data [82]. Besides inefficiency issues, this implementation requires that the compiler on the callee side treats local pointers differently from remote ones. This violates one of RPC principles: the compiler should not be affected by the presence of RPCs.

In some other systems [41], a dereference swaps a memory page via the communication media, and the data is accessed locally. This scheme, however, requires homogeneous machines and operating systems, and must ensure that only one copy of a page exists within the system.

The threads can be executed on homogeneous machines presenting *homogeneity at the machine level*, or heterogeneous ones presenting *heterogeneity at the machine level*. Different machines have different primitive data types. In a message-passing system, some con-

version is required when sending data to a dissimilar machine. In a shared memory system, a transformation is needed when the shared memory is accessed by a processor that represents data in a different way than the shared memory.

At the machine level, the data representation protocol must select from three possible dimensions of heterogeneity between machines: (1) the number of bits used to represent a primitive data type, (2) the order of bits, and (3) the coding method. Order of bits refers to choices such as big endian, little endian, and so on. Coding method refers to choices such as 2's complement, sign magnitude, exponent bias, hidden bit, and so forth.

A hardware representation transformation scheme for primitive data types in a heterogeneous distributed computer system is suggested in [73]. It also performs an extensive analysis of the complexity of the several possible architecture models to implement it.

As observed in [73], the primitive data type transformation can be *local*, *central*, or *distributed* through the system. In the *local* model, the machine or a *transformation unit* attached to it performs the transformation of the data into a common format. This data in a common format is sent to the destination, and the transformation unit there transforms the data into the destination format. This approach always requires two transformations, even when dealing with two homogeneous machines. It also requires the specification of a powerful common format that can represent all data types that may be transferred.

In the *central* approach, a *central unit* performs all the transformations. The *central unit* receives the data in the source format, transforms it to the destination format, and sends it to the destination. This central unit keeps information about all the interconnected machines, and each piece of data to be transformed needs to have attached to it its source machine, its data type in the source machine, and the destination machine type.

In the *distributed* approach, *specialized transformation units* scattered throughout the system perform the transformations. A specialized transformation unit can only perform transformations from one source type into one destination type. If no specialized unit exists to transform type A into type C, but there are specialized units to transform type A to type B, and from type B to C, a transformation from A to C can be accomplished via A to B, and B to C. The number of transformation units may be reduced by using a partially connected matrix, and by rotating the data through several specialized transformation units.

At the software level the most common approaches to performing these transformations are the local method and a degeneration of the distributed one. In the local method, the caller stub converts the data from the source representation into a standard one. The callee stub then does the conversion back to the original format. This scheme requires that both caller and callee agree upon a standard representation, and that two transformations are always performed. The use of a standard data representation to bridge heterogeneity between components of a distributed system is not restricted to the communication level. One example is the approach used in [84] for process migration in heterogeneous systems by recompilation. The strategy used is to suspend a running process on one machine at *migration points* to which the states of the abstract and physical machines correspond. Then the state of the binary machine program is represented by a source program that describes the corresponding abstract machine state. This source program is transferred to the remote machine, recompiled there, and continues execution there. The compiler intermediate language is used to communicate between a language-dependent front-end compiler, and a machine-dependent code generator.

In the degenerated distributed approach, either the caller stub (*sender makes it right method*) or the callee stub (*receiver makes it right method*) does the transformation from the source directly into the destination representation. Only one transformation is needed. Nevertheless, the side performing the conversion needs to know all the representation details used by the other side to properly interpret the data. The caller packs the data in its favorite data format if the callee side recognizes it. The callee needs only to convert the data if the format used to transmit the data is not the one used locally. This avoids intermediary con-

versions when homogeneous machines are communicating, and makes the process extensible, because the data format can be extended to specify other features such as new encodings or packing disciplines. In the local approach, the connection of a new machine or language into the system requires only that the stubs for this new component convert from its representation to the standard one. In the distributed method, all the old components must be updated with the information about this new component, or the new component must be updated about all the old ones.

When using the local approach, the standardized representation can be either *self-describing* (*tagged*) or *untagged*. Self-addressing schemes have benefits such as enhanced reliability in data transfer, potential for programs to handle values whose types are not completely predetermined, and extensibility. Untagged representations make more efficient use of the bandwith and require less processing time to marshal and demarshal. The caller and callee must nevertheless agree on the standard representation.

Heterogeneity at the representation level, or more specifically, at the mapping level between abstract data types and machine bytes, has already been studied in a nondistributed environment. A mechanism for communicating abstract data types between regions of a system that use distinct representations for the abstract data values is presented in [34]. A local approach is used, in which the main concerns is modularity. The implementation of each data abstraction includes the transmission implementation details of instances of that particular abstraction. Each data type implementation must then define how to translate between its internal representation and the canonical representation, and the reverse, which is done through the definition of an *external representation type* for that type. The system supports the transmission of a set of *transmissible types*, that are high-level data types, including shared and cyclic data structures. The external representation type of an abstract type T is any convenient transmissible type XT, which can be another abstract type if wanted.

Stubs are responsible for conversion of data at the language level as well as at the machine level. Over the years, distinct approaches have been developed to attack the conversion problem. Some of the approaches concentrate on solving only the machine level conversion problems, assuming homogeneity at the language level. Others tried to solve only the language level conversions, assuming that communication protocols at a lower level would be responsible for performing the machine-level conversions. More ambitious approaches tried to deal with both problems. In the following section, distinct approaches are studied.

5.2.1 Stub Generator Designs

The Cedar [13] system designed the first stub generator, called **Lupine** [13]. The stub code was responsible for packing and unpacking arguments and for dispatching to the correct procedure for incoming calls in the server side. The Mesa interface language was used without any modifications or extensions. It assumed homogeneity at the machine level and the language level.

Matchmaker [39, 38] was one of the earliest efforts toward connecting programs written in different languages. The purpose was to deal with heterogeneity at the language level. The same idea was later explored by the MLP [33] system. On the other hand, HORUS [28], developed almost concurrently with MLP, aimed at also dealing with heterogeneity at the machine level. These three systems shared the same approach of designing their interface description language, and generating the stubs based on the interfaces.

The HORUS [28] was one of the primary efforts in designing a stub generator for RPC that would deal with heterogeneity at the data representation level and at the language level. All interfaces are written (or rewritten) in a specific interface specification language. The aim was to provide a clean way to restrict the RPC system data structures and interfaces that could be reasonably supported in all target languages. This simplifies the stub generator and provides a uniform documentation of all remote interfaces. The system used a single *external data representation* that defined a standard representation of a *common-language* and *common-machine* data types over

the wire. Data types were then converted from the source language into *common-language*, and later from the *common-language* into the target language. This simplified the stub generator and stub design, because only one representation was allowed. Nevertheless, the external data representation may not represent efficiently some data values from a source or target language. On the other hand, new source languages and machine types can be added to the system without requiring an update in the entire network. The HORUS defined precisely the classes of knowledge needed for a marshaling system, and this classification can be considered one of its major achievements. The information was classified as: *language dependent*, *machine dependent*, and *language and machine independent*. It also defined a format for this knowledge that was easy to manipulate, that was general enough to contain all the information required, and that was self-documenting. All the *language dependent* knowledge required for the marshaling is stored as *language specifications*. They contain entries for each *common language* data type, and additional (data type independent) specifications needed to correctly generate the caller and callee stubs in the particular language. The *machine dependent* information, including primitive data type representation, order of bits, and coding method of all possible target machines is stored as *machine specifications*. Entries for both, machine and language dependent information, are expressed in an internal HORUS command-driven language. The language specification entries contain a set of code pieces for each pair of basic data types (or type constructors) in the *common language* and a possible target language. The set completely specifies all information required by the stub generator to marshal that type for that target language. It contains a piece of code for generating a declaration of the type in the target language, a piece of code to marshal an argument of this type, and a piece of code to demarshal. HORUS works as a compile time interpreter as shown in Figure 6. It generates the stubs and header files by parsing the interface written in *common language* and, for each declaration, it consults the corresponding table and entry, and *executes* the appropriate piece of code associated with

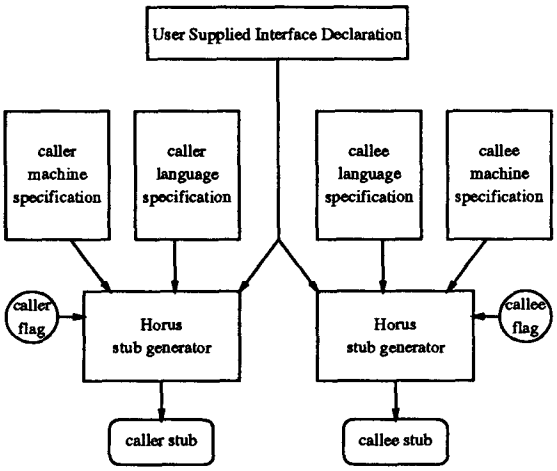

Figure 6: Structure of HORUS Stub Generator

that entry, generating the stubs and headers. HORUS provides `IN`, `OUT`, and `INOUT` parameter types to optimize passing data by value-result. HORUS provides for the marshaling of pointer-based data types, as well as arbitrary graph structures. A pointer argument is marshaled into an offset and the value it references. The offset indicates where in the byte array this value is stored. Pointers are chased depth first until the entire graph structure is marshaled. HORUS constructs a runtime table of objects and byte array addresses to detect shared and repeated objects.

The *Universal Type System* [33] (*UTS*) language, another effort towards the support of heterogeneity at the language level was developed concurrently with HORUS. Unlike HORUS, it did not deal with heterogeneity at the machine level. The language was designed to specify interfaces in the *Mixed Language Programming System* (*MLP System*), where a *Mixed Language Program* was written in two or more programming languages. A program consisted of several *program components*, each written in a *host language*. The UTS, a set of types, and a type constructor contain a language for constructing *type expressions* or *signatures* that express sets of types. The language consists of the alternation operator `or`, the symbol - that indicates one unspecified bound in an array or string, the symbol * that indicates an arbitrary number of arguments or unspecified bounds, and the symbol ?, which is

the universal specification symbol that denotes the set of all types. The representation of a value of UTS type is tagged, usually indicating the type and length of the data, and is stored in a machine- and language-independent way.

For each host language in the MLP system, a *language binding* is defined, which is a set of conventions specifying the mapping between the host language types and the language type of the UTS language. The mapping is divided into three parts: mapping for types in which the two languages agree, meaning that each type has a direct equivalent in the other type system; mapping for types that do not have a direct counterpart, but that are representable in the other language; and mapping for UTS types that do not fall in the above categories. For each language binding, there is an *agent* process that performs the mapping, and is written in a combination of that language and a system programming language. This use of multiple languages may be necessary, because the agents need to perform low-level data transformations, and yet be callable from the host language.

In the MLP system, there is a distinction between the interface that a module exports (*exported interface*), and the (possibly several) interfaces that other modules import (*imported interfaces*). These interfaces are specified in the UTS language as signatures that describe the number and UTS type of the arguments and returned values of a procedure. Signatures that make use of the additional symbols in UTS are *underspecified*, because the data types may not map into only one host language-level type, but a set of legal types. The MLP linker, therefore, takes a list of program components that constitute a mixed-language program and performs static type checking of the arguments and parameters of the intercomponent calls. In UTS, a match between an exported signature and an imported one has a more formal notion than simply determining if the signatures are equivalent. As defined in [33]:

> An import signature matches, or *unifies*, with the corresponding export signature if and only if the set denoted by the former is a subset of the set denoted by the latter.

This rule guarantees that the conversion can be done. When a signature does not unify, MLP delays the decoding of the UTS value and transfers the problem to the user to handle. This is done through the use of a *representative* for an UTS value as the argument value. MLP provides the user with a library of routines that can be called passing a *representative value* as an argument to inquire the UTS type of the corresponding value, or to perform explicit conversion of types. Valid operations for the host language types are not allowed to be applied on the representative itself, which is maintained by the agent. Representatives are similar to tickets that are presented to agents: they provide access to the data.

Matchmaker [39, 38], an interface specification language for distributed processing, was one of the earliest efforts toward connecting programs written in different languages. It provides a language for specifying the interfaces between processes being executed within the SPICE network, and a multitargeted compiler that converts these specifications into code for each of the major (*target*) languages used within the SPICE environment. Matchmaker was intended to be a language rich enough to express any data structure that could be efficiently represented over the network, and reasonably represented in all *target languages*. The major target languages were C, PERQ Pascal, Common Lisp, and Ada.

Each remote or local procedure call contains an *object port* parameter, and a list of *in* and *out* parameters. The *object port* specifies the object in which the operation is performed. All parameters, but the object port, are passed by value. Pointers, variable-sized arrays, and unions can only occur in top-level remote declarations, and may not be used when constructing other types. Time-out values can be specified, and the reply wait can be made asynchronous as well.

The Matchmaker language is used to mask language differences, such as language syntax, type representations, record field layout, procedure call semantics, and exception handling mechanisms. Representation for arguments over the network is chosen by the Matchmaker compiler. A Matchmaker specification includes complete descriptions of the types of every ar-

gument that is passed, and the compiler generates the target language type declarations to be imported into the generated code. Matchmaker assigns a unique identification (id) to each message, which is used at run time to identify messages for an interface. After the procedure (message) is identified, the types of all fields within it are also known, because messages are strongly typed at compile time. The compiler is internally structured to allow the addition of code generators for other languages as they are added to the SPICE environment. This allows a callee to be automatically accessible to callers written in any of the supported languages, regardless of the language in which the callee is written.

The *Abstract Syntax Notation One* [37] (*ASN.1*), a more general purpose interface description language, was later designed, and has been widely used in international standard specifications. The ASN.1 interfaces are compiled into its transfer syntax based on the *Basic Encoding Rules* (*BER*), which is used as the external data representation. Each data type in ASN.1 has a rule to translate it into its corresponding transfer syntax. The *ED* [54] library is an implementation of the ASN.1 BER. The library includes encoding and decoding routines that can be used as primitive functions to compose encoders and decoders for arbitrarily complicated ASN.1 data types. It also contains routines that translate data values directly between C and the BER transfer syntax without converting it to an intermediate format. From the ED library, *CASN1*, an ASN.1 compiler, was designed and implemented for translating ASN.1 protocol specifications into C, and for generating the BER encoders and decoders for the protocol defined data types.

Because of the difficulty introduced with parameter passing by reference, or pointer-based data in general, in an RPC, some systems simply do not allow arguments of such type in the declaration of functions that can be called remotely. An example is the SUN RPC [79] implementation, which disallows it and restricts the number of arguments that can be passed in an RPC to one, as well as the return value. If the user wants to pass more than one argument, or receive more than one value, the data must be grouped into a structure data type. The user then ends up being responsible for most marshaling and demarshaling of data. The *rpcgen* [79], the SUN stub generator, generates, from an RPC specification file, the caller and callee stubs, as well as a header file. The *rpcgen* generates the remote procedure names by converting all letters in the names to lowercase, and appending an underscore and the program version number to end of the names. The SUN defined and used a canonical representation for data over the wire, the *eXternal Data Representation* [78] (*XDR*) protocol.

To avoid constrainting the user, another approach was to allow the user to implement special routines for the marshaling and demarshaling of complex data structures. This approach was used, for instance, by the Network Interface Definition Language (NIDL) of the NCA/RPC [22] system and by HRPC [10].

In NIDL, the routines convert the *transmissible types* defined by the user to or from types that the NIDL compiler can marshal. The heterogeneity at the data representation level was dealt by NCA with a Network Data Representation protocol (NDR). A set of values across the communication media is represented by a format label that defines the scalar value representation, and a byte stream with the values. The NDR does not specify the representation of the information in packets, however. The NIDL compiler and the NCA/RPC system encode the format label in packet headers, fragment the byte stream into packet-sized pieces, and pack the fragments in packet bodies. This implementation is very flexible, allowing the *receiver makes it right* approach to deal with heterogeneity in the data representation level.

In HRPC [10], each primitive type or type constructor defined by the IDL is translated from the specific data type in the host machine representation to or from a standard over-the-wire format. No direct support is provided for the marshaling of data types involving pointers, although the user can provide marshaling routines for complicated data types.

The main concern of other systems during the design of the stub generator is the maintenance of location transparency. The Eden system [2] consists of objects called *Ejects* that communicate via *invocations*. It is transparent for the user whether the invocation is for a

local Eject or a remote one. The system transparently does the packing and stub generation when required.

6 Process Modules

The *binding disciplines*, *language support*, and *exception handling* are the main components of the process modules. The *binding disciplines* define a way of identifying and addressing threads, and binding procedure names to threads. The *language support* of the RPC mechanism allows the user to make these mappings, and to issue the remote calls in situations where syntax transparency is not supported. The *exception handling* mechanism supports the handling of abnormal situations.

6.1 Binding Disciplines

In conventional procedure call mechanisms, a *binding* usually consists of finding the address of piece of code, by using the name of a procedure. In the RPC model, a *binding discipline* has also to identify the thread in which the call will be executed, and several approaches may be taken. First of all, there may be several threads that can or are willing to execute a specific procedure. These threads are usually *server threads* (*server processes*), because they give the illusion that they exist to serve calls to these procedures. Some implementations require the caller to identify a specific thread to execute its call. Some require only the name of the machine in which the caller wants the call to be executed, independently of the particular thread. Finally, some implementations require that the caller identify only the procedure to be called, and dynamically and transparently, the runtime support locates the most appropriate thread to execute the call (Cedar [13]). In this way, an RPC binding can change during the caller's execution, and a caller can be *virtually* bound to several similar callees simultaneously.

The binding can be static, performed during compilation or linkage time, or dynamic, performed during runtime. Static bindings are useful if the application configuration is static or seldom changes. They can also be used in situations where the extra cost of binding immediately before the call may, significantly, affect performance, as may be the situation in rising alarms in a process control application.

Two phases are involved in acquiring a binding that fully connects a caller and a callee: *naming*, and *port determination*. The *naming* is the process of translating the caller-specified callee name into the network address of the host on which the callee resides. The *port determination* identifies a callee thread in one host machine. The network address produced during naming is not enough for addressing a callee, because multiple callees may be running on a single host. So, each callee has its own communication port and network address, which uniquely identifies the callee thread. In the conventional situation, the procedure's name is enough, because everything else is known by default.

Some systems allow the threads to export additional *attributes* associated with a binding that provides the callees with more information so that they can decide the thread to use, or with several ways of identifying a thread besides its name. For instance, a thread may export the version number of its implementation, so a callee may choose the newest version, or a particular version of interest. A thread may also publish the data objects upon which it is updated, so a callee might choose a thread that does, or does not, modify a piece of data. The *binding handler* is all the information needed to identify a port.

The binding process works as follows. A thread willing to accept RPCs *exports* (*publishes*) its port address and the procedures that it wants to execute. A thread willing to issue RPCs *imports* the port information of a thread or set of threads it thinks is adequate to execute its calls. The exported or imported address information can connect the caller directly to the callee, or can connect the caller to the RPC runtime in the callee machine that will act as bridge to the callee. The main goal is to make this process work as similar as possible to the conventional procedure scheme, hiding from the application user the *import* and *export* processes. Concert [90, 7] is the system that supports the most conventional and flexible binding mechanism.

To import and export procedures, some systems (Cedar RPC [13], NCA RPC [22],

HRPC [10, 55], and so on) manage a *global name space*, that is, a set of names whose associated data can be accessed from anywhere in the environment. Export corresponds to writing or updating information into this global name space, whereas import corresponds to reading this information. Some other systems use *well-known ports*, where each callee is assigned a specific port number. Callers usually consult a widely published document to get the well-known port numbers. The handler consists of this port number and the home of the machine in which the callee is supposed to run. Because the *global name service* usually involves permanent memory, systems should detect when a callee no longer exists, and *unexport* its associated information. Other systems use a *Binding Daemon* (SUN RPC [79]), a process that resides at a well-known port on each host and manages the port numbers assigned to callees running on that host. Another way would be to have a *server-spawning daemon process*, a process that also resides at a well-known port on each host, but that serves import requests by creating new ports (through forking other processes) and passing to the callee the new port number.

Some systems provide thread facilities to import or export a single procedure, whereas others provide only for the import or export of a set of procedures (an interface). In some situations the interfaces represent a functional related grouping of functions, that typically operate on shared data structures. The unit of binding being the interface is sometimes justified as a means of ensuring that all the functions in that interface are executed by the same server. This is a step towards an object-oriented approach, where program objects can only be accessed through an specific set of procedures.

Another aspect of the binding discipline is the interface exposed to the user. The most adequate interface would be the one that would provide the user with the same call interface (*syntax transparency*), independently of whether the user was making an RPC or a local call. Most systems however require that the RPC handler be an argument of any RPC call (NCA/RPC [22], SUN RPC [79]). This RPC handler must also be the product of calling an *import* function upon an interface or procedure. Some other systems allow for the declaration of an interface-wise global variable that will keep the address of the handler. Although the user interface is similar to a local one, the side-effects of the call can be unfortunate if the user is not careful enough to update the handler when needed.

6.1.1 Different Binding Discipline Designs

In the Cedar RPC Facility [13], the unit exported or imported is an interface, and it is identified by its type and instance. The type specifies in some level of abstraction the interface functionality and the instance, the implementor. For each instance, the *Grapevine* distributed database is used to store the network address of the machine where it is running. At the application user point of view, for each interface imported or exported there is a related function to import or export the binding (network address) of that interface. The *export* function gets the type and instance name of a particular instance of an interface as arguments and updates the database with this information and the machine's network address. The set of users that can export particular interfaces is restricted by the access controls that restrict updates to the database. The *import* function retrieves this information from the database and issues a call to the RPC runtime package running on that machine, asking for the binding information associated with that type and instance. This binding information is then accessible to the caller stubs of that interface. If there is any problem in the callee side, by the time the binding is requested, the caller side is notified accordingly and the application user is supposed to handle the exception. There are several choices for the binding time, depending on the information the application user specifies in the import calls. If the user gives only the type of the interface, the instance is chosen dynamically at the time the call is made by the run time support. On the other hand, the user can specify the network address of a particular machine as the instance identification for binding at compile time.

In NCA/RPC [22], as mentioned previously,

RPCs require a first argument, a handle, that specifies the remote object (process) that receives and executes the call. The primary goal of this scheme is to support *location transparency*, that is, to allow dynamic determination of the location of an object. The local calls have the same syntax. Handles are obtained by calling the runtime support and passing the object (process) identification, the UUID (*U*niversal *U*nique *Id*entification), and network location as input arguments. One flexibility is the implicit semantics. The user declares a global variable for each interface that stores the information necessary to acquire the handle, and designs a procedure that converts this variable into a handle. The handle argument is not passed in the call and the conversion procedure called automatically by the the stubs during run time. To import and export interfaces in NCA, the NCA/LB (Location Broker) was designed, which retrieves location information based on UUIDs. A callee side process exports some functionality by registering its location and the objects, type of objects, or interfaces that it wants to export as a global (to the entire network) or a local (limited users) service. The LB is composed of the *Global Location Database*, a replicated object that contains the registration information of all globally registered processes; and the *Local Location Broker* that contains the information of the local exportations. The Global Database provides a weakly consistent replicated package, meaning that replicas can be inconsistent at any time, but, without updates, all replicas converge to a consistent state in a finite amount of time. A caller process can inquire about the information of a specific node, or can make an RPC by providing an incomplete binding handle, and specifying, for instance, only an interface, and not a specific process, and the runtime system asks the Local Location Broker the reminder of the information.

In HRPC [10, 55], to support the heterogeneity at the data representation protocol, transport protocol, and runtime protocol, the binding is much richer, containing not only the location, but also a separate set of procedure pointers for each of these components. Calls to these routines of these components are made indirectly through these pointers.

Initially, the HRPC binding subsystem queries the name service to retrieve a *binding descriptor*. A binding descriptor consists of a machine-independent description of the information needed to construct the machine-specific and address-space-specific *binding*. It indicates the control component, data representation component, and transport component that the callee uses, as well as, a network address, a program number, a port number, and a flag indicating whether the binding protocol for this callee involves indirection through a binding agent. The HCS Name Service (HNS) applies the concept of *direct-access-naming*, where each HCS system has a naming mechanism and HNS accesses the information in these name services directly, rather than reregistering all the information from all the existing name services. This design allows closed systems named *insular systems* to evolve without interfering with the HNS, and makes newly added insular services immediately available, eliminating consistency and scalability problems. The HCS name service (*HNS*) can be viewed as a mapping between the *global name* of an object and the name of that object in its local system. The local system is the one responsible for the final name to data mapping. Each HNS name then consists of two pieces: the *context*, which determines the specific name service used to store the data associated with the name, and the *individual name* or *local name*. Each name service connected to HNS has a set of *name semantics managers* (*NSMs*). These routines convert any HNS query call to the interface, format, and data semantics of a local call to the name service. These routines are remote routines and can be added to the system dynamically without recompilation of any existing code. For a new system to be added to the HNS, NSMs for that system are built and registered with the HNS; the data stored in them becomes available through the HNS immediately.

Using the *context* part of the name and the query type, the HNS returns a binding for the NSM in charge of handling this query type for this context. The caller uses this binding to call the NSM, passing it the name and query type parameters as well as any query type specific parameters. The NSM then obtains the

information, usually by interrogating the name service in which it is stored.

SUN RPC [79] has no strongly expressed binding model. Some procedures take the IP host name of the machine to which the caller wants to communicate. Others simply take file descriptors to open sockets that are assumed to already be connected to the correct machine. Each callee's host machine has a *port mapper daemon* that can be contacted to locate a specific program. The caller has to make an explicit call to the port mapper to acquire an RPC handler. The caller specifies the callee's host machine, program number, version number, and TCP or UDP protocol to be used. The RPC handler is the second argument of the RPC. The port mapper also provides a more flexible way of locating possible callees. The caller sends a *broadcast* request for a specified remote program, version, and procedure to a particular port mapper's well-known port. If the specified procedure is registered within the port mapper, the port mapper passes the request to the procedure. When the procedure is finished, it returns the response to the port mapper instead, which forwards it to the caller. The response also carries the port number of the procedure so that the caller can send later requests directly to the remote program.

In the Template-Based model [68] for distributed applications design, an application consists of modules that communicate with each other via RPC. Each module can have only one exportable procedure, but it can call any of its local (internal to the module) procedures, as well as any procedure exported by other modules. The module's interaction is expressed through a set of attribute bindings known as *template attachments*. A *template* is a set of characteristics that can be used to specify the scheduling and synchronization structure of a module in an application. It describes the interaction of one module with others. Every module can have up to three templates to specify its behavior: *input*, which describes the correct scheduling and synchronization of incoming calls, *output*, where similar to input for calls originated by the module, and *body*, which describes attributes that can modify the module's execution behavior in the distributed environment. In this model, the unit of binding is a module, which can have only one exportable procedure. The binding is done at call time.

Concert [90, 7] expresses a binding in the most transparent way: as a function pointer. Languages like C, which have already function pointers, need no extensions. Concert simply treats functions as *closures*. The *closures* identify a code body and a process environment in which the body is executed. In Concert, when a process assigns the address of a function body to a function pointer, not only is the function body address implicitly assigned, but also the encompassing process environment. Any function pointer passed to another process preserves this information, exporting a function pointer or binding.

6.2 Language Support

In the traditional procedure call model, the callee process, being the same as the caller process, by default, is always active. In the RPC model, an issue exists as to *when* to activate the process in which the callee is supposed to run. These problems are called *activation problem* or *callee management problem* or *callee semantics problem*. In some systems, the application user is responsible for this activation. In others, callees processes are activated by the underlying process model. There is a *callee manager* that can create callee processes on demand, spawn new processes, or simply act as a resource manager that chooses an idle process from a pool of processes created earlier. Such callees can remain in existence indefinitely, and can retain state between successive procedure calls; or they may last a *session* as a set of procedure calls from a specific client, or they may last only one call, which is a *instance-per-call* strategy. When the activation process is controlled by the runtime system, several instances of a callee can be installed on the same or different machines transparently to the user. This may be only to provide either load balancing, or some measurement of resilience to failure. The binding mechanisms would allow one of these instances to be selected at import time.

Some languages allow the user to start processes that will either be caller or callee from the operating-system level, and also offer lan-

guage support to start new processes during run time. In the latter situation, an interface should be powerful enough to allow passing arguments between the parent and the children, as pointed out in [89]. Supporting parameter passing for newly created processes allows for compile-time protection, more readability, and structuring. Concert [90] allows the passing of arguments between parent and children, and these arguments can be remote procedure references as well.

In Athena RPC [70], callees are characterized by their lifetime, whether they are shared among clients or are private, and the means through which they are instantiated. In the interface definition, the user specifies these characteristics through *attributes* that are associated with callee's stubs. The system is very flexible, allowing callee processes to be instantiated either manually or automatically in response to a request from a caller, or being long-lived, belonging to a pool, and potentially answering multiple callers. This instantiated mode is transparent to the caller.

RPC systems also differ in the way callees respond to the calls. For the syntax of the callee reception, the receive can be *explicit* or *implicit*. In *explicit* mode, the language provides a function that can be called by an already active process just like any other. In the *implicit* mode, the language provides a mechanism for handling calls or messages. The receive is not performed by any language-level active process. The callees are dedicated processes that perform only remote calls. This mode may look more like the behavior in a sequential procedure environment, where the arrival of the call triggers the execution of an appropriate body of code, much as an interrupt triggers its handler (see [65] for further discussion). In essence, a classification can be made [65] in which explicit receive is a message based-mode, whereas implicit receipt is a procedure-based one. The issue of synchronization, though, is orthogonal to the issue of the syntax of how the callee process responds to it.

In Cedar [13], for instance, there is no user-level control to the service of calls on the callee side. There is a pool of callee side processes in the idle state waiting to handle incoming calls. In this way, a call can be handled without the cost of process creation and initialization of state. In peak situations, additional processes can be created to help and terminated after the situation returns to normal. This scheme is very efficient, because it reduces the number of process switches in a call, but the user has no control during run time of the callee side processes.

In *explicit* mode, one of the most common statements is the `accept` statement that accepts a procedure name, or a set of procedures, as input and blocks until a call to one of the specified procedures is made. The `select` statement allows the callee to express Boolean expressions (*guards*) that must be attended to before the callee accepts a call for the corresponding procedure. Most of the languages that do have a select construct on the callee side such as Ada [20], do not allow RPC calls to be part of the Boolean expressions, that is, act as a guard. A discussion about situations in which the absence of such a construct causes loss of expressiveness, of robustness, and even of the wanted semantics, is presented in [25]. An expressive extension of Ada's select construct that solve the problem in an elegant way is also described. Having RPC calls as guards in select constructs is an advantage in situations such as: *conjunctive transmission* (*broadcasting*), *disjunctive transmission*, *pipelining*, *transmission stopped by a signal*, and so on. The *broadcasting* occurs when the RPC caller gets some information that it wants to broadcast to a specific set of processes. It has the calls it is to make, but the order in which the callees are ready to answer the call is unknown. The caller does not want to block waiting for a busy callee to answer while it can be calling other callees and afterwards try the busy one again. The *disjunctive transmission* occurs when a caller wants to call *any* potential callees. The *pipelining* happens when processes $P_1, \ldots, P_n$ are to form a pipeline, where process P_i receives items from P_{i-1} and sends them to P_{i+1}. Each P_i is simultaneously ready to either accept an item from P_{i-1} or to send an item to P_{i+1}, and is willing to choose by the availability of its neighbors. The *transmission stopped by a signal* occurs when mixed calls and accepts are necessary—for example, when you want a process to continuously make specific calls un-

til another process calls it when it is supposed to accept and stop calling.

Concert [90] presents a set of language supports for the handling of RPCs in the explicit mode, which is discussed in Section C of the Appendix.

6.3 Exception Handling

In a single thread of execution, there are two modes of operation: the whole (including caller and callee) program works, or the whole program fails. With RPC, different failure modes are introduced: either the callee or the caller malfunction, or the network fails. Even with good exception raising schemes, the user is forced to check for errors or exceptions where logically one should not be possible, making the RPC code different from the local code, and violating transparency.

Transparency is lost either because RPC has exceptions not present in the local case, or because of the different treatment in the RPC or local case of exceptions common to both. The programming convention in single machine programs is that, if an exception is to be notified to the caller, it should be defined in the interface module, while the others should be handled only by a debugger [13]. Based on this assertion, Cedar RPC [13] emulated the semantics of local calls for all RPC exceptions defined in the interfaces exported. The exception for call failure is raised by the runtime support if any communication difficulty should occur.

To prevent a call from tying up a valuable resource in the callee side, the callee should be notified of the caller's failure [86]. For a caller's failure, or any other abnormal termination, the resource should be freed automatically. The general problem of *garbage collection* in a distributed system could be very expensive. The NCA/RPC [36] exposes to the client exceptions that occur during the execution of RPCs. It also propagates interrupts occurring to the caller process, all the way to the callee process executing on the caller's behalf.

7 Execution-Time Components

The *runtime support* and *communication media support* constitute the execution time components of RPC. The *runtime support* consists of a set of libraries provided by the RPC system. Its a software layer between the other RPC components and the underlying architecture. It receives and enqueues calls to the callee stubs, sends the call messages generated by the caller stubs, guarantees that calls are executed in the correct order, and so on. The *communication media support* is the transport media that carries the messages. At the operating-system level, there are several possible configurations for an RPC call, as shown previously in Figure 3. Depending on the particular system and the underlying architecture, the communication media chosen for each particular configuration varies. Nevertheless, when the caller and callee threads reside on different machines, network transport support is needed. This support has been the subject matter of several RPC systems, and it is discussed in the following section.

7.1 Network Transport Support

For the network transport support, the primary question is which protocol is the most suitable for the RPC paradigm.

For better performance in the extended exchange case, TCP could be chosen, since it provides better flow and error control, with negligible processing overhead after the connection is established. It also provides the predictability required for RPC. On the other hand, it is typically expensive in terms of space and time. In space, it requires the maintenance of connection and state information, and, in time, during connection establishment, it requires usually three messages to set it up, and three more to tear it down. RPC systems usually have short periods of information exchange between caller and callee (the duration of a call), and a callee must be able to easily handle calls from hundreds of callers. Network implementations usually are strict about the number of concurrently open connections. Even without this restriction, it would be unacceptable for a callee to

maintain information about such a large number of connections.

Datagram-oriented network services (such as UDP) do not provide for all the security an RPC mechanism needs, but are inexpensive and are available in any network.

In general, RPC-based systems make use of the datagram service provided by the network, and they implement a transport protocol on the top of the datagram, which meets RPC requirements. This protocol, usually, performs the flow and error control of TCP protocols, but the connection establishment mechanism is light weight.

The designers of the Cedar RPC Facility [13] found that the design and implementation of a transport protocol specifically for RPC would be a great source of performance gains. First, the elapsed real-time between initiating a call and getting results should be minimized. The adequacy for bulk data transfer, where most of the time is spent transferring data, was not a major concern in the project. Second, the amount of state information per connection, and the handshaking protocol to establish one, should be minimized. Finally, at-most-once semantics should be used.

The scheme adopted bypasses the software layers that correspond to the normal layers of a protocol. The communication cost was minimized when all the arguments of a call fit in a single packet. The protocol works as follows. To issue a call, the caller sends a packet containing a *call identifier*, a procedure identifier, and the arguments. The *call identifier* consists of the calling machine identifier, a machine-relative identifier of the calling process, and a sequence number. The pair [`machine identifier, process`] is named *activity*. Each *activity* has at most one outstanding remote call at any time. The sequence number must be monotonic for each activity. The *call identifier* allows the caller to identify the result packet, and the callee to eliminate duplicate call packets. Subsequent packets are used for implicit acknowledgment of previous packets. Connection state information is simply the shared information between an *activity* on a calling machine, and the RPC runtime support on the callee machine, making possible a light-weight connection management. When the connection is idle, the caller has only a counter, the sequence number of its next call, and the callee has the *call identifier* of the last call. When the arguments or results are too large to fit in a single packet, multiple packets are sent alternatively by the caller and callee. The caller sends data packets, and the callee responds with acknowledgments both during arguments transmission, and the reverse, during results transmission. This allows the implementation to use only one packet buffer at each end for a call, and avoids buffering and flow control strategies in normal-bulk data transfer protocols. In this way, costs for establishing and terminating connections are avoided, and the cost of maintenance is minimized.

The NCA/RPC [22] protocol is robust enough to be built on top of an unreliable network service and yet to deal with lost, duplicated, out-of-order, long-delayed messages that cause callee side processes to fail. It also guarantees at-most-once semantics when necessary. A similar approach to the Cedar project is used. The connection information kept between processes constitutes, basically, a last-call sequence number. This number can be queried from both ends for synchronization purposes through *ping* packets. The runtime support, though, uses the socket abstraction to hide the details of several protocols, allowing protocol independent code in this way. A pool of sockets is maintained for each process: allocated when a remote call is issued, and returned to the pool when the call is over. A similar end-to-end mechanism is implemented by the Athena RPC [70] and by NCS/RPC [22].

SUN RPC [79] allows the user to choose the transport protocol based on the user's needs. SUN RPCs require an RPC handler as the second argument. This handler is obtained by calling the *port mapper*, and passing the transport protocol type specified by the user, TCP, or UDP. When using TPC, there is no limit to the size of the arguments or result values. An integer at the beginning of every record determines the number of bytes in the record. With UDP, the total size of arguments must not generate a UDP packet that exceeds 8,192 bytes in length. The same holds for the returned values.

ASTRA [3] developed *RDTP/IP*, a *reliable datagram transport protocol*, that is built on

UDP. It provides a low-cost reliable transport, where calls and replies are guaranteed through a connectionless protocol. For flow and error control, it uses a *stop and wait* protocol. It also provides an *urgent datagram* option that causes RDTP to push the packet out to the network immediately. On the callee side, the *urgent datagram* is stored in a special buffer, and a signal is raised to the user. A user can receive urgent datagrams by specifying the *urgent datagram* option in receiving statements.

The *synchronized clock message protocol* [48] (*SCMP*) is a new protocol that guarantees at-most-once delivery of messages at low cost and is based on synchronized clocks over a distributed system. It assumes that every node has a clock and that the clocks of the nodes are loosely synchronized with some skew ϵ, less than a hundred milliseconds. Nodes carry out a *clock synchronization protocol* that ensure such precision. This synchronization protocol is being used for other purposes, such as for authentication and for capabilities that expire.

The protocol can be summarized as follows: every thread of control T has a *current time*, *T.time*, that is read from the clock of the node in which T is running. Every message has a timestamp, *m.ts*, that is T.time of the sending thread at the time *m* is created. Each message contains a *connection identifier*, *m.conn*, chosen by the client without consultation with the server. Each server maintains a *connection table*, *T.CT*, that is a mapping from connection identification to connection information. One piece of information is the timestamp of the last message accepted on that connection. Not all connections have an entry in T.CT. Any T is free to remove an entry T.CT[C] for a connection C from its table, if T.CT[C].ts $\leq$ T.time $-\rho$, where ρ is the interval during which a connection information must be retained. A callee also keeps the upper bound, *T.upper*, on the timestamps that are removed from the table. Because only old timestamps are removed from the table, T.upper $\leq$ T.time $-\rho$. The mechanism then distinguishes *new* from *duplicated* (*old*) messages, based on the notion of *bound* of a particular connection. On the callee side, the *bound* of a connection is the timestamp of the most recent previously accepted message, if the message's connection has an entry in the table. Otherwise, the global bound T.upper is used. Because no information is in the table for this connection, the last message on this connection, if there was any, had a time stamp $\leq$ T.upper. Provided that ρ is sufficiently large, and having T.upper $\leq$ T.time $-\rho$, the chances of mistakenly assuming a message as duplicate is very low.

The above protocol is nevertheless for callees that do not survive failures. For resilient callees, it is necessary to determine by the time of a message arrival, whether the message is a new one or a duplicate of a message that arrived before the failure. An estimate of the timestamp of the latest message that may have been received before the failure occurred. The system operates as if no message with a timestamp greater than the estimate was received before the failure, and therefore, messages older than the estimate are new. An upper bound on the timestamps of accepted messages is enforced and is named *T.latest*. After the failure, T.latest is used to initialize T.upper. Roughly speaking, a server periodically writes T.time to stable storage. After a failure, T.latest is the most recent value written to stable storage. For example, SCMP can be used by higher-level protocols to establish connections. It can be used as a security reinforcement mechanism that allows only *new* messages to be passed to higher-level mechanisms that use it. Performance measurements [48] show that at-most-once semantics for callers calling callees occasionally are achieved with about the same cost as UDP-based SUN RPC, and with significantly better performance than the TCP-based SUN RPC. If the caller makes several calls to the same callee, none major overhead cost is added over the UDP, and it performs better than the TCP-based RPC.

When the RPC involves processes connected through a wide-area network, the protocol must support location services and transparent communication among the distant connected processes. An adequate protocol for local-area networks would be inappropriated for wide-area networks, because the latter has a very high latency rate. Amoeba distributed operating system [85] introduced a session layer gateway to support the interconnection of local-area networks. Through a *publish* function,

target servers (callees) export their port and wide-area network address to other Amoeba sites. When receiving this information, each site installs a *server agent*. The *server agent* acts as a virtual server (callee) for all calls to that service, listening locally to requests to that port from that site. When issuing an RPC, the callee calls the server agent, which forwards the request across the wide-area network using the published address. When the request arrives at the remote Amoeba site, a *client agent* process is created there that acts as the virtual client (caller) of the target server and starts a local call to it. After execution, the target server sends the results to the client agent, which forwards the results through the wide-area network to the server agent. The server agent then completes the call, returning the results to the real client. Amoeba provides full protocol transparency for the user. For wide-area communication, it uses whatever protocol is available and without the client and server processes knowing about it. For local communication, it uses protocols optimized for local networks. The error recovery is very powerful in this model, because the client agent notifies the server agent, and the reverse, when a shutdown of the client occurs.

REX [57] is a remote execution protocol that extends the RPC concept to a wider range of remote process-to-process interactions for an object-oriented distributed processing architecture.

8 Supporting and Extending the RPC Mechanism

This section discusses several implementation techniques to enhance the RPC mechanism, and extensions added to the concept. The extensions try to model the RPC paradigm so that it will be more efficient, or more suitable, for particular applications.

8.1 Process Management

Of foremost concern in designing and implementing an RPC mechanism are three factors: the language that is extended to support RPC, the operating system in which the mechanism is built, and the networking latency of the underlying implementation.

An operating system should be capable of supporting several threads of control [82]. Single-threaded environments do not provide for the performance and ability for concurrent access of shared information required for the implementation of the mechanism. An attempt at implementing an RPC prototype as part of the MIT's Project Athena, around 1986, on the top of 4.2 BSD, is described in [70]. There were no support for more than one thread of execution within a single address space. The communication protocol was a user mode request and response layered on UDP (UDP-RR). The 4.2BSD does not provide the low latency required. The process scheduling algorithm is nonpreemptive, which tends to increase the call latency.

One of the most useful properties of a language that is to be extended with RPC, or any synchronous mechanism, is the support of process management. This topic is discussed in detail in [46], where the limitations of synchronous communication with static process structure are presented. Two situations in which a process could block are identified: *local delay* and *remote delay*. When the current activity needs a local resource that is unavailable, the activity is said to be blocking because of a *local delay*. A callee in this situation should suspend the execution of this call, and turn its attention to requests from other callers. If the current activity makes a remote call to other activity and is delayed, the other activity is blocked because of a *remote delay*. A *remote delay* can actually be a communication delay, which can be large in some networks. It can also happen because the called activity is busy with another request, or must perform considerable computing in response to the request. The caller process, while waiting, should be able to work on something else. The user should be able to design several threads of execution inside a process. This provides the operating system with alternative functions to execute while that call, on the callee or caller side, is blocked.

Several alternative ways to avoid or overcome the blocking in the absence of dynamic process creation are described as follows and the de-

ficiencies inherited in the alternative ways are analyzed [46]. Examples of some solutions are:

- *Refusal:* a callee that accepts a request and is unable to execute it, because of a delay, returns an exception to the caller to indicate that the caller should try again later.
- *Nested accept:* a callee that encounters a local delay accepts another request in a nested accept statement. This scheme requires that the new call is completed before the old call is completed. This may result in an ordering that may not correspond to the needs of applications.
- *Task families:* instead of using a single process to implement a callee, a family of identical processes is used. These processes together constitute the callee, and synchronize with one another in their use of common data.
- *Early reply:* simulate asynchronous primitives with synchronous primitives. A callee that accepts a call from a caller simply records the request and returns an immediate acknowledgment.
- *Dynamic task creation in the higher-level callee:* the higher-level server uses early reply to communicate with callers. When it receives a request, it creates a subsidiary task (or allocates an existing task) and returns the name of the task to the caller. This would handle remote delay.

A synchronization mechanism may provide adequate expressive power to handle local delays only if it permits scheduling decisions to be made on the basis of the the name of the called operation, the order in which requests are received, the arguments of the call, and the state of the resource [46].

NIL [76] was one of the primary efforts in having processes as the main program structuring construct at the language level, which are both units of independent activity, and units of data ownership. Processes can be dynamically created and destroyed in groups called *components*, and each component accepts a list of *creation-time parameters*. These parameters are used to pass initial data to an *initialization routine* within each process being created. The choice of processes to load into a component is made at run-time. Hermes [75] and Concert [90] are languages that came afterwards, also based upon the process model.

8.2 Fault Tolerance

The *fault tolerance* and *recoverability* are primary issues in distributed environments. A distributed environment provides a decentralized computing environment. The failure of a single node in such environments represents only a *partial failure*, that is, the damage (loss) is confined to the node that failed. The rest of the system should be able to continue processing. A system is *fault tolerant* [8] if it continues functioning properly in the face of processor (machine) failures, allowing distributed applications to proceed with their execution and allowing users to continue using the system. The *recoverability* refers to the ability to recover from a failure, partial or not.

There are three sources of RPC failures:

- *Network failure:* The network is down, or there is a network partition. The caller and callee cannot send, or receive any data.
- *Caller site failure:* The caller process or the caller host fails.
- *Callee site failure:* The callee process or the callee host fails. This might cause the caller to be indefinitely suspended while waiting for a response.

The system may hide the distributed environment, or other underlying details from the user, thus providing *transparency*. The system can provide *location transparency*, meaning that the user does not have to be aware of the machine boundaries and the physical locations of a process to make an invocation on it. On the other hand, the simplest approach to handling failures is to ignore them, or to let the user take care of them.

The fault model for *node failures* is as follows: either a node works according to its specifications, or it stops working (*fails*). After

a failure, a node is repaired within a finite amount of time and made active again.

Usually, the receipt of a reply message from the callee process constitutes a *normal termination* of a call. A *normal termination* is possible if [58]:

- No communication or node failures occur during the call.
- The RPC mechanism can handle a fixed finite number of communication failures.
- The RPC mechanism can handle a fixed number of communication failures and callee node failures.
- The RPC mechanism can handle a fixed number of communication failures and callee node failures and there is tolerance to a fixed finite number of caller node failures.

The appropriate choice of fault tolerance capabilities and the semantics under normal and abnormal situations, are among the most important decisions to be taken in an RPC design [58]. A *correctness criterion* for an RPC implementation is described in [58]. Let C_i denote a call, and W_i represent the corresponding computation invoked at the callee side. Let C_i and C_j be any two calls made such that: (1) C_j happens after C_i (denoted by C_i then C_j), and (2) computations W_i and W_j share some data such that W_i and/or W_j modify the shared data. The *correctness criterion* (*CR*) then establishes that an RPC implementation must meet in the presence of failures the following CR:

CR: C_i then C_j implies W_i then W_j

A call is terminated *abnormally* if the termination occurs because no reply message is received from the callee. Network protocols typically employ *time-outs* to prevent a process that is waiting for a message from being held up indefinitely.

If the call is terminated abnormally (the time-out expires), there are four mutually exclusive situations to consider:

- The callee did not receive the request, but it was made.
- The caller did not receive the reply message, but it was sent.
- The callee failed during the execution of the respective call. The callee either resumes the execution after failure recovery, or does not resume the execution even after starting again.
- The callee is still executing the call, but the time-out was issued too soon.

There are two situations in which the abnormal termination can cause the caller to issue the same call twice. If the time-out is issued too soon for a particular call, the caller may erroneously assume that the call could not be executed, and might decide to call another callee process, while the former one is still processing the call. If the two executions share some data, the computations can interfere with each other. Another situation can occur when the caller recovers from a failure but does not know whether the call was already issued or not, and makes the call again. Executions that are failed are *orphans*, and many schemes have been devised for detecting and treating orphans [40, 53].

Network failures can be classified [58] as follows:

- A message transmitted from a node does not reach its intended destination (a *communication failure*).
- Messages are not received in the same order as they are sent.
- A message is corrupted during its transmission.
- A message is replicated during its transmission.

There are well-known mechanisms (based on checksums and sequence numbers) that a receiver can use to treat messages that arrive out of order, are corrupted, or that are copies of previously received messages. Therefore, *network failure* is the treatment of communication failures only. If the message handling facility is such that messages occasionally get lost, a caller would be justified in resending a message when it suspects a loss. This can sometimes

result in more than one execution at the callee side.

When a process survives failures on its node, the process is a *resilent process*. It might require maintenance of information in a log on a nonvolatile storage medium.

Because of the difficulties in achieving conventional call semantics for remote calls, because of problems with argument passing and problems with failures, some works [31] argue that remote procedures should be treated differently from the start, resulting in a completely nontransparent RPC mechanism.

8.2.1 Atomicity

A major design decision of any RPC mechanism is the choice of the *call semantics* of an RPC in the presence of failures. Some systems adopt the notion that RPC should have only last-of-many semantics when failures occurred to model procedure calls in a centralized systems [53]. In Argus [45], RPC has at-most-once semantics and malfunctions are treated by exception handling. The Cedar [13] implementation enforces at-most-once semantics by sending *probe* messages, and by using Mesa's powerful exception handling mechanism. Other systems [67, 80] follow the idea that RPC should have exactly-once semantics to be useful.

An implementation of an atomic RPC mechanism, which maintains totality and serializability, is presented in [42]. Serializability is guaranteed through concurrency control by attaching a call graph path identifier to each message representing a procedure call. Each procedure keeps the message path of the last accepted call. This path is used to make comparisons with incoming message paths, and only calls that can be serialized are accepted. In this system, procedures are separated entities. A procedure's data can be either *static* or *automatic*. The *automatic* data is deleted after a procedure's activation, while *static* data is retained. Associated states of static variables are saved in backup processors to survive failures.

Because of concurrency control and to maintain atomicity, some calls to procedures with static data can be rejected. For instance, suppose a procedure A that concurrently invokes calls on procedures B and C, both of which call procedures D, which has static data (Figure 7). If B calls D first, C's call to D should be rejected until B returns to A. If C's call is accepted, and if B calls D again later, B's and C's effects on D cannot be serialized, violating the atomicity requirement.

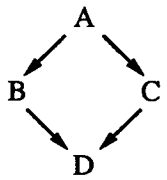

Figure 7: Concurrent Call of D by B and C

Procedure calls can be *atomic*, that is, total and serializable, or *nonatomic*. Exactly-once semantics are guaranteed for atomic calls, while only at-most-once semantics are assured for nonatomic ones.

Calls invoked by the callees of an atomic call are also executed as atomic calls. The totality property requires that when a caller malfunctions and recovers to an old state, all the states of its callees must also be restored. If a callee fails and recovers, its caller is not affected, but the call must be repeated. The algorithm recovers a failed procedure's data state to its state after the last time it finished an atomic call. A procedure that was called by a failed atomic procedure is recovered to the state before the failed atomic procedure made the first direct or indirect call to the procedure. The backup states for a procedure are *Associated States* (*AS*). An AS is the procedure's data state (static variable and process identification) that is tagged with a version number, which is increased by one every time a new AS is created. An AS is created on procedure initiation and updated every time after the procedure returns from an atomic call.

A *Transactional RPC* design and implementation is presented in [23]. The design consists of a preprocessor, a small extra runtime library, and the usual RPC components. The preprocessor accepts interfaces written in DCE IDL and generates, for each procedure, a *shadow client stub* and a *shadow manager stub* (shadow server stub). It also generates an *internal interface*, which contains a new version of the procedures declared in the original interface,

where an extra in-out argument is inserted. This extra argument contains the data information needed for the transaction system. The internal interface is the input to the stub generator.

When issuing the RPC, the user calls the *shadow client stub*, which communicates with the transaction subsystem. The shadow client registers the call, obtains some registering data (*piggyback data*), and updates the log file. After that, it calls the shadow manager stub generated for the internal interface and passes the piggyback data as the extra argument. On other side, the manager stub receives the call. It recovers the piggyback data argument, and interacts with the transaction system to communicate the acceptance of the request. Thereafter, it calls the real caller stub with the reminder of the arguments. A similar protocol is used when the call is finished. The shadow server interacts with the transaction system communicating the completion of the call, and obtaining some more piggyback data. It then passes the piggyback data and the procedure results back to the client shadow. The client shadow, once again, interacts with the transaction system by passing this piggyback data, and then returns the results of the procedure to the original caller. The user interfaces are not modified, and the user with transactional applications benefits in a completely transparent manner.

A reliable RPC mechanism that adopts the *exactly once* semantics is described in [67]. At the RPC system language level, the user can specify additional information that enables the underlying process to manage the receipt and sending of messages. Orphans are not treated at this level. The level immediately above this requires that all programs be atomic actions with the *all or nothing* property. Executing callees in an atomic way guarantees that the repeated execution at the callee side is performed in a logically serial order with orphan actions terminating without producing any results. The syntax transparency is lost, and an RPC call requires, at least, the name of the callee's host and a status and a time-out argument. The time-out specifies how long the caller is willing to wait for a response to his request. The status indicates whether the call was executed without problems, whether the call was not done, whether the callee was absent or could not execute the call. The system assigns sequence numbers (SN) to messages. SNs are unique over the entire system. A callee maintains only the largest SN received in a failure proof storage. All retry messages are sent by a sender with the same SN as the original message. A callee accepts only messages whose SN is greater than the current value of the last largest SN. A similar approach is provided at the caller's side. For the generation of network-wide unique sequence numbers, the loosely synchronized clock approach is used. Each node is equipped with a clock and is also assigned a unique node number. A sequence number at each node is the current clock value concatenated with the node number.

8.2.2 Replication

Some systems use replication to support fault tolerance against hardware failure. Schemes that use replication can be classified in two categories: *primary standby* and *modular redundancy*. In the *primary standby* approach one copy is the active *primary* copy and the others are *passive secondary* copies. The primary copy receives all the requests and is responsible for providing the responses. It also updates periodically the checkpoints of the passive secondary copies, so that they can assume the role of primary copy, if this is the case. If the primary copy fails, a prespecified secondary copy assumes its role. The major drawback of such a scheme is the considerable system support required for checkpointing and message recovery. In *modular redundancy* scheme, all the replicated copies receive the requests, perform the operation, and send the results. There is no distinction made between the copies. The caller usually waits for all the results to be returned, and may employ voting to decide the correct result. The major disadvantage of this scheme is the high overhead of issuing so many requests and so many responses. If there are m requests and n copies of the requested service, the number of messages needed to complete the request is $O(m \times n)$.

The *Replicated Procedure Call* [18] is an example of modular redundancy-based mecha-

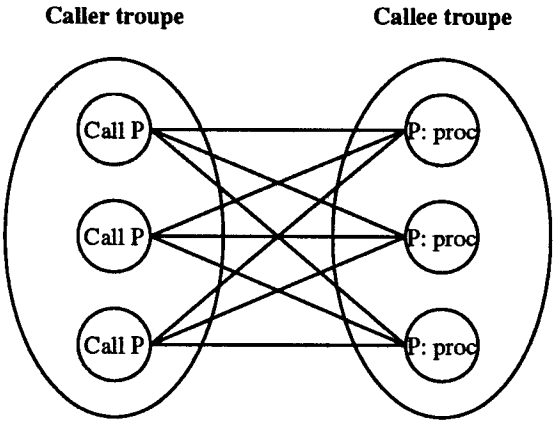

Figure 8: Replicated Procedure Call

nism for RPC-based environments. The set of replicas (instances) of a program module is a *troupe*. A *replicated procedure call* is made from a caller troupe to a callee troupe. Each member of the caller troupe makes a *one-to-many call* to the callee troupe, and each callee member troupe handles *many-to-one call* from the caller troupe as shown in Figure 8. Each member of the callee troupe only performs the requested procedure once, while each member of the caller troupe receives several results. To the programmer this looks like normal procedure calls. Distributed applications continue to function normally despite failure if at least one member of each troupe survives. The degree of replication can be adjusted to achieve varying levels of reliability. Increasing the number of nodes spanned by a troupe increases its resilience to failures.

To issue a *replicated procedure call*, the caller's side of the protocol sends the same call message, with the same sequence number, to each member of the callee troupe. On the callee's side, a callee member receives call messages from each caller's troupe member, performs the procedure once, and sends the same result to each member of the caller's troupe. For this call, each troupe has a unique troupe id, assigned by a binding agent, and each message contains the caller's troupe id. When a request arrives, the callee checks the members of the caller's troupe id, and determines from whom to expect call messages as part of the *many-to-one call*. The callee also needs a way of grouping the related requests, that is, to decide whether the requests are unrelated or are part of the same replicated call, because the requests may not be synchronized or identical. The system does not require *complete determinism*, that is, that each member of a troupe reaches the same stable state through the execution of the same procedures with the same arguments in the same order. The system is based on the notion of *state equivalence relation* between modules, and *parameter equivalence* and *result equivalence* among procedure calls. Procedure executions in a module have an equivalence relation induced by the relations. Two executions are equivalent if: (1) the results are result equivalent, (2) the new states are state equivalent, (3) the traces are identical up to parameter equivalence of the remote procedures called. A module is *deterministic up to equivalence* if: (1) all executions of a procedure with a specific set of arguments in a specific state are equivalent, or (2) whenever a parameter is equivalent to another parameter, and a state is also equivalent to another state, all executions of a specific procedure in the former state with the former arguments are equivalent to the executions of the same procedure with the second argument in the second state.

To assure the *deterministic up to equivalence* relation, the notion of call stack is extended to *distribute call stack*, which consists of the sequence of contexts, possibly across machine boundaries, that were entered and not returned from yet. At the base of the stack is the top-level context that originated the chain of stacks. The *distribute call stack* is, once again, extended to *replicated call stack*, in which an entry for an active procedure consists of a set of contexts, one from each member of the troupe implementing that procedure. Given an active replicated procedure call, the replicated call stack can be traced back, and a unique troupe can be found at the base level. This is the *root troupe* of the replicated procedure call.

In this way, a *root id* is added to each call message. A *root id* contains the troupe id of the root troupe of the call, and the sequence number of the root troupe's original replicated call. The *root id* is a transaction identifier, and whenever a callee makes a replicated call on behalf of a caller, it propagates the root id of

the call that it is currently performing. The *root ids* have the essential property that two or more call messages arriving at a callee have the same root id if they are part of the same replicated call. This allows a callee to handle *many-to-one* calls.

A combination of the modular redundancy approach and the primary standby is described in [88], which provides for an RPC mechanism tolerant to hardware failures. Whereas all the replicated copies execute each call concurrently, as in the modular scheme, the caller only makes one call, which is then transmitted to the others. The mechanism exposes to the user a *cluster* abstraction, which constitutes a group of replicated incarnations of a procedure. The incarnations are identical, but reside in different nodes. An RPC call is issued by a *caller cluster*, and the *callee cluster* is the one that executes the call. The caller process, the process that starts the call, is not replicated, and is a special instance of a caller cluster with only one instance. In recursive and nested calls, a cluster can function both as a caller and as a callee.

There are *primary* and *secondary* incarnations of a procedure. During cluster creation time, an incarnation is designated *primary*, and is responsible for handling the communication between clusters. All others are considered *secondary*. If the primary incarnation fails, a secondary incarnation assumes its function and becomes the primary incarnation. A secondary incarnation can only interact with other cluster's processes, if all its superior incarnations fail and it has become primary. The incarnation hierarchy in a cluster is determined when the service is created. All incarnations play an active role: all the incarnations in the callee cluster execute a call, and every incarnation in a caller cluster receives a copy of the result. In the absence of failure, though, the caller makes a single call and only one copy of the result is sent to the caller cluster. The system assumes that a cluster is deterministic: when receiving the same call each of its incarnations produces the same result and has the same side-effects. The callee procedures must be idempotent.

8.2.3 Orphan Detection

Rajdoot [58] uses three built-in mechanisms to cope with orphans in an efficient way. It adopts *exactly once semantics*. For callee failure, the call is guaranteed to be terminated abnormally. There are three orphan handling mechanisms:

- ($M1$:) If a client call is terminated abnormally, any computations that the call may have generated are also terminated.
- ($M2$:) Consider a node that fails and makes a remote call to node C after recovery. If C has any orphans because of the caller's failure, they are terminated before the execution of the call starts at C.
- ($M3$:) If the node remains down, or if after recovery never makes calls to C, any orphans on C are detected and removed within a finite amount of time.

To ensure $M1$, every call has a deadline (time-out) argument that indicates to the callee the maximum time available for execution. When the deadline expires, the callee terminates the execution and the call is terminated abnormally. To ensure $M2$, every node maintains a *crashcount* in permanent storage. The *crashcount* is a counter that is incremented immediately after a node recovers from a failure. Also in stable storage, a node maintains a table of crashcount values for callers that made calls to it. Every call request contains the caller's crashcount value. If the value in a request is greater than the value in the callee's table for that caller, orphans may be at the callee. These orphans are terminated before the callee proceeds with the call. To ensure $M3$, every node runs a *terminator* process that occasionally checks the crashcount values of other nodes by sending messages to them and receiving replies, and then terminates any orphans when it detects failures.

The RPC call syntax is as follows, where parameters and results are passed by values:

```
RPC(server:  ...; call:...;
    time-out:...; retry:...;
    var reply:...;var rpc_status:...);
```

The first parameter specifies the callee's host. The second parameter, which is the name of the

function and the arguments. The **retry** indicates the number of times that the call is to be retried (the default value being zero). If after issuing a call, no reply is received within the duration specified by **time-out**, the call is reissued. The process is repeated a maximum of **retry** times. The *manager processes*, which are run by each process and accessed via a well-known address, are responsible for creating callee processes that execute remote calls of a client. All remote calls of the client are directed to the callee process.

SUN RPC does not specify the call semantics to be supported and has no provision for orphan treatment. The transport protocol chosen at the time that the RPC handler is acquired determines the call semantics. Similarly, Xerox Courier RPC [87] appears to support exactly once semantics, but its description is not precise about its fault tolerance capabilities and no support for orphan treatment is provided.

8.3 Debugging and Monitoring RPC Systems

Concurrency and communication among the concurrent parts of a distributed program greatly increase the difficulty associated with program debugging. For this reason, a system for prototyping and monitoring RPCs was designed [91]. From programmer-written callee definition files, the system can prototype callee programs, caller driver programs, interface description files written in NIDL for each callee, stubs, and prototype programs of RPC procedures for each callee.

The system consists of two parts: a *prototyping generator* and a *monitor*. The *prototyping generator* gets the callees' definition files, and generates the above-mentioned files. It consists of a set of specialized generators that provide for the gradual development of the components of a distributed environment. The *server program generator* generates a callee stub and a callee program for each callee definition file. The *sequential client program generator* generates, for each callee, a caller stub and a caller driver program. The *parallel client program generator* generates a driver program for each callee which can make use of any number of remote procedures provided by the callee concurrently. For a group of callees, the *distributed client program generator* generates a driver program that can make use of any number of remote procedures provided by these callees.

The produced programs are monitored by the *monitor* when the *debug* option is on, and during prototyping generation. The *monitor* has a *controller* and a group of *managing servers*. The *controller* offers a user interface and has a *filter*. Each host that has monitored program parts on it has a *managing server* consisting of a *server* (*MS*) and an *event database*.

The monitoring is done in three steps. During the *monitoring step*, all *events* that occur on one host are monitored by the local MS and recorded into the local event database. An *event* is an RPC and execution, a process creation (through a fork), and termination, or any combined event defined by the user through the *Event Definition File* (*EDF*). During the second step, the *ordering step*, the events saved in the local event databases are ordered through messages interchanged among the MSs. The controllers can be invoked by any host. The last step, the *replaying step*, consists of combining the results on all related event databases. The *filter* presents the execution trace of the distributed program to the user. After the program is debugged, the user needs only to program the RPC body and the application interface.

8.4 Asymmetry

Another aspect of the RPC mechanism that might not be appropriate for certain applications is the *asymmetry* inherited by the RPC model. In the RPC model, the caller can choose the particular callee, or at least the callee's host machine. Seldom can the callee make any restrictions about the caller, because the callee cannot distinguish among callers to provide them with different levels of service or to extend to them different levels of trust. Callees must deal with callers that they do not understand, and certainly cannot trust [64].

This issue is referred as *naming* (or *addressing*) of the parties involved in an interaction [8]. A communication scheme based on direct naming is *symmetric* if both the sender and re-

ceiver can name each other. In an *asymmetric* scheme, only the sender names the receiver. When the interprocess communication mechanism provides for explicit message receipt, the receiver has more control over the acceptance. The receiver can be in many different *states* and accept different types of messages in each state. The *indirect naming* involves an intermediate object, usually called a *mailbox*, to which the sender directs its message and to which the receiver listens.

In LYNX [64], a link is provided as a built-in data type, and is used to represent a resource. Although the semantics of a call over a link are the same as an RPC, both ends of the call can be moved. Callees not only can specify the set of callers that might be connected to the other side of the link, but also are free to rearrange their interconnection to meet the needs of a changing user community, and to control access to the resources they provide.

Also related to symmetry is the *nondeterminism* issue. A process may want to wait for information from any of a group of other processes, rather than from one specific process. In some situations, it is not known in advance the member (or members) of the group that will have its information available first. Such behavior is nondeterministic.

On the other extreme, communication through shared data is anonymous. A process accessing data does not know or care who sent the data.

Another criticism of the RPC model is the *passiveness* of the callee [82]. Except for the results, a callee is not able to initiate action to signal the caller. The point is that the procedure call model is asymmetric and the callee has no elegant way of communicating or awaking the callers. One example [17] where the callee should call the caller is when the network server (callee) needs to signal to an upper layer in a protocol, or when a window manager server needs to respond to user print.

The procedure mechanism provides a synchronous interface to call *downward* through successive layers of abstraction. RPCs extend the mechanism and allow the layers to reside in different address spaces. The *distributed UpCalls* mechanism [17] is a facility which allows a lower level of abstraction to pass information to a higher level of abstraction in an elegant way. It is implemented as part of a server structuring system called CLAM. Users can layer *abstractions* in caller processes (statically bound) or dynamically load *layers* into the callee process. Consider two abstractions P and Q in different layers with P in a higher layer than Q. The P passes to Q pointers to functions in which P is willing to accept UpCalls. To pass pointers, P calls a Q registering function, with the function pointers as arguments, which is similar to exporting in normal RPC. When an event occurs that requires an UpCall to be made, Q decides the higher level of abstraction that should receive the call, which could be P. The P and Q can be in the same address space (local UpCalls) or in different ones (remote UpCalls). This is a different way of expressing, for instance, exception handling.

8.5 Lack of Parallelism

The inherited lack of parallelism between the caller and the callee, in an RPC model, is another feature that deserved a lot of attention by the scientific community. With RPC, while the callee is active, the caller is always idle and waiting for the response [82]. Parallelism is not possible. To gain performance, the caller should continue computing while the callee is working. Related to the above issue, is the fact that there is no way of *interleaving* between the caller and the callee [82].

A caller can only have one outstanding call at a time. The RPC paradigm is inherently a two-party interaction [82]. With the large number of available processors possible in a distributed environment, the caller is actually forbidden by the mechanism to use such parallelism. Many distributed applications could benefit from concurrent access to multiple callees [86].

A caller may wait to make several calls simultaneously. This is *bulk data* situations for an RPC environment.

A solution would be to build RPC-based applications with several threads, so that, while waiting for a call, only that call's thread blocks and the others proceed being executed. This solution does not scale well [3]. Since the cumulative cost of thread creation, context switching, and thread destruction can be too great

in large distributed environments, where the number of RPC calls grows and shrink dynamically. Moreover, threads are not universally supported.

The interactions between a caller and a callee can be classified as follows [3]:

- *Intermittent exchange:* the caller makes a few intermittent request-response type calls to the callee.
- *Extended exchange:* the caller is either involved in bulk data transfer, or makes many request-response type calls to a callee.

Several solutions have been proposed to extend the RPC concept, so that it is both more efficient for bulk data transfer, and so that it allows for asynchronous communication. All solutions try to maintain as much as possible the semantics of a procedure call.

A *call buffering* approach [30] is a mechanism that provides a *call buffering server*, where the requests and replies are stored (buffered). An RPC call results in a call to the *buffering server*, which stores the request, the caller names and the callee name. After that, the caller makes periodic requests to the *call buffering server* to check whether the call is executed. In the meantime, callees poll the *call buffer server* to check whether there are any calls waiting. If so, the arguments are retrieved, the call is executed, and the callee calls the *call buffer server* back to store the results. The next time the caller pools the buffer, the results can be retrieved.

In the Mercury System [45], a mechanism is proposed that generalizes and unifies RPC and byte-stream communication mechanisms. The mechanism supports efficient communication, is language independent and is called *call stream* or *stream*. To add the mechanism to a particular language, a few *extensions* may be added to the language. Such extensions are *language veneer*. Veneers for three languages have already been provided: Argus, C, and Lisp.

A *call stream* connects two components of a distributed program. The component *sender* makes calls to the component *receiver*. The next call can be made before the reply to the previous call received. The mechanism can deliver the calls in the order they are made, and deliver the replies from the sender in the order of the corresponding calls. The purpose of the proposed mechanism is to achieve high throughput where calls are buffered and flushed when convenient. Low latency can be achieved by explicitly flushing the calls. The *call stream* relies solely on a specific, reliable byte-stream transport such as TCP, making it more suitable for bulk data transfer. The use of TCP leads to higher overhead for most transactional applications in which a request-response protocol is more appropriate.

The mechanism offers three types of calls:

- *Ordinary RPCs:* the receiver can make another call only after receiving the reply.
- *Stream calls:* the sender can make more calls before receiving the reply. To the user, the sequence of calls has the same effect as if the user had waited to receive the nth reply before doing the $(n+1)$-th call. The system delivers calls and replies in the correct order.
- *Sends:* the sender is not interested in the reply.

The receiver provides a single interface, and does not distinguish among the three kinds of calls. The underlying system takes care of buffering messages where appropriate, and delivering calls in the correct order. At the moment of the call, the senders can choose independently the particular type of call desired.

To minimize the delay for the ordinary RPC mode, requests and replies are sent over the network immediately, that is, as soon as they are issued or available. On the other hand, stream calls and sends are buffered and sent when convenient.

The paradigm also introduces the abstraction of *entities*. An entity has a set of *port groups* and *activities* (processes). The *port groups* group ports for sequencing purposes. On the sender side, entities have *agents* and a port group that define the sending end of a stream. In this way, the user and the system establish the sequencing in which calls from ports of the same group, calls from different streams but from the same entity, or calls from different streams and different entities must be executed.

In the CRONUS [3] system, *Future* is designed only for low latency, and yet the order of execution in Future may vary with the order called. Stream and Future do not optimize for intramachine calls.

Another extension to the RPC paradigm was the new data type introduced in the work described in [43], *promise*. The goal was to preserve the semantics of an RPC call, yet prevent the caller from blocking until the moment the caller really needs the values returned by the call. An RPC is declared as returning a *promise*, a place holder for a value that will exist in the future. The caller issues the call, and continues executing in parallel with the callee. When the callee returns the results, they are stored in the *promise*, where they can be *claimed* by the caller when needed. The system guarantees that the calls are executed in the order that they were called, even if the result of call $i+1$ is claimed before the result of call i, if this result is ever claimed. The programmer explicitly *claims* the result, and the introduction of this data type shows the user the distinction between call time and claiming time. Functions are restricted to have only argument passing by value and return only one value that constitutes the promise value.

The extension is language independent and the abstraction can be incorporated in any language. An RPC mechanism could implement the idea behind the mechanism in a transparent way, confining the problem at the implementation level, but not at the language level. The system could do some data flow analysis in the program and allow the parallel execution of caller and callees, blocking the caller only when values that are expected to be affected by RPC calls are claimed. The complication here is to take care of pointer-based arguments that are modified as a side-effect of the remote execution.

ASTRA [3] tries to combine low-latency and high-throughput communication into a single asynchronous RPC model, and provides the user with the flexibility of choosing among different transport mechanisms at bind time. It uses the methodology that the programmer user knows better, letting the programmer decide whether a specific call requires low latency or high throughput. The system does all the optimizations necessary. The caller does not block during a call, and the replies can be claimed when needed similar to promises. All calls are executed in the same order as called. If the reply is not available when the caller claims it, the caller can unblock the receive operation by specifying a no delay option. ASTRA also provides optimized intramachines calls, bypassing the data conversion and network communication. For binding it uses a *superserver daemon* that must reside on every machine that runs a callee process, and also detects whether a particular server is still active. The major drawback of such a design is the loss of transparency between local and remote procedure calls.

Another new data type, *Pipes* [29], proposes to combine the advantages of RPC with the efficient transfer of bulk data. It tries to combine low-latency and high-throughput communication into a single framework, allowing for bulk data incremental results to be efficiently passed in a type safe manner. RPCs are *first-class* values, that is, have the same status as any other values. They can be the value of an expression, or they can be passed as an argument. Therefore, they can be freely exchanged among nodes. Unlike procedure calls, pipe calls do not return values and do not block a caller. Pipes are optimized for maximum throughput, where RPCs are optimized for minimum latency. RPC and pipes are *channels* used in the same manner as local procedures. The node that makes the call is the *source node*, and the node that processes calls made on a channel is the channel's *sink node*. For timing and sequencing among calls, channels can be collected into a *channel group*. A *channel group* is a set of pipes and procedures that share the same sink node and that observe a sequential ordering constraint for a source process. This ordering constraint guarantees that calls made by a process on the members of a channel group are processed in the order that they were made. The user makes the grouping.

A survey of asynchronous remote procedure calls can be found in [4].

8.6 Security

A major problem introduced by an external communication network is the providing of data integrity and security in such an open communication network. Security in an RPC mechanism [71] involves several issues:

- *Authentication:* To verify the identity of each caller.
- *Availability:* To ensure that callee access cannot be maliciously interrupted.
- *Secrecy:* To ensure that callee information is disclosed only to authorized callers.
- *Integrity:* To ensure that callee information is not destroyed.

Systems treat these problems differently, providing more insight or priority according to their most suitable applications.

The Cedar RPC Facility [13] uses the Grapevine [12, 62] Distributed Database as an authentication service or key distribution center for data encryption. It treats the RPC transport protocol as the level of abstraction at which to apply end-to-end authentication and secure transmission measures [61]. The Andrew system [61] independently chose the same approach.

Andrew's RPC mechanism [61] supports security. When a caller wants to communicate with a callee, it calls a *bind* operation. The *binding* sets up a logical *connection* at one of the four levels offered by the system:

- `OpenKimono`: the information is neither authenticated nor encrypted.
- `AuthOnly`: the information is authenticated, but not encrypted.
- `HeadersOnly`: the information is authenticated, and the RPC packet headers, but not bodies, are encrypted.
- `Secure`: the information is authenticated, and each RPC packet is fully encrypted.

When establishing the connection, the caller also specifies the kind of encryption to be used. The callee rejects the kinds of encryption that it cannot handle. The bind operation involves a 3-phase handshake between the caller and the callee. By the end of the operation, the caller and the callee share one *handshake key*. The caller is assured that the callee can derive this key from its identification, and the callee is assured that the caller possesses the correct handshake key. The RPC mechanism does not have access to the format of a caller's identification, nor the way in which the key can be derived from this identification. All this functionality is hidden by function pointers with which the security mechanism supplies the RPC runtime system. The code for the details of the authentication handshake is small, self-contained, and can be treated as a black box, allowing for alternative mutual authentication techniques to be substituted with relative easy. A call to an *unbind* operation terminates the connection, and destroys every state associated with that connection.

One way of guaranteeing security in RPC systems, is to enhance the transport layer protocol at the implementation level and to extend the call interface at the user level, as in [11]. The new abstraction data type *conversations* is introduced, through which users interact with the security facilities. To create a conversation, a caller passes its name, its *private key*, and the name of the other principal to the RPC runtime system. That conversation is then used as an argument of an RPC, and the runtime system ensures that the call is performed securely using a conversation key known only to the two principals.

The scheme is based on the use of *private keys* and built upon an *authentication service* (or *key distribution center*). Each principal has a private key known only to the principal and the authentication service. A caller and a callee that want to communicate negotiate with the *authentication service* to obtain a shared *conversation key*. This key is used to encrypt subsequent communication between the two principals. The federal Data Encryption Standard [52] (DES) is used for encryption, because very fast and inexpensive hardware implementations are broadly available.

For example, if principal A wants to communicate securely with principal B, A's program includes a call on the RPC runtime system in the form:

```
conv <- R.P.C.Create
          Conv[from:   nameOfA,
               to:     nameOfB,
               key:    privateKeyOfA]
```

Principal A can then make remote calls to procedure P.Q implemented by B, such as:

```
x <- P.Q[thisConv:conv,arg:y]
```

Inside the implementation of P.Q, principal B can find the identity of the caller by a call in the RPC runtime system in the form:

```
caller <- RPC.GetCaller[thisConv]
```

The concept of conversation is orthogonal to the other abstractions in a call. Multiple processes can participate in a conversation, and multiple simultaneous calls may be involved in a conversation.

8.7 Broadcasting and Multicasting

It is argued in the research community that there are applications to which RPC seems to be an inappropriate paradigm. Several extensions to this model have therefore been proposed. One extension is applications based on the necessity for multicasting or broadcasting.

Programming language support for multicast communication in distributed systems that can be built on RPC is presented in [19]. The callee interface has no modifications at all. The caller has additional binding disciplines, which provide for binding a call to a group of callees, and additional linguistic support. Callers have to deal with several replies, instead of just one, and may have to block only until the first n replies are received.

8.8 Remote Evaluation

One of the main motivations behind interprocess communication and distributed environments is that specialized processors can be interconnected. This kind of environment seems like a single, multitarget system able to execute, possibly efficiently, several specialized computations. In an RPC model, the specialized processors are generally mapped to callee processes that offer (export) a certain interface. Because of the wide range of applications to which these processors may be applicable, there may be no ideal set of remote procedures that a callee should provide [71]. In the worst situation, the RPC model requires as many customized interfaces from a set of callee processes as there are applications that may make use of it. Without an appropriate RPC interface, a distributed application can suffer degraded performance because of the communications overhead. It is claimed in [71] that, in the RPC model, a program has a unique partition into fragments for local and remote execution.

Remote Evaluation (*REV*) is the ability to evaluate a program expression at a remote computer [71]. A computer that supports REV makes a set of procedures available to other computers by *exporting* them. The set of procedures exported by a computer is its *interface*. When a computer (*client computer*) sends a program expression to another computer (*server computer*), this is a REV request. The *server* evaluates the program expression and returns the results (if any) to the client.

Before sending the program expression, the client prepares the code portion for transmission, in the *encoding process*. The end result of this encoding is a self-contained sequence of bits that represents the code portion of the request in a form that can be used by any server supporting the corresponding service. It may consist of compiled code, source code, or other program representation. This choice affects mainly the run-time performance and server security.

The use of precompiled REV requests can improve run-time performance, because the servers directly execute the REV requests instead of using an interpreter. It may not be very useful in a heterogeneous computing environment, and has the potential for compromising server security. Dynamic compilation of REV requests at servers is certainly possible, but the net effect on performance depends on the REV requests. The performance of an RPC mechanism is bounded by the overhead of internode communication. The REV may eliminate such overhead. Because trade-off is between generality and performance when RPCs are used, service designers typically choose performance over generality. When REV is avail-

able, however, programmers can construct distributed applications that need both. While in the RPC model, a callee process exports a fixed set of procedures, a REV environment allows the application programmer to compose server procedures to create new procedures that have equal standing with the exported ones.

The RPC REV relationship is compared with the relationship between built-in data type and new data type definitions. The REV is claimed to be a generalization of, and an alternative to, RPCs.

One of the drawbacks of REV is that it does not allow functions to be passed as arguments of REV program expressions. The *Network Command Language* [24] (*NCL*) defines a new programming model that is symmetric, that is, not restricted to caller and callee network architectures. Both callers and callees are allowed to use NCL to send expressions that specify an algorithm for an operation.

To communicate with a remote host, the caller has to establish a session with it, which is equivalent to logging into a *server* by passing appropriate credentials. If the caller fulfills the authorization requirements, it enters an interactive LISP read-eval-print loop with the host's evaluator (server). The server connects back with the caller in the same way. The caller and server exchange NCL expressions representing requests and responses over this session as if each were a user typing to an interactive evaluator. A caller or server can vary the programming dynamically to suit changing requirements within a heterogeneous distributed system. In this way, a server can perform all of the operations necessary for a collection of heterogeneous callers without previous definition, linkages, or interfaces modules for every combination of primitives. As a result, the ultimate point of compatibility for distributed systems shifts to the NCL syntax, libraries, and canonical data formats.

The primary advantage of this scheme is that many application programs can be built that translate tasks from the language that the client is used to, to expressions to be evaluated remotely. This advantage makes this software highly extensibly.

To transmit expressions, LISP-like representations are used that are very economical. On the other hand, it can be rather inefficient having these expressions interpreted on the server's machine. The integration with RPC-based systems is also not easy.

9 Performance Analysis

It is a common practice for experienced programmers to use small procedures instead of inline code, because it is more modular and does not affect performance too much [82]. If such procedures are run remotely however, instead of locally, the performance may be severely degraded.

RPCs incur an overhead between two and three orders of magnitude greater than local calls [86]. The cost of making an RPC in the absence of network errors can be classified as follows [86]:

```
call time = parameter packing
          + transmission queuing
          + network transmission
          + callee queuing and
            scheduling
          + parameter unpacking
          + execution
          + results packing
          + transmission queuing
          + network transmission
          + caller scheduling
          + results unpacking
```

It can be rewritten as:

```
call time = parameter transformations
          + network transmission
          + execution
          + operating-system delays
```

Programs that can issue RPCs can be modeled, and a cost analysis can be developed based on simple measurements, as shown in [50]. RPCs can be classified according to the size of input, the size of output, and the amount of computation performed by the procedure. Application programmers are provided with some guidelines, such as, how often the user can afford to call an expensive RPC considering the degradation of performance because of remote execute instead of local execution, the cases in which the remote host can

perform the computation faster and the speeding up factor, or, more generally, the expected execution time of a program with a certain mix of RPCs.

An extensive analysis of several RPC protocols and implementations developed before 1984 was presented in [53], and the first measurements of a real implementation given in [13]. The measurements were made for remote calls between two Dorados connected by an Ethernet with a raw data rate of 2.94 megabits per second, lightly loaded. Table 1 shows the time spent on average, and the fraction of that time that was spent in transmission and handling of local calls in the process. These measurements are made without using any encryption facility from the moment an RPC is requested to the moment at which the result is in the final destination.

In the Modula-2 extension [1] to support RPC, it was also observed that RPC timings are directly affected by the number and size of parameters, as shown in Table 2. The performance was much better also for out arguments, than for in-out, and for idempotent procedures than for non-idempotent ones.

In distributed (RPC or message passing based) operating systems, large percentage (94% to 99%) of the communication is only cross-domain, not cross-machine [9]. Effort has been made to optimize the cross-domain implementation of RPCs to bring considerable performance improvements. The *Lightweight RPC* [9] (*LRPC*) consisted of a design and implementation effort to optimize the cross-domain use for the TAOS operating system of the DEC SRC Firefly multiprocessor workstation. The improvements are the result of three simple aspects:

- *Simple control transfer:* the caller's thread executes the requested procedure in the callee's domain.
- *Simple data transfer:* a shared memory exists between the caller and the callee, and the arguments are copied only once for all situations.
- *Simple stubs:* are a consequence of the simple model of control and data transfer between threads.

Procedure	Average Time	Trans. Time	Local Call Time
no args/results	1097	131	9
10 args/results	1278	239	17
240 word array	2926	1219	98

Table 1: RPC Performance in the Cedar Project (in μs).

Number of bytes	Time for Local Calls	Time for RPCs
1	1.05	3.51
25	1.79	4.17
1016	2.99	8.22
3000	6.66	19.19

Table 2: RPC Performance in Extended Modula-2 (in *ms*).

The system adopts a concurrency-based design that avoids shared data structure bottlenecks. It also caches domains context on idle processors, avoiding context switch. A performance of 157 *ms* is achieved for the null cross-domain call, and of 227 *ms* for the 200-byte in-out argument.

Another approach used to improve performance was to carefully analyze the steps taken by an RPC, and to precisely identify the bottlenecks [63]. An optimization for the RPC mechanism for the Firefly multiprocessor computer is described in [63]. The architecture is based on multiprocessors in the same machine sharing only one I/O bus connected to the CPU 0. This feature could affect RPC latency on other processors if a high load on this particular CPU occurs. Great performance increases are achieved through the identification of the major bottlenecks in the mechanism and the optimization of the process in the particular points. During marshaling, the stubs use customized code that directly copy arguments to and from the call or result packet, instead of calling library routines or an interpreter. The RPC and transport mechanisms communicate through a buffer pool that resides in a shared memory accessible to all user address spaces, to the Nub, and which are also permanently mapped into the I/O space. In this way, all these components can read and

write packet buffers in the same address space with no need for copying or extra address mapping operations. Nevertheless, no protection has been added and this strategy can lead to leaks into the system. An efficient buffer management is done, making the server stub reuse the call packet for the results, and the receiver interrupt handler to immediately replace the buffer used by an arriving call or result packet, checking for additional packets to process before termination. A speedup by a factor of almost three was obtained rewriting a particular path in assembly language. The demultiplexing of RPC packets is done at the interrupt routine, instead of the usual approach of awakening an operating-system thread to demultiplex the incoming packet. The Firefly RPC offers the choice of different transport mechanisms at bind time. The integration of Firefly RPC into a heterogeneous environment, where RPC can be especially beneficial, is not discussed. Heterogeneity is also a potential source of performance optimization problems, because differences in machine architecture and information representation can make time-consuming conversion operations unavoidable. Portions of the software are implemented in assembly language, which may cause portability and maintenance problems. Possible additional hardware and software speedup mechanisms are discussed, offering estimated speedups of 4% to 50%.

An effort was made to take some of the operations in communications in distributed systems to the hardware level. Hardware support for message passing at the level of the operating system primitives was proposed in [60]. A message coprocessor is introduced that interacts with the host and network interface through shared memory, and provides concurrent message processing while the host executes other functions. In control of the network interface the coprocessor takes care of the communication process. This involves checking the validity of the call, addressing and manipulating control blocks, kernel buffering, short-term scheduling decisions, and sending networks packets, when necessary, and so on.

10 Conclusion

Distributed systems provide for the interconnection of numerous and diverse processing units. Software applications must relish such broad functionality, the inherited parallelism available, and the reliability and fault tolerance mechanisms that can be provided. Several language-support paradigms have been proposed to the modeling of applications in distributed environments. An inappropriate choice of language abstractions and data type constructors may only add to the ever-increasing software development crisis. It may prevent software maintenance, portability and reusability. Today's conception of a distributed system is far behind the desirable model of computation. It is mandatory that new software paradigms be as independent as possible of the intrinsic characteristics of a specific architecture and, consequently, must be more abstract.

In this work, we studied RPC, a new programming language paradigm. A survey of the major RPC-based systems and a discussion of the major techniques used to support the concept were presented. The importance of RPC is because of its conservative view toward already developed software systems. The goal of RPC development is to provide easy interconnection of new and old software systems, despite its language, operating system, or machine dependency. To achieve this goal, the foundation of RPC is based upon a strong language constructor, the procedure call mechanism, which exists in almost any modern language.

This work may inspire important future work. Syntax and semantic transparency are ever-challenging research topics and the suitability of the RPC paradigm for new distributed systems applications, such as multimedia, is also an important topic to be studied.

Acknowledgements

I am thankful to Shaula Yemini for suggesting the writing of this survey, and for giving significant comments on its overall organization. I am also thankful to Prof. Yechiam Yemini, my advisor, for carefully reviewing earlier drafts of this work. Finally, I would like to thank the

referees for their careful reading of this paper and for their valuable comments.

References

[1] Guy T. Almes. The impact of language and system on remote procedure call design. In *The 6th International Conference on Distributed Computing Systems*, pages 414–421, Cambridge, Massachusetts, May 1986. IEEE Computer Society Press.

[2] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–58, January 1985.

[3] A. L. Ananda, B. H. Tay, and E. K. Koh. ASTRA - an asynchronous remote procedure call facility. In *The 10th International Conference on Distributed Computing Systems*, pages 172–179, Arlington, Texas, May 1991. IEEE Computer Society Press.

[4] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *ACM Operating Systems Review*, 26(2):92–109, April 1992.

[5] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[6] Gregory R. Andrews and Ronald A. Olsson. The evolution of the sr language. *Distributed Computing*, 1(3):133–149, 1986.

[7] Joshua Auerbach. Concert/C specification - under development. Technical report, IBM T. J. Watson Research Center, April 1992.

[8] Henry E. Bal, Jennifer G. Steiner, and Andrew S. Tanembaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[9] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[10] Brian N. Bershad, Dennis T Ching, Edward D Lazowska, Jan Sanislo, and Michael Schwartz. A remote call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, 13(8):880–894, August 1987.

[11] Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[12] Andrew D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *ACM Communications*, 4(25):260–274, April 1982.

[13] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[14] John R. Callahan and James M. Purtilo. A packing system for heterogeneous execution environment. *IEEE Transactions on Software Engineering*, 17(6):626–635, June 1991.

[15] B. E. Carpenter and R. Cailliau. Experience with remote procedure calls in a real-time control system. *Software Practice and Experience*, 14(9):901–907, September 1984.

[16] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[17] David L. Cohrs, Barton P. Miller, and Lisa A. Call. Distributed upcalls: A mechanism for layering asynchronous abstractions. In *The 8th International Conference on Distributed Computing Systems*, pages 55–62, San Jose, California, June 1988.

[18] E. C. Cooper. Replicated procedure call. In *3rd ACM Symposium on Principles*

of Distributed Computing, pages 220–232, Vancouver, BC, 1984.

[19] Eric C. Cooper. Programming language support for multicast communication in distributed systems. In *The 10th International Conference on Distributed Computing Systems*, pages 450–457, Paris, France, June 1990. IEEE Computer Society Press.

[20] Department of Defense, Department of Defense, Ada Joint Program Office, Washington, DC. *Reference Manual for the Ada Programming Language*, July 1982 edition.

[21] L. P. Deutsch and E. A. Taft. Requirements for an exceptional programming environment. Technical Report CSL-80-10, Xerox Palo Alto Research Center, Palo Alto, California, 1980.

[22] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. The network computing architecture and system: An environment for developing distributed applications. In *USENIX Conference*, pages 385–398, Phoenix, Arizona, June 1987.

[23] Jeffrey L. Eppinger, Naveen Saxena, and Alfred Z. Spector. Transactional RPC. In *Proceedings of Silicon Valley Networking Conference*, April 1991.

[24] Joseph R. Falcone. A programmable interface language for heterogenous distributed systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[25] Nissim Francez and Shaula A. Yemini. Symmetric intertask communication. *ACM Transactions on Programming Languages and Systems*, 7(4):622–636, October 1985.

[26] Neil Gammage and Liam Casey. XMS: A rendezvous-based distributed system software architecture. *IEEE Software*, 2(3):9–19, May 1985.

[27] Narain H. Gehani and William D. Roome. Rendezvous facilities: Concurrent C and the Ada language. *IEEE Transactions on Software Engineering*, 14(11):1546–1553, November 1988.

[28] Phillip B. Gibbons. A stub generator for multi-language RPC in heterogenous environments. *IEEE Transactions on Software Engineering*, pages 77–87, 1987.

[29] David K. Gifford and Nathan Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[30] R. Gimson. Call buffering service. Technical Report 19, Programming Research Group, Oxford University, Oxford University, Oxford, England, 1985.

[31] K. G. Hamilton. *A Remote Procedure Call System.* PhD thesis, University of Cambridge, Computing Lab., Univ. Cambridge, Cambridge, England, 1984.

[32] Per Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications ACM*, 21(11):934–941, November 1978.

[33] Roger Hayes and Richard D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 13(12):1254–1264, December 1987.

[34] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[35] C. A. R. Hoare. Communicating sequential processes. *Communications of ACM*, 21(8):666–677, August 1978.

[36] Apollo Computer Inc. Apollo NCA and sun ONC: A comparison. In *1987 Summer USENIX Conference*, Phoenix, Arizona, June 1987.

[37] International Standard for Information Processing System. *Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*, 1987. ISO Final 8824.

[38] Michael B. Jones and Richard F. Rashid. Mach and matchmaker: Kernel language support for object-oriented distributed systems. In *OOPSLA'86 Proceedings*, pages 67–77, Portland, Oregon, November 1986. SIGPLAN Notices.

[39] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An interface specification language for distributed processing. In *12th ACM Symposium on Principles of Programming Languages*, pages 225–235, New Orleans, LA, January 1985.

[40] B. Lampson. Remote procedure calls. *Lecture Notes in Computer Science*, 105:365–370, 1981.

[41] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local area network. *IEEE Journal on Selected Areas in Communications*, pages 842–857, 1983.

[42] Kwei-Jay Lin and John D. Gannon. Atomic remote procedure call. *IEEE Transactions On Software Engineering*, 11(10):1126–1135, October 1985.

[43] Barbara Liskov and Li uba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, Atlanta, Georgia, June 1988.

[44] Barbara Liskov. Distributed programming in Argus. *Communications ACM*, 31(3):300–312, March 1988.

[45] Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler, and William Weihl. Communication in the Mercury system. In *21st Annu. Hawaii Int. Conf. Syst. Sc.*, pages 178–187, January 1988.

[46] Barbara Liskov, Maurice Herlihy, and Lucy Gilbert. Limitations of synchronous communication with static process structure in languages for distributed computing. In *13th ACM Symposium on Principles of Programming Languages*, pages 150–159, St. Petersburg, Florida, 1986.

[47] Barbara Liskov and R Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems*, 5:381–404, 1983.

[48] Barbara Liskov, Liuba Shrira, and John Wroclaswski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.

[49] Klaus-Peter Lohr, Joachim Muller, and Lutz Nentwig. Support for distributed applications programming in heterogeneous computer networks. In *The 8th International Conference on Distributed Computing Systems*, pages 63–71, San Jose, California, June 1988.

[50] Dan C. Marinescu. Modeling of programs with remote procedures. In R. Popescu-Zeletin, G. Le Lann, and K. H. Kim, editors, *The 7th International Conference on Distributed Computing Systems*, pages 98–104, Berlin, West Germany, September 1987.

[51] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual (version 5.0). Technical Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[52] National Bureau of Standards (NBS), NBS, Department of Commerce, Washington, D.C. *Data Encryption Standard*, 1977.

[53] B. J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie Mellon University, Pitsburgh, Pensilvania, 1981.

[54] Gerald W. Neufeld and Yueli Yang. The design and implementation of an ASN.1-C compiler. *IEEE Transactions on Software Engineering*, 16(10):1209–1220, October 1990.

[55] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan

Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, March 1988.

[56] David Notkin, Norman Hutchinson, Jan Sanislo, and Michael Schwartz. Heterogeneous computing environments: Report on the ACM SIGOPS workshop on accommodating heterogeneity. *Communications of the ACM*, 30(2):132–140, February 1987.

[57] Dave Otway and Ed Oskiewicz. REX: A remote execution protocol for object-oriented distributed applications. In R. Popescu-Zeletin, G. Le Lann, and K. H. Kim, editors, *The 7th International Conference on Distributed Computing Systems*, pages 113–118, Berlin, West Germany, September 1987.

[58] Fabio Panzieri and Santosh K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, January 1988.

[59] C. Pu, D. Florissi, P. G. Soares, P. S. Yu, and K. Wu. Performance comparison of sender-active and receiver-active mutual data serving. In *HPDC-1, Symposium on High Performance Distributed Computing*, Syracuse, New York, September 1992.

[60] Umakishore Ramachandran, Marvin Solomon, and Mary K. Vernon. Hardware support for interprocess communication. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):318–329, July 1990.

[61] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[62] Michael D. Schroeder, Andrew D. Birrel, and Roger M. Needham. Experience with grapevine: The growth of a distributed system. *ACM Transactions on Distributed Systems*, 2(1):3–23, February 1984.

[63] Michael D. Schroeder and Michael Burrows. Performance of firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[64] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.

[65] Michael Lee Scott. Messages vs. remote procedures is a false dichotomy. *SIGPLAN Notices*, 18(5):57–62, May 1983.

[66] Robert Seliger. Extending C++ to support remote procedure call, concurrency, exception handling, and garbage collection. In *Proceedings of USENIX C++ Conference*, 1990.

[67] S. K. Shrivastava and F. Panzieri. The design of a reliable remote procedure call mechanism. *IEEE Transactions on Computers*, 31:692–697, July 1982.

[68] Ajit Singh, Jonathan Schaeffer, and Mark Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[69] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, third edition, 1989.

[70] Robert J. Souza and Steven P. Miller. UNIX and remote procedure calls: A peaceful coexistence? In *6th International Conference on Distributed Computing System*, pages 268–277, Cambridge, Massachusetts, May 1986.

[71] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[72] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[73] Michael W. Strevell and Harvey G. Cragon. High-speed transformation of primitive data types in a heterogeneous

distributed computer system. In *The 8th International Conference on Distributed Computing Systems*, pages 41–46, San Jose, California, June 1988.

[74] Robert Strom and Nagui Halim. A new programming methodology for long-lived software systems. *IBM Journal Research Development*, 28(1):52–59, January 1984.

[75] Robert E. Strom, David F. Bacon, Ar thur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes — A Language for Distributed Computing.* Prentice Hall, 1991.

[76] Robert E. Strom and Shaula Yemini. NIL: An integrated language and system for distributed programming. In *Proceedings of SIGPLAN Symposium on Programming Language Issues in Software System*, pages 73–82. ACM, June 1983.

[77] Mark Sullivan and David Anderson. Marionette: A system for parallel distributed programming using a master/slave model. In *The 9th International Conference on Distributed Computing Systems*, pages 181–188, Southern California, 1989. IEEE Computer Society Press.

[78] Sun Microsystems, Network Information Center, SRI International. *XDR: External Data Representation Standard (RFC 1014)*, June 1987. Internet Network Working Group Requests for Comments, No. 10014.

[79] Sun Microsystems, Network Information Center, SRI International. *RPC: Remote Procedure Call, Protocol Specification, Version 2 (RFC 1057)*, June 1988. Internet Network Working Group Requests for Comments, No. 10057.

[80] L. Svobodova. Resilient distributed computing. *IEEE Transactions on Software Engineering*, 10:257–268, May 1984.

[81] D. C. Swinehart, P. T. Zellweger, and R. B. Hagmann. The structure of cedar. *SIGPLAN Notices*, 20(7):230–244, July 1985.

[82] Andrew S. Tanenbaum and Robbert van Renesse. A critique of the remote procedure call paradigm. In R. Speth, editor, *Proceedings of the EUTECO 88 Conference*, pages 775–783, Vienna, Austria, April 1988. Elsevier Science Publishers B. V. (North-Holland).

[83] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *ACM Operating Systems Review*, 24(3):68–79, July 1990.

[84] Marvin M. Theimer and Barry Hayes. Heterogeneous process migration by recompilation. In *11th International Conference*, pages 18–25, Arlington, Texas, May 1991. IEEE Computer Society Press.

[85] Robbert van Renesse and Jane Hall. Connecting RPC-based distributed systems using wide-area networks. In R. Popescu-Zeletin, G. Le Lann, and K. H. Kim, editors, *The 7th International Conference on Distributed Computing Systems*, pages 28–34, Berlin, West Germany, September 1987.

[86] Steve Wilbur and Ben Bacarisse. Building distributed systems with remote procedure call. *Software Engineering Journal*, pages 148–159, September 87.

[87] Xerox Corporation, Xerox OPD. *Courier: The Remote Procedure Call Protocol*, December 1981. Xerox System Integration Standard 038112.

[88] M. Yap, P. Jalote, and S. K. Tripathi. Fault tolerant remote procedure call. In *The 8th International Conference on Distributed Computing Systems*, pages 48–54, San Jose, California, June 1988. IEEE Computer Society Press.

[89] Shaula Yemini. On the suitability of Ada multitasking for expressing parallel algorithms. In *Proceedings of the AdaTEC Conference on Ada*, pages 91–97, Arlington, VA, October 1982. ACM.

[90] Shaula A. Yemini, German S. Goldszmidt, Alexander D. Stoyenko, and Yi-Hsiu Wei. Concert: A high level language approach to heterogeneous distributed systems. In *9th International Conference*

on Distributed Computing Systems, pages 162–171, Newport Beach, CA, June 1989.

[91] Wanlei Zhou. PM: A system for prototyping and monitoring remote procedure call programs. *ACM Software Engineering Notes*, 15(1):59–63, January 1990.

Appendix

A Operational Semantics of Procedure Call Binding

Programming languages, together with the operating systems, provide the programmer user with an abstract view of the underlying physical architecture. From the user point of view, a program manipulates *entities* such as variables, statements, abstract data types, procedures, and so on, that are identified through their names. To do the appropriate mapping between these entities and the physical machine, the language designers usually maintain a list of *attributes* (*properties*) associated with each entity. For instance, an entity of type procedure must have attributes that represent the name and type of formal parameters, its result's name and type, if any, and parameter passing conventions. It also maintains an *instruction pointer* (*ip*) and an *environment pointer* (*ep* or the *access environment*). The ip allows for the localization of the code of the procedure inside the current address space. The ep provides access to the context for resolving *free variables* (global variables) in a procedure body. In this way, when the programmer refers to a specific entity through its name, the system can access a list of attributes and perform the correct mapping. The attributes must be specified before an entity is processed during run time. The *binding* corresponds to the specification of the precise value of an attribute of a particular entity. This binding information is usually in data structure *descriptor* of the entity. Some attributes can be bound at compilation time, others at run time (so that they can change dynamically), and some are defined by the language designers (the user cannot change them). Programming languages differ (1) on the kind of entities that they support, (2) on the kind of attributes they have associated with one entity type, and (3) on the *binding time* for each attribute.

In conventional procedure call mechanisms, a *binding* usually consists of finding the address of the respective piece of code (ip) by using the name of a procedure. The context in which free variables are to be resolved is the context of the caller procedure at the moment of the call accessed through the ep. This binding can be done during compilation time, if the body of the function is defined in the same module that the call is; during link-editing time if the procedure's body is in a module different from the caller's module; or during run time if the procedure's name is stored in variables (pointers in C) whose values can only be determined dynamically. In the last case, the compiler produces a mapping between a procedure's name and its address that can be accessed indirectly during run time.

B Operational Semantics of RPC Binding

An RPC is as an extension of the conventional procedure call mechanism to which a little binding information must be added. In a conventional situation, it is assumed that the caller and callee belong to the same program unit, that is, thread of execution, even though the language supports several threads of execution in the same operating system address space or in distributed programs as shown in Figure 1. In other words, there are no interthread calls, only intrathread calls, where the binding limits its search space to the program unit in which the procedure is called. The same happens with the execution environment when the called procedure is being identified. The callee also inherits the caller's context.

In RPC-based distributed applications, the same thread assumption is relaxed and interthread calls are allowed, as shown in Figure 2. The binding information must then contain additional attributes that allow the identification of the particular thread in which the callee procedure is to be executed (callee

thread). The callee thread now has its own context, perhaps completely apart from the caller's context, and the threads exchange information only through the parameters passed. Some additional semantics is needed, though, to define the behavior of parameter passing by reference and pointer-based memory access.

C Brief Historical Survey

The idea of extending the concept of procedure calls to distributed environment was first introduced in *Distributed Processes* [32]. In Distributed Processes, a distributed application consisted of a fixed number of sequential processes that could be executed simultaneously. A process has variables that are not directly accessible to other processes, has an initial statement, and has some *common procedures*. A process can call local common procedures or the common procedures of other processes (*external-request* RPC). An external request call has the syntax:

```
call <process_name>.<proc_name>
        (<expressions>, <variables>)
```

where `<process_name>` identifies the process in which the common procedure `<proc_name>` is to be executed. Before the procedure is executed, the `<expressions>` values of the call are assigned to the input parameters. After execution, the output parameters are then assigned to the `<variables>` of the call. Distributed Processes also provided *guarded commands* and *guarded regions* to allow nondeterministic execution of statements. They allow a process to make arbitrary choices among several statements based on the current state of its variables.

One of the major pioneers of RPC-based Distributed Environment was the Cedar [21, 81] project that was developed around 1984. The RPC Facility was mostly based on the concepts described in [53]. The Interface Description Language used was Mesa [51] with no extensions, and the user could import and export only interfaces, not procedures. The Grapevine [12, 62] distributed database was used as a means to store the exported information. The user was responsible for exporting and importing the interfaces before making the corresponding RPC. There was a flexibility in the binding time, which could be at compilation or runtime. Only a fixed, restricted set of data types, not including pointers, could be passed as arguments or returned in RPCs. The environment assumed was completely homogeneous. The callee processes were started and terminated by the runtime support acting as permanent servers of their interface, and executing no other computation and leaving no control to the user application to manage the service of the calls. An independent transport protocol was entirely developed, and was concerned mainly with efficient establishing and canceling of connections, and with their lightweight management. For system malfunctions, the Cedar mechanism did not use time-outs to prevent indefinite waiting by a client, but made use of special *probe messages* to detect whether the called server was still running.

An extension of P+, a language for applications in control system, to support RPC was proposed in [15]. The control system in which P+ was implemented already had a *remote execution protocol*. Any programmer could request the remote execution of any part of the program, which was written in a specialized syntax. This part was transmitted in ASCII code to an interpreter (server) in the remote machine. The server executed the transmitted source code interpretatively, and returned the results to the calling program, using a specialized syntax. To extend P+, routine *REM* was introduced. The *REM* accepted a runtime call descriptor, which includes the remote procedure name, the remote computer name, and for each parameter, a tag that indicates the parameter type and mode (read-only, read-write, write-only), the size of an array, and the address of the parameter. The REM encoded the descriptor in a datagram format written in the specialized syntax (containing the necessary interpretative code in ASCII) and binary values of the parameters. Making use of the remote protocol already present in the system, REM sent the datagram to the server and waited for the results. The server executed the code received interpretatively, which was a single procedure call, that was unaffected by the call issued by REM.

ARGUS [47] is an integrated programming language and system for distributed program development. The fundamental concept in the language is *atomicity*. The logical units of distribution are *guardians* and an atomic activity is an *action*. An action can be completed either by *committing*, when all involved objects take on their new states, or by an *abort* procedure, where the effect is as if the action had never started. A distributed application consists of a set of guardians that control access to a group of resources available to its users only through functions named *handlers* (local and possibly remote functions). A guardian contains processes that execute background tasks and handlers. Each call to a handler triggers the spawning of a separate process inside that guardian, which can execute many calls concurrently. A guardian can be created dynamically, and the user need only specify the node in which it must be created. Guardian and handler names are first-class elements. They can be exchanged through handler calls, which avoids the explicit importing and exporting of functions. Handler calls are location independent, and continue to work even though the corresponding guardian changes its location. They have *at-most-once semantics* (defined in Section 5.1) and they are based on the termination model of exception, terminating either normally or in one of several user-defined exception conditions.

In the Eden system [2], a distributed application is a collection of *Eden objects* or *Ejects*. Each *Eject* has a *concrete Edentype*, the piece of code executed by the particular Eject, and an *abstract Edentype*, which describes its behavior. Several *concrete Edentypes* can implement the same *abstract Edentype*. Ejects communicate via *invocations* procedures from one Eject to another. To issue an *invocation*, an Eject must have a *capability* for the callee Eject. Each Eject has one capability to which invocations can be addressed. Invocation procedures have *CallersRights* as a parameter, allowing the callee Eject to check the caller's right. The *Eden Programming Language* (*EPL*) provides facilities to control concurrency within Ejects. If the code fragments appear in different Ejects, it is the the job of the EPL to ensure parameter packing and stub generation. On the callee side, the translator builds a version of *CallInvocationProcedure* that is tailored to the invocation procedure it defines. On the caller side, the usual stub is built. On the EPL level, the language provides *ReceiveOperation*, *ReceiveSpecific*, and *ReceiveAny* that are used directly by the programmer. The *ReceiveOperation* takes a set of operations, and returns only when one of them is called, whereas *ReceiveSpecific* is provided only when universal or singleton sets are wanted.

Modula-2 was also extended to support RPC calls [1], on top of the V System implementation for SUN workstations. The goal was to allow entire *external modules*, as defined in Modula-2, to be remote, that is, a callee would be defined by the *definition module* and implemented (specified) by the *implementation module*. A major effort was made to generate as few as possible dissimilarities between a remote external module and a *local* module. Only a few attributes and constraints were added to the definition module; a stub generator was implemented for a particular definition module that generated the callee and caller stubs as well as headers; and a small runtime module was developed that supported the paradigm during execution time. The binding was done automatically the first time that a remote function was called, and this binding lasted for the entire execution time. Null RPCs, with no arguments, executed in around 3.32 milliseconds.

SR [6] is a distributed programming language that was concerned mainly with expressiveness (to make it possible to solve relevant problems in a straightforward way), efficiency in compiling and executing, and simplicity in understanding and using the language. The problem domain was distributed programming. SR provides primitives, which, when combined in a variety of ways, support semaphores, rendezvous, asynchronous message-passing, and local and remote procedure calls. SR's main component is *resources*, which have a *specification part* and an *implementation part*. The *specification part* defines the *operations* provided by the resource that is implemented by possibly several processes running on the same processor. All resources and processes are created dynamically and explicitly by execution statements. To call operations, SR provides mainly the `call` and `send` statements. The `call` statement has the

semantics of a procedure call, either locally or remotely, while **send** has the semantics of *semi-asynchronous message passing*. A **send** invocation is returned after the call message is delivered to the remote machine. On the callee side, SR provides **proc** and **in** statements to determine the operations execution time. An operation declared as a **proc** operation triggers a separate process to execute each call received. The **in** statement provides for the execution of a selected operation using the value of its associated Boolean expression. A **receive** statement waits for a call of an specified operation, and then saves the arguments of the call, but the call is not executed. To allow the early termination of an operation, the **return** statement terminates the smallest enclosing **in** statment or **proc** statement. Similarly, the **reply** statment terminates the respective statements without terminating the process. It provides for **conversations**, in which caller and callee continuously exchange information.

LYNX [64] supports RPC calls on the top of *links*, which are built-in data types. A *link* is a virtual circuit abstraction that represents a resource. Callee processes can specify the processes that can be bound to the other end of the link. In this way, LYNX provides for some symmetry between caller and callee, because both can choose with whom to communicate. Processes can exchange link bindings, and once exchanged, the previous owner loses access to the link. Procedures are called through **connect** statements that use a specified link to send a call through it. This operation blocks until the results are returned in a way similar to procedure calls. The process bound to the other end of the link accepts the call, executes it, and replies the results back, unblocking the caller process. A process can contain a single thread of control that receives requests *explicitly*, or a process can receive them *implicitly*, where each appropriate request creates its own thread automatically. A link end can be bound to more than one operation, which leads to a type security check on a message-by-message basis.

Around 1987, Apollo presented the Network Computing Architecture [22] (NCA), an object-oriented environment framework for the development of distributed applications, together with the Network Computing System (NCS), a portable NCA implementation. Heterogeneity was then included in the RPC designers concerns, but mainly at the level of the machine data representation. To interconnect heterogeneous distributed systems, NCA designed an RPC facility support called NCA/RPC, a Network Interface Definition Language called NIDL, and a Network Data Representation called NDR. The NCA/RPC is a transport-independent RPC system built on multiple threads of execution in a single address space. A replicated object location broker is used for location purposes. The concept of object-oriented programming was extended to abstract the location (where it is going to be executed) from the user, making objects the unit of distribution, reconfiguration, and reliability. The unit to be exported or imported was an object, an object type, or an interface. Location transparency was enforced by the flexibility of importing only objects or functionality independently of their location. The replication aspect of the system as a whole brought failure transparency, because several objects (processes) could export the same functions. One important extension was the flexibility of having user-designed procedures to marshal and demarshal complex data structures, thereby widening the usability of the paradigm. It also provided the possibility of attaching binding procedures to data types defined in the interface, allowing the development of even more customized code. An extensive analysis of the major advantages of Apollo NCA over the SUN/ONC (Open Network Computing) system is presented in [36].

SUN RPC system [79] is a stub generator called *rpcgen*, an *eXternal Data Representation* [78] (*XDR*), and a runtime library. From an RPC specification file, *rpcgen* generates the caller and callee stubs, and a header file that is included by both files. Each callee's host must have a daemon that works as a *port mapper* that can be contacted to locate a specific program and version. The caller has to specify the host's name and the transport protocol to acquire an *RPC handler*. The *RPC handler* is used as the second argument of an RPC call. SUN supports either TCP or UDP (explained in Section 7.1). An RPC call returns a **NULL** value in case any error occurs. An effort is

made to provide at-most-once semantics. Each RPC handler has a unique transaction *identifier* (*id*). Each distinct request using the RPC handler has a different value that is obtained by slightly modifying the id. On the caller side, the protocol uses tests to match this id before returning an RPC value. This matching guarantees that the response is from the correct request.

If the protocol chosen is UDP, the UDP callee has an option of recording all of the caller's requests that it receives. It maintains a cache with the result values obtained that can be indexed by the transaction id, program number, version number, procedure number, and caller's UDP address. Before executing a request, the callee checks into this cache, and if the call is duplicated, the result value saved into the cache is simply returned, on the assumption that the previous response was lost or damaged.

For security, SUN provides three forms of authentication. The *Null authentication* is the default situation, where no authentication occurs. The *Unix authentication* transmits with every ROC request a timestamp, caller's host name, caller's user id, caller's group id, and a list of all other group ids to which the caller belongs to. The callee decides, based on this information, whether it wants to grant the caller's request. The *DES authentication* uses the SUN RPC protocol.

Mainly concerned with heterogeneity, and searching for accommodation of old and new distributed systems in a single environment, HCS [55, 10] was designed. Major component was the HCS Remote Procedure Call (HRPC), designed on a *plug replacement basis*, where the basic components could be independently built and plugged together. The goal of HRPC was to identify (factorize) the major components of an RPC system, and to define precisely their interfaces. In this way, components could be replaced easily, if the new components supported the interface and functionality of the former one and hide the implementation details. The interconnection of *closed systems* (systems in which changes were neither possible nor wanted) were accomplished by simply building components that emulated the behavior of the systems. This interconnection made possible the interconnection of different RPC mechanisms that were heterogeneous on several levels, but mainly on the system level. There was considerable and broad reusability of components already built. HRPC divided an RPC System into five major components: the compile-time support (programming language, IDL and stub generator), the binding protocol, the data representation, the transport protocol, and the control protocol (basically a runtime support). The *localized translation* was another basic principle adopted by HRPC: if there exists an expert to do the job, let it do it, instead of concentrating a narrower knowledge in a more-general purpose component. For instance, one way of implementing a name server to deal with heterogeneous databases is to build a centralized database capable of doing all the different searches and conversions between systems. HRPC's name service (NSM) is a simple bridge between the several homogeneous name services that are connected to the RPC mechanism. Lazy decision making or *procrastination*, was also adopted, because in dealing with heterogeneous systems, an a late-as-possible commitment represents the most flexible one.

Marionette [77] is a software package based on the *master/slave* model for distributed environments. A distributed application in this model is one *master process* and many *slave processes*, all of them sequential. The interaction is done either through shared data structures created and updated by the master process, or through operation invocation, which can be a *worker operation* or a *context operation*. The context operations modify the slaves' process state and are executed by all slaves. Worker operations are each executed in a single slave, are not specified by the master, and are invoked through asynchronous RPC. Later, the master explicitly accepts the operation's result. To deal with heterogeneity at the data representation level, all data exchanged between machines is translated into SUN External Data Representation [78] (XDR). The conversion is done through user-defined routines. The import and export of functions is avoided by having all programs specify *linkage arrays*. One linkage array contains descriptors for worker operation functions, another contains descriptors for context operation functions, and another contains descriptors for types of shared

data structures. An index into one of the arrays works as a function identifier.

C++ was also extended to support RPC, concurrency, exception handling and garbage collection to form a superset of C++ named Extended-C++ [66]. A translator was designed to compile code in Extended-C++ into C++ code, that along with a runtime library, supported the extensions. For the support of RPC, attributes were added to the syntax that allowed the declaration of classes as *remotable*: its member functions were called remotely. A class was instantiated only for executing the callee side of RPCs. In this situation, only the class declaration was required in the caller's side code, avoiding linking the code with the definitions of the functions that can be made remote. There is an explicit distinction between a pointer to a local value and a pointer to a remote value, and the user must be careful to dereference correctly, or else a compilation error occurs. The pointer is declared as being remotable and the syntax used for dereferecing it is different from the syntax to dereference a local pointer. Only a subset of the types can be used as the types of remotable function arguments or returned values. A type in this set is a *transmissible type*. The fundamental data types and a remotable pointer are transmissibles. A local pointer is not transmissible: a user-defined encoder and decoder are needed for each class to make it a transmissible class. For encoding and decoding complex structures, such as, circular linked list the degree of sharing of remotable objects can be controlled. An object in Extended-C++ can be encoded and decoded as a *shared* object or as a *value* object. During a remotable call, a shared object can be encoded (decoded) at most once, whereas a value object has no limit on the number of times it can be encoded (decoded). This avoids looping during encoding (decoding) for circular lists, for instance, through the correct specification of an object encoder (decoder) function.

DAPHNE [49] consists of language-independent tools and runtime support to allow programs to be divided into parts for distributed execution on nodes of a heterogeneous computer network. A callee process is not shared between concurrent callers belonging to different users, and is private and created on demand. A caller process can issue the parallel execution of several callees and resumes control only when it receives a reply from all calls. Each machine has a *spawner* process that supports the creation of callee processes on demand. It is reached via RPC using a well-known transport address.

In Concert [90, 7], a distributed process model and an interface description language are designed. The ultimate goal of Concert is to allow the interprocess communication of processes that support its process model and IDL. The interoperability is possible even in the presence of heterogeneity at the language level, operating-system level, or machine level.

The process model is integrated at the language level, which provides greater portability of the concept. Languages usually must be minimally extended to support the model. Extensions encompass process dynamics, rendezvous, RPC, and asynchronous RPC. The model defines additional operations and three data abstractions: *processes*, *ports*, and *bindings*. The *processes* consist of a thread of control and some encapsulated data local to a process. Processes can be dynamically created and terminated at the language level, and may not correspond to operating-system processes. The interprocess communication is achieved via directed typed channels, whose receiving ends are represented by *ports*. The *bindings* represent the capability to call a port or to send a typed message to. The operations allow the manipulation of these abstract types, such as, the dynamic creation of processes, ports, and bindings, the query as to whether the ports have messages queued, the ability to wait for messages to arrive on specific ports, and the ability to process these messages and to reply if needed.

The `ports` introduce a new storage class, which have queues associated with them to hold calls. Functions can be declared to be in the `port` storage class, and can be passed to other processes, and can then be used for interprocess communication. On the callee side, the function `accept()` receives a collection of `port` storage-class functions and executes a rendezvous on one of the `ports`. If no calls are pending on any of the `port` functions, the `accept` function waits for a call to

become pending on at least one of the **ports**. Otherwise, it selects one of the **port** functions nondeterministically. The **accept** function demarshals the arguments, invokes the respective function, and transmits the results to the caller after the function is finished.

Concert is an ambitious design in which not only RPC but also asynchronous communication and process dynamics are provided at the language level. To support asynchronous communication, Concert adds two new data types and five new operators. The **receiveport** operator specifies ports for one-way messages; the **cmsg** (call message) operator specifies a two-way message that was received, but not replied to yet; the **send** operator sends a message in a non-blocking manner, that is, the caller is unblocked right after the message is sent, probably before it is received and processed; the **poll** operator allows a process to check whether messages are present at any of a collection of ports, but without processing the messages; the **send** operator is similar to the **poll** operator, but blocks if no call is pending and until some other process makes a call; the **receive** operator receives (dequeues) a message of any kind from any kind of port; and the **reply** operator replies to a two-way message which was received and represented as a **cmsg** operator.

Hermes [75] was the first language of the Concert family to be implemented. Hermes is a process-oriented language that fully supports all of the Concert process model in type-safe manner.

About the Author

Patrícia Gomes Soares received the B.Sc. and M.Sc. degrees in Computer Science from Universidade Federal de Pernambuco (UFPE), Recife, Brazil, in 1987 and 1989, respectively. Since 1989 she is a Ph.D. candidate in the Computer Science Department, Columbia University, New York. Her research interests include languages, compilers, and software systems in general.

She can be reached at the address Computer Science Department, Columbia University, 450 Computer Science Building, New York, NY, 10027. Her Internet address is soares@cs.columbia.edu.