# Semantics for Parameter Passing in a Type-Complete Persistent RPC

M. Mira da Silva and M. P. Atkinson

Dept of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

A. P. Black

Dept of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
Portland, OR 97291, USA

## Abstract

*Current RPC mechanisms for persistent languages are either pass by reference — in which case they do not scale — or pass by copy — in which case they duplicate objects and destroy sharing relationships. In this paper we argue that to build very large distributed persistent applications a compromise between these two mechanisms is needed. The ultimate goal of our research is to build a scalable persistent RPC while still maintaining object sharing, type safety, type completeness and semantics that are readily understood by application programmers.*

## 1 Introduction

This research concerns the construction and maintenance of long-lived, large-scale, persistent applications that store and manipulate large quantities of complex inter-related data. An example is a health-care information system managing data for hospitals, patients and doctors, and their relationships with funding and regulatory agencies. These applications can be characterised by two attributes.

**Persistence.** They need to store both complex structured data and the algorithms to manipulate them, together, for years or decades. The applications grow by evolution; they cannot be shutdown and restarted.

**Distribution.** They are constructed by connecting a number of component applications, which may run on separate systems. This is because a single system cannot cope with all the data or processing needed, and because data location must respect autonomy, geographic distribution and ownership requirements.

Persistent programming languages [1, 2] integrate those features traditionally delegated to a database management system into the programming language itself. This significantly simplifies the construction of persistent applications.

To provide distribution, persistent programming languages must be augmented by an inter-process communication (IPC) mechanism that provides a means of *passing information* between stores,[1] and a means of *synchronising activities* in different stores.

These stores may be executing on different machines that are autonomously managed, subject to independent failure, geographically separated and linked by relatively slow and unreliable networks. The component applications must continue to operate when parts of the system fail. As larger or longer-lived applications are considered, the IPC model must take into account stores dynamically joining and leaving the system. This context imposes a further requirement on the IPC facility, *to report subsystem failures.*

An IPC mechanism and a persistent programming language is an appealing combination for programming the class of applications described above. We propose that the design of such a "persistent IPC" should be guided by three principles.

**Generality.** A large number of distributed persistent applications should be implementable with the same IPC. Instead of offering a number of specific features, the IPC should be small and flexible enough to support a variety of communication models.

**Simplicity.** The existing persistent programming model should be disturbed as little as possible, so that code and techniques that worked locally still work locally. New distributed techniques should maintain the local programming model; but where that is infeasible, the differences should be easily understood by programmers.

[1] We use the word "store" in the rest of the paper to mean an application and the data it stores and manages; others have used address space, program, process, node or machine with the same or similar sense.

**Realism.** The properties of real networks and computers supporting the application — such as bandwidth limitations, latency and failure — must be accommodated. Requirements evolving over time will inevitably make some parts of the application inconsistent with other parts, especially in a distributed environment with a (large) number of geographically dispersed programmers. As the number of stores grows, algorithms must scale so that all the required actions can be performed in reasonable time.

Many models of providing IPC have been proposed for constructing such applications, for example: message passing [12], remote procedure call (RPC) [4], and asynchronous RPC [20]. We choose to conduct our discussion in the context of RPC, but similar issues arise with the other mechanisms.

RPC mechanisms have been built with many different objectives. For example, there are RPC mechanisms designed for heterogeneity [7], performance [3, 28, 18], flexibility [9], or simplicity [5].

Adding persistence to RPC introduces new issues, many of which have been addressed elsewhere [30, 23, 24]. In this paper we concentrate on the semantics for parameter passing in a type-complete *persistent RPC*. We use this term to refer to an RPC for a persistent programming language.

The next section discusses the problem raised by passing parameters and returning results[2] in a type-complete persistent RPC. Sections 3 and 4 discuss existing parameter passing semantics, and why they do not solve the problem. We then present our model for parameter passing and our experience with it in section 5. A summary and future work can be found in section 6.

## 2 Parameters in a persistent RPC

RPC is a well-understood inter-process communication mechanism, but high-level persistent languages introduce new potential and expectations.

### 2.1 Persistent programming languages

Our research concerns a class of programming languages that exhibit the following properties.

**Orthogonal persistence.** Objects of any type may persist, i.e., the type of an object does not determine whether it is persistent or transient.

**Data type completeness.** There are no restrictions on constructing types; all manipulable values can be stored.

[2] The issues for returning results are identical with those for passing parameters. Throughout the rest of the paper we use "passing parameters" to denote both activities.

**Strong type checking.** All operations are type-checked to protect long-term valuable data from invalid code. This increases the reliability of the store.

**Higher-order.** Procedures or objects are first-class citizens and have the same rights as values of any other type, including the right to persist and be passed as arguments to, or returned as results from, other procedures.

This class of languages is more fully defined in references [1] and [2]. We choose to work with them based on their long-term potential and because virtually all other languages can then be supported by the same mechanisms.

In our research we use Napier88 [25, 16]. Persistence in Napier88 is defined by reachability from a persistent root and referential integrity is maintained automatically. Napier88 also enjoys a *rich type system* with parametric polymorphism, abstract data types, infinite unions, and *linguistic reflection*, whereby a callable compiler permits applications to change or extend their own behaviour at execution time.

### 2.2 Combining persistence and RPC

Now let us examine some of the consequences of combining the above properties of persistence with the principles for RPC design listed in section 1.

1. Generality and simplicity combine with orthogonal persistence to require that the RPC permit *values of any type as parameters or results in remote calls*; this includes procedures, objects, and values of abstract and infinite union types.

2. Many RPC systems ignore the problem of distributed type checking. However, strong type checking is even more important for safety in a distributed persistent system because sub-applications are created and changed independently, and procedures may execute in remote stores.

3. To retain simplicity, object sharing and referential integrity need to be maintained across the distributed application. For example, an object in one store should be able to refer to an object in another store just as if it were local.

4. Realism dictates that we *cannot* ignore failures; it is not acceptable to stop the entire application whenever one of its components breaks. Thus, whenever a program operates on a remote reference, the programmer must indicate what to do if the referent is unreachable. This conflicts with the previous observation.

5. We have observed that *most of the contents of the store can be reachable from some objects.* Deep copying[3] of parameters in these circumstances is infeasible. However, a partial copy creates remote references, which, as we have just seen, is undesirable from the point of view of reliability. It also results in a semantics that is hard to define, and hence conflicts with the goal of simplicity.

Thus, we see that it is impossible to maintain all the attributes of persistence in a realistic distributed environment. A compromise is necessary, but any relaxation of persistent features must be carefully considered. The new semantics should be close to the original (local) persistent semantics. The perturbations introduced should be easily understood and accepted by persistent programmers, and any techniques and tools designed for a local environment should still work locally. Adaptation of existing software and methods from local to distributed processing should be as simple as possible, without ignoring the requirement for realism.

Our research goal is to identify such a compromise. This paper presents our current understanding of the space in which this compromise will be found.

## 2.3 Basic models for parameter passing

Many parameter passing schemes have been proposed in the literature. They can be classified into two models.

**Passing by reference.** Instead of sending the argument itself to the remote procedure, some RPC mechanisms send only a (remote) reference to the argument, which remains at the call site. Call by reference is also what many persistent languages do for local procedure calls, since it is efficient at call time, especially if the argument value is large.

**Passing by copy.** Most RPC mechanisms pass parameters by copying the value of the arguments to the callee. Call by copy is generally more efficient if the value is relatively small because it avoids further callbacks to fetch components of the parameter or to request operations on it.

Call by copy is usually the mechanism of choice when the application must cope with partial failure and recovery, because communication occurs only at call and return time.

However, passing parameters by copy forces application programmers to keep the data being copied to a minimum by restricting the uses that they make of references. They are also required to manage the consistency and identity issues arising from the creation of multiple copies of the same object.

[3] A deep copy is a copy of all of the reachable data. Conversely, a partial copy copies only a part of the reachable data, including the references that it contains, so that the copy shares sub-structure with the original. A shallow copy is a partial copy in which only the top-level structure is replicated, and all deeper sub-structure is shared.

## 2.4 The need for a compromise

We have seen that an RPC mechanism that passes parameters by reference would be ideal because remote references are invisible and maintain the local semantics, techniques and tools. But, as we will see in section 3, call by reference saturates the network and amplifies the effects of partial failures, creating difficulties in scaling the distributed application beyond a few tens of stores.

The reader may ask whether it is appropriate to apply the doctrine of separation of concerns. For example, is it not possible to first apply well-known techniques for masking failures, and then to employ reference semantics in the resulting failure-free system?

We believe the answer to be that this is a worthwhile enterprise, but that *it can never be completely successful.* By increasing failure resilience, it is possible to reduce the frequency with which an operation on a remote reference fails. *But the probability can never be reduced to zero.* So long as it is non-zero, the programmer must write a failure handler — which is exactly the conceptual burden that we were seeking to avoid. Without such a handler, the only way in which we can preserve the illusion of a failure-free system is to force the calling store to fail too. Failures will now cascade from one store to the next without any containment, causing failures in all of the other threads executing in those stores and raising the overall frequency of failure once again.

It is also possible to ameliorate the disadvantages of call by copy by automatically maintaining coherence between the copy and the original. However, the coherence protocol will fail if replicas become unavailable, so this attempt to simplify the programmer's life in one area will complicate it in another: it will create a new kind of failure that must in turn be handled.

The problem is like a balloon: we can squeeze it anywhere we want, but the consequence is that it just bulges out somewhere else.

## 3 Call by reference

When a parameter is passed by reference, the RPC mechanism sends only a token for the argument to the callee. That token may already exist or it may be created for the purpose; in either case it has to uniquely identify the argument throughout the application.

### 3.1 Examples

DPS-algol [30] is an upwardly-compatible distributed version of the persistent programming language PS-algol [1]. DPS-algol offers a basic model of transparent distribution with an RPC mechanism that gives programmers some control over the placement of computations. Objects themselves cannot move; instead, the RPC mechanism always passes objects by reference. If the computation needs to access the value of an object, it has to migrate to the store where the object resides. However, if the object is of a built-in immutable type, it is instead copied to the computation.

Emerald [6, 14] is an object-based language and system for building distributed applications. Emerald uses call by reference in all invocations, local or remote; the semantics are the same in both cases. However, it is possible in Emerald to move the parameter objects to the remote site (in the same network message as the call) in order to increase performance.

More recently, the research work by Kato and others on HiRPC [15] and its follow-up [17] has demonstrated how to achieve efficiency in an RPC with call by reference semantics. As in Emerald, they employ a language construct to choose between call by reference or call by move. In addition, the RPC system also maintains a cache for frequently accessed remote data (coherency is automatically maintained). HiRPC employs variable depth copy into the cache.

### 3.2 Advantages

The main advantage of passing parameters by reference is its simplicity, as it offers the well-known local parameter semantics in a distributed environment.

Because mutable values are not copied, update and sharing semantics are automatically preserved. The implementation can invisibly copy immutable values. This guarantees that a distributed object has a well defined (in fact, unique) value, and its state is correctly shared by all its users at any one time.

### 3.3 Challenges

**Network traffic.** Passing by reference gives the illusion that only a small amount of data (i.e., the object identifier) is shipped between the two address spaces. However, sooner or later the computation may need to access the object, and either the object or the computation (and a sufficient part of its context) has to migrate so that both reside in the same address space for the computation to proceed.

The consequence is that call by reference will generate less traffic than call by copy only when the argument is never examined by the callee or when it is moved and sharing is sequential rather than concurrent. Furthermore, the resultant traffic is decoupled from the call; this makes failure handling more difficult.

**Inter-store dependencies.** A reference, once shipped to another store, may be assigned to local variables or object components, or dispatched to a third store. In practice, we have found that the objects that form a distributed application become strongly interconnected, increasing the dependency between parts of the application.

Such dependencies can be acceptable in a tightly-coupled distributed application running on a reliable local-area network, but they prevent scaling the application to a level where autonomy between stores is required. Autonomy is needed for availability, i.e., to cope with partial failures. Persistence implies that the application will run long enough for partial failures to be significant.

**Dealing with failures.** Ideally the RPC system would try to hide all failures from the application. This is unrealistic because some failures may persist indefinitely, and undesirable because users of the application may need to be informed. It is also impossible to foresee all kinds of failures in advance; unexpected system conditions are presented to the application as failures.

In some cases application programmers may be well prepared to deal with failures due to their knowledge of the application. However, they should not be asked to deal with transient or low-level failures (e.g., network congestion) as it unnecessarily complicates their work; it is unreasonable to ask application programmers to deal with failures below their domain.

A persistent RPC with call by reference has to provide mechanisms to deal with failures when accessing remote objects. This is in contrast with local references in a persistent environment.

## 4 Call by copy

When an RPC passes an argument by copy, the value of the argument is duplicated at the destination of the call. Passing by copy makes stores more autonomous because each has now a local copy of the object and does not depend on the network or the other store to access its value.

### 4.1 Examples

Most RPC systems pass parameters by copy, including the first RPC mechanism implemented by Birrel and Nelson [4], Hamilton's RPC [10], Argus [19], the RPC product from Sun [29] and OSF [27], XEROX's ILU [13], and our RPC described below.

Napier88/RPC [24] is a type-safe RPC mechanism for Napier88. Based on Sun/RPC [29], Napier88/RPC supports only a restricted number of type constructors as arguments to remote procedures, and passes parameters by copying them to the target program.

However, Napier88/RPC differs from Sun/RPC in a number of aspects. The implementation uses techniques made possible by Napier88, such as creating new stubs at run-time using reflection and storing them for later use. Napier88/RPC also supports a binding service based on *type sessions*. These are pairwise agreements between a client and a server stub that permit type-checking to occur only once per session.

Tycoon/RPC [23] is a type-complete RPC built for the persistent language Tycoon [21]. Tycoon/RPC uses an existing pair of procedures that flatten (and un-flatten) an object to (and from) a byte array. Flattening and un-flattening an object of arbitrary type is possible with access to the object's implementation, but great care is needed to comply with the language's type-checking. Because any type can be passed as an argument to the remote procedure, a variety of distributed techniques is possible with Tycoon/RPC, including migrating threads [22].

## 4.2 Advantages

If the value of the argument is copied to the target, the source store is free to proceed autonomously until the target finishes its computation and sends the result back. This is appropriate in large systems where the computation may be of a long duration or the source and the target are not guaranteed to be available all the time.

More importantly, in the absence of updates, with a local copy of the full transitive closure the computation in the target on the copy is identical to the same computation on the original object in the original source program.

## 4.3 Challenges

**Large reachability graphs.** Call by copy duplicates all the objects reachable from the parameters; the entire transitive closure of the reference graph must be copied, preserving any shared and cyclic data structures [11]. This closure can be large, and is limited only by the size of the address space of the source program.

In a persistent language the transitive closure of the parameters may include objects in the persistent store. Procedures that are first-class citizens may have very large transitive closures — potentially as large as the entire store — because the "address space" in a persistent program is the persistent store. The transitive closure may also include many objects that already exist in the *target* store, either because they are standard (e.g., a procedure to write a string on the screen) or because they have been copied as arguments to a previous call.

**Destroying the semantics of object sharing.** Loss of object sharing happens because, when a mutable object is passed as a parameter, the new copy of the object that is created remotely *is a different object* (the original object and the copy have the same value but different identities). If either the copy or the original is updated, inconsistent copies now exist for what is conceptually a single object. Worse, if the same object is passed as a parameter again, or returned as a result in a remote procedure, different copies of the same object may be created in the *same* store.

As an object may be copied repeatedly between stores, it may arrive at a store by several different routes. This further increases the complexity and the cost of verifying identity and achieving coherent update.

## 5 Call by substitution

In this section we propose a model for parameter passing that tries to incorporate the benefits — and avoid the problems — of both call by reference and call by copy. The fundamental requirements for our new model of *call by substitution* may be summarised as follows.

1. Exclude remote references that unnecessarily decrease store autonomy, create message traffic and aggravate the effect of partial failures.

2. Avoid duplicating objects that already exist in the target store.

3. Provide a well-defined, easily understood semantics with a simple application programming interface.

The essence of our model is to use different parameter passing semantics for different classes of values.

## 5.1 Partitioned parameter spaces

In each store computation proceeds in a value space $V$. We partition $V$ into three sets, although in principle there could be further partitions. The sets establish semantics for passing parameters and are defined as follows.

**Immutable set** ($I$)**.** Immutable values (such as integers, reals, booleans and strings) are simply copied; the semantics of computation is unperturbed by their replication.

**Substituted set** ($S$)**.** These objects are not copied; instead, they are *replaced* by equivalent objects in the target store. This set is explicitly defined by system managers or programmers using mechanisms outside the scope of this paper.

**Copied set** ($C$). These objects are deep copied on transmission, and are identified by set difference (values that are not in $I$ or in $S$ must be in $C$).

In Napier88 we have $I = int \cup real \cup bool \cup string \cup pixel \cup null$. The procedures that form the standard library [16] are good candidates for $S$ partition. The implementation must keep a database maintaining an enumeration of the $S$. Although $S$ may be the same between all pairs of stores, we treat it as a pairwise agreement so that $S$ may be changed without global collaboration.

Note that remote references do not appear in this scheme, and duplication is kept to a minimum.

## 5.2 Passing parameters by substitution

Before replacing a parameter object by its equivalent in the target store, application programmers in the source and target need to agree on which objects are substitutable and register them in a pair of tables. An initial set may be composed of all "standard" objects, i.e., those guaranteed to exist in any store.

However, what is "standard" may depend on many factors. For example, in Glasgow a programmer may understand "standard" to mean objects that belong to the Glasgow Libraries [31]. In addition, some programmers may also agree between themselves what is "standard" for an application.

Note that the substitution tables are needed only at call time. This permits the programmer to extend and/or reduce these databases at run-time. We can envisage some interesting uses of this flexibility by extending the range of "standard objects" at run-time.

After the RPC system has been set-up with substitutable objects in both the source and target stores, the algorithm for passing a parameter by substitution works as follows.

1. If the parameter value is in $I$ (determined from its type) its value is copied to the target store.

2. If the parameter objects belongs to the substitutable set, only a unique identifier for it is sent to the target store. When the identifier is received in the target store, it is used as a key to the local substitution table and is replaced by the equivalent local object.

3. Otherwise, the parameter is copied to the target store. This algorithm is applied recursively to embedded references.

There are a number of potential difficulties with call by substitution. If the substitutable object has mutable state, the state may not be consistent in the other store. Hence the absence of mutable state is a requirement for library objects.

It should also be noted that even though call by substitution can drastically reduce the amount of data being passed for many objects, it cannot guarantee that large closures will not be copied.

## 5.3 Implementation status

We have built three RPC prototypes in Napier88.

**Release 1.** Described in reference [24], the first release is based on Sun/RPC [29] and as such supports only a restricted set of constructor types. The parameters are deep copied to the target store, including shared and cyclic data structures [11]. Sharing is lost between the arguments or in successive remote calls.

**Release 2.** This second release supports any type as an argument to a remote procedure because it uses two procedures that write an object of any type to (and read from) the stable store [26]. When using this system, we encountered many arguments with large transitive closures that had to be copied to another store.

**Release 3.** This third and latest release performs call by substitution as described above. Programmers specify in the source and target stores which objects are substitutable. These specifications are maintained in a pair of substitution tables, but at present no attempt is made to verify the coherence of these tables.

## 5.4 Example application

The Library Explorer [8] is a persistent application that uses information retrieval techniques to find relevant procedures in the Glasgow Libraries [31]. The Explorer builds indexes over the contents of the library. These indexes are quite large in absolute terms (500 kB) but small compared to the library itself (approximately 11 MB).

An initial version of the Remote Explorer using release 1 of Napier88/RPC was built by dividing the Explorer into a client (mainly for user interface purposes) running at the programmer's store and a server (where the indexes are stored and the main processing takes place). Release 1 was adequate because only simple data types were exchanged between the client and the server.

The need to access the index remotely can be avoided if a copy of the index is cached at the client. Release 1 could not be used to copy the index because it is implemented as a complex data structure that contains two procedures (`isEqual` and `lessThan`). However, our first attempt to migrate the index using release 2 of Napier/RPC was unsuccessful; the procedures refer to their context, which happens to include the persistent root. Thus each attempt to copy the index would try to copy the entire store.

Using release 3 and call by substitution we succeeded in copying the index. The two procedures are now registered as substitutable at both the source and the target, and they do not actually migrate. A "definitive library server" holds the library and generates the index, which can then be copied to a number of clients. As a result, this latest version of the Remote Explorer has the advantages of centralisation (library maintenance at the server) and distribution (fast and reliable access to the index at the client).

## 6 Summary and future work

The contributions of this paper are as follows.

1. We have presented the major issues involved in the design of a type-complete persistent RPC that will help us to build the large-scale, long-lived applications of the future (sections 1 and 2).

2. We have shown that existing semantics for parameter passing — *call by copy* and *call by reference* — are not suitable for building the class of distributed persistent applications that we want to support (sections 3 and 4).

3. We have proposed a compromise for parameter passing semantics based on *call by substitution* (section 5).

This is a first step towards our ultimate goal of a generic, simple, realistic mechanism for inter-process communication in a persistent environment.

In order to further address these issues, we are currently instrumenting the use of our prototype in the Library Explorer (see section 5.4 above). Even though the explorer may not represent one of the large-scale applications that we want to support in the future, it is sufficiently long-lived to validate our claim for realism and to stress our design.

A number of distributed applications will also be built to test our claims of simplicity and generality. Both experienced and novice persistent programmers will be involved in this experiment.

### Acknowledgements

We would like to acknowledge Bernd Mathiske for sharing with us his experiences with Tycoon/RPC. Kazuhiko Kato's hospitality and insight into issues for this class of RPC mechanisms was highly appreciated. The research environment in Glasgow is partially supported by projects FIDE$_2$, IMIS and DRASTIC. Miguel Mira da Silva is supported by JNICT. Andrew Black is partially supported by ARPA contract number N00014-94-1-0845.

## References

[1] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, Nov. 1983.

[2] M. Atkinson and R. Morrison. Orthogonal persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *Operating Systems Review*, 23(5):102–113, Dec. 1989.

[4] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Dec. 1993.

[6] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan. 1987.

[7] A. Black, E. Lazowska, H. Levy, D. Notkin, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Commun. ACM*, Mar. 1988.

[8] J. Brown. A library explorer for the Napier88 GlasgowLibraries. Master's thesis, Department of Computing Science, University of Glasgow, Sept. 1993.

[9] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A flexible base for distributed programming. Technical Report SMLI TR-93-13, Sun Microsystems Laboratories, 1993.

[10] K. Hamilton. *A Remote Procedure Call System.* PhD thesis, University of Cambridge Computer Laboratory, 1984.

[11] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. Prog. Lang. Syst.*, 4(4):527–551, Oct. 1982.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–678, 1978.

[13] B. Janssen, D. Severson, and M. Spreitzer. *ILU 1.6.4 Reference Manual.* Xerox Corporation, May 1994.

[14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.*, 6(1), Feb. 1988.

[15] K. Kato, A. Ohori, T. Murakami, and T. Masuda. Distributed C language based on a higher-order RPC technique. *Computer Software*, 9(3), 1993. (in Japanese, also published in English by Iwanamim Shoten, Publishers and Academic Press, Inc.).

[16] G. Kirby, A. Brown, R. Connor, Q. Cutts, A. Dearle, V. Moore, R. Morrison, and D. Munro. The Napier88 standard library reference manual version 2.2. Technical Report FIDE/94/105, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994.

[17] K. Kono, K. Kato, and T. Masuda. Smart remote procedure calls: Transparent treatment of remote pointers. In *Proceedings of the 14th International Conference on Distributed Computing Systems (Poznan, Poland, June 21–24, 1994).* IEEE Computer Society Press, 1994.

[18] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Dec. 1993.

[19] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, Mar. 1988.

[20] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Lnaguage Design and Implementation*, pages 260–267, 1988.

[21] B. Mathiske, F. Matthes, and S. Mussig. The Tycoon system and library manual. Technical Report DBIS Tycoon Report 212-93, Computer Science Department, University of Hamburg, Dec. 1993.

[22] B. Mathiske, F. Matthes, and J. W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems (Naharia, Israel, June 1995)*, 1995.

[23] B. Mathiske, F. Matthes, and J. W. Schmidt. Scaling database languages to higher-order distributed programming. In *Proceedings of the Fifth International Workshop on Database Programming Languages (Gubbio, Umbria, Italy, 6th-8th September 1995)*, 1995.

[24] M. Mira da Silva. Automating type-safe RPC. In *Proceedings of The Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management (Taipei, Taiwan, 6th–7th March 1995)*, pages 100–107. IEEE Computer Society Press, 1995.

[25] R. Morrison, A. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby, and D. Munro. The Napier88 reference manual release 2.0. Technical Report FIDE/94/104, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994.

[26] D. S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, University of St Andrews, 1993.

[27] OSF — Open Software Foundation. *Remote Procedure Call in a Distributed Computing Environment: A White Paper*, 1991.

[28] M. D. Schroeder and M. Burrows. Performance of firefly RPC. *Operating Systems Review*, 23(5):83–90, Dec. 1989.

[29] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *RPC Programming Guide*, 1993.

[30] F. Wai. *Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation.* PhD thesis, University of Glasgow, Apr. 1988.

[31] C. Waite, R. Welland, T. Printezis, A. Pirmohamed, P. Philbrow, G. Montgomery, M. Mira da Silva, S. Macneill, D. Lavery, C. Hertzig, A. Froggatt, R. Cooper, and M. Atkinson. Glasgow libraries for orthogonally persistent systems — philosophy, organisation and contents. Technical Report FIDE/95/132, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1995.