A formal treatment of interference in remote procedure calls

G. Pappalardo, S.K. Shrivastava

Abstract

Remote procedure calls (rpcs) are a distributed programming facility enabling a client program to call a procedure that will be executed by a server computation running on a remote machine. It would be desirable that remote and local procedure calls could be treated uniformly, so as to make distribution transparent to the programmer. But transparency may be impaired if, because of communication or machine failures, computations serving the rpc of a client are allowed to interfere with computations serving later rpcs of that client. This paper gives a formal treatment of interference in rpcs. The formalism used is that of occurrence graphs. The main result is obtained is a sufficient condition for rpcs to be interference-free. The practical significance of this condition is highlighted by relating it to interference prevention techniques often adopted in the design of rpc mechanisms.
Bibliographical details

PAPPALARDO, Giuseppe

A formal treatment of interference in remote procedure calls.

Newcastle upon Tyne: University of Newcastle upon Tyne,

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series, no. 263.)

Added entries

SHRIVASTAVA, S.K.
UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Laboratory. Technical Report Series. 263

Abstract

Remote procedure calls (rpcs) are a distributed programming facility enabling a client program to call a procedure that will be executed by a server computation running on a remote machine. It would be desirable that remote and local procedure calls could be treated uniformly, so as to make distribution transparent to the programmer. But transparency may be impaired if, because of communication or machine failures, computations serving the rpc of a client are allowed to interfere with computations serving later rpcs of that client. This paper gives a formal treatment of interference in rpcs. The formalism used is that of occurrence graphs. The main result is obtained is a sufficient condition for rpcs to be interference-free. The practical significance of this condition is highlighted by relating it to interference prevention techniques often adopted in the design of rpc mechanisms.

About the authors

Mr. Pappalardo has been at the Computing Laboratory since April, 1986 as a Research Associate.

Professor Shrivastava joined the Computing Laboratory in August, 1975, where he is a Professor.

Suggested keywords

ATOMICITY
DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
NETWORK PROTOCOLS
RELIABILITY
REMOTE PROCEDURE CALLS
VERIFICATION

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.687
A formal treatment of interference in remote procedure calls

Giuseppe Pappalardo¹,² and Santosh K. Shrivastava¹

(1) Computing Laboratory, The University of Newcastle upon Tyne, UK
(2) Università di Reggio Calabria, Italy

ABSTRACT

Remote procedure calls (rpcs) are a distributed programming facility enabling a client program to call a procedure that will be executed by a server computation running on a remote machine. It would be desirable that remote and local procedure calls could be treated uniformly, so as to make distribution transparent to the programmer. But transparency may be impaired if, because of communication or machine failures, computations serving the rpc of a client are allowed to interfere with computations serving later rpcs of that client. This paper gives a formal treatment of interference in rpcs. The formalism used is that of occurrence graphs. The main result obtained is a sufficient condition for rpcs to be interference-free. The practical significance of this condition is highlighted by relating it to interference prevention techniques often adopted in the design of rpc mechanisms.

1. Introduction

Remote procedure calls (rpcs) are a widely used language facility for distributed programming. They provide a client program with the ability to call procedures whose code and parameters may reside on a remote machine. After invoking an rpc the client blocks, while an underlying mechanism, the rpc protocol, takes care of: (i) sending a call message to the remote machine, (ii) starting on it a server computation which executes the called procedure, and (iii) transmitting the computation results back to the client’s machine. Eventually the client returns from the call - either successfully, with some results received from the remote machine, or abnormally, with an exception signalling that some problem has occurred. Within this fundamental scheme many variations can be considered, concerning mainly the mapping of computations onto processes, and the adoption of reliability measures to cope with machine or communication failures.

The above description implies that rpcs afford some degree of abstraction from distribution and the related message passing. But in the sequel we shall require that abstraction from distribution, or transparency, should be as complete as possible. This is because, ideally, the programmer should be able to treat remote and local procedure calls (lpcs) uniformly. Transparency issues are discussed at length in [Nel]; among them we focus on the notions of call semantics, nested rpcs, orphans and interference.

Exactly-once semantics. When a lpc returns, the called procedure is known to have been executed exactly once if the return from it is successful, or partially - up to the point when an exception was raised - if the return is abnormal. Transparency requires that the same should apply to rpcs.

Nested rpcs. Like lpcs, rpcs should be allowed to nest: a computation serving an rpc should be able to invoke rpcs itself. It is therefore natural to associate with each computation a chain of ancestor rpcs.
Orphans. The execution of the procedure called by a rpc cannot continue after the rpc has returned (because the return of a rpc is always the consequence of the callee terminating), nor can it survive a crash that disrupts the process executing the caller program (because caller and callee are executed by the same process). For transparency's sake (and to avoid wasting resources) such orphan executions must not arise in the remote case either: no server computation should continue after any of its ancestor rpcs has returned successfully, returned abnormally, or has been disrupted by a crash of the calling client. Unfortunately, in the last two circumstances the requirement is not an easy one to meet, for client and server are on different machines connected by a communication medium, and each of these may fail independently. Consider e.g. the difficulties arising if the rpc protocol causes rpcs to return abnormally, should no reply be received within a given interval: when this happens, the remote server computation may well be still executing, and will become an orphan unless appropriate measures are taken. Likewise, measures are needed to deal with computations that become orphans because a crash disrupts the client they are serving.

Interference. The loss of transparency due to orphans makes the programmer's burden heavier, by forcing him to worry about interference between orphans and legitimate computations. As an example, suppose that a client obtains a lock on a remote terminal and issues two rpcs display(file1), display(file2). If display(file1) returns abnormally and leaves an orphan which is allowed to interfere with the computation for display(file2), then the lines of the two files may freely mix on the terminal screen! As the problem would not arise were the terminal local, the example illustrates the inconvenience of non-transparency. Even subtler problems may arise with nested rpcs; e.g. even a call that returns successfully may leave orphans that will interfere with computations for subsequent calls. This is illustrated in Fig. 1.1: the client p1 on site A makes an rpc c1 to site B and the server computation p2 on B makes an rpc c2 to site C; c2 returns abnormally, leaving an orphan p3 on C, but p2 on B completes and causes c1 to return successfully. A subsequent rpc c1' made by p1 to C may now produce at C a computation p3' that will interfere with p3.

![Diagram](image-url)

Fig. 1.1 - Interference in a nested call

At-least-once semantics. For particular applications one might settle for a non-transparent semantics allowing more than one computation to take place on behalf of a single rpc [LaB]. Even for such applications, however, it is unlikely that the ban on interference may be lifted. Rather, the designer who adopts at-least-once semantics will have to watch out also for interference between the multiple executions serving a call.
At-most-once semantics. Under this semantics, adopted e.g. in [LS], an rpc returns successfully iff it has given rise to one execution, abnormally iff it has had no effects. Nelson [Nel] argues - and we share his view - that at-most-once semantics is "pleasant", but is not required for transparency. At-most-once rpcs enjoy the property which is sometimes called "atomicity with respect to failure", but we shall use the term atomicity to mean "atomicity with respect to concurrency", in the sense introduced in Section 2.

This paper gives a mathematical treatment of interference in remote procedure calls. The formalism used, occurrence graphs [BR], is briefly introduced in Section 2. Sections 3 and 4 informally describe the assumed rpc execution model (a sort of common core of all rpc protocols), and orphans and interference. These concepts are formalized in Section 4 and 5. Although the formalization is of interest in itself, the main result of the paper, to be found in Section 6, is the formulation of a sufficient condition for rpcs to be interference-free in the presence of orphans; basically, the condition says that, if a call c precedes a call c', then on each node any work for c should precede any work for c'. Section 7 discusses how this condition can be ensured by extermination, expiration and crashcounts - three interference prevention techniques often adopted in the design of rpc protocols [Nel], [PS].

2. A brief overview of occurrence graphs

Occurrence graphs were introduced in [BR] as a means of studying atomicity and interference in concurrent systems. In this section we briefly review some definitions and concepts of that paper.

Definition 2.1 An occurrence graph (og) is a pair (Act,→), where Act is a set and → is an irreflexive binary relation over Act.

Ogs are intended as a formal means of describing computations (i.e. schedules) performed by parallel programs, whereby members of Act are interpreted as primitive program actions, hereafter termed simple actions, and the relation → as an ordering over them. Let ⇒ denote the transitive closure of →, then we give the

Definition 2.2 Given an og (Act,→), and two distinct simple actions a and b:
- if a⇒b, we say that a happens before or precedes b;
- if neither a⇒b nor b⇒a, we write a@[b] and say that a and b are concurrent.

Note that, if an og is acyclic, the relation ⇒ is irreflexive and therefore a (strict) partial order; in this case it coincides with the the time ordering of [LaL].

As it is ⇒ rather than → that conveys the desired meaning of a temporal ordering, we will not distinguish between ogs that have the same ⇒. Given a computation of a program, we require the relation ⇒ of the computation's og to contain the pair a⇒b whenever:

PO (program ordering): a and b occur in the program within some construct that constrains a to precede b; or, if PO does not apply:

DO (data ordering): a and b access the same data object†, and a is performed before b in the computation considered.

This definition means that, even though a may be performed before b in a computation, if a and b are unconstrained by PO and are not mutually exclusive, i.e. if they are potentially concurrent, then a@b will hold in the og describing the computation. The motivation for this approach is that it permits the formulation of the atomicity criterion expounded below.

† In fact, it would be sufficient to require a⇒b only when a and b access the same object and either of them is a write.
Consider a concurrent program \( P \) containing the fragments \( A_1, \ldots, A_n \), which will be momentarily assumed disjoint. Traditionally, \( A_1, \ldots, A_n \) have been said to be atomic actions in a given computation of \( P \) if this computation is serializable, i.e. has the same effect as one in which the executions of the \( A_i \)'s do not interleave. Best and Randell [BR] formalize these concepts through the use of ogs. They consider the \( A_i \)'s to be atomic if the computation og becomes an acyclic graph when the actions of each \( A_i \) are "collapsed" into a single node. Otherwise, if the collapsed og contains a cycle \( A_i \Rightarrow A_j \Rightarrow A_i \), \( A_i \) will be said to be interfered with by \( A_j \). These definitions can be understood as follows. Suppose that \( A_i \) accesses a subset \( s_i \) of the program state. In a computation whose collapsed og is acyclic, no \( A_i \) can be interleaved with any extraneous simple actions on \( s_i \), or by DO the og would not be acyclic. Thus the program state is transformed by each \( A_i \) as though \( A_i \) were executed in isolation. We may conclude that if the computation og is acyclic, the \( A_i \)'s are atomic actions.

[BR] generalize their atomicity criterion to nested actions as detailed below.

**Definition 2.3** An abstraction hierarchy over a set \( Act \) is a tree having subsets of \( Act \) as nodes and satisfying the following:
- \( Act \) is its root;
- if \( a \in Act \), then \{a\} (still termed a simple action, with slight licence) is a leaf;
- if \( A_i \) is an internal node and \( \{A_1, A_2, \ldots, A_n\} \) is the set of its children, then \( \bigcup A_i = A \) and \( \bigcap A_i = \emptyset \) hold; \( A \) is termed an activity or compound action and \( A_1, \ldots, A_n \) its subactivities.

Note that the left-to-right ordering of hierarchy tree nodes is irrelevant to the hierarchy defined.

**Definition 2.4** A structured occurrence graph (sog) is a triple \( (Act, \rightarrow, AbsH) \) such that \( (Act, \rightarrow) \) is an og and \( AbsH \) an abstraction hierarchy over \( Act \).

**Definition 2.5** An abstraction level of an abstraction hierarchy over \( Act \) is the set of (simple or compound) actions left immediately below by a "cut" intercepting the edges of the abstraction hierarchy tree. \( Act \) (viz. the hierarchy tree root) is called the most abstract level. The set of the hierarchy tree leaves is called the least abstract level.

It can be easily proved that an abstraction level \( L \) satisfies to: \( \bigcup \{A | A \in L\} = Act \), and \( \bigcap \{A | A \in L\} = \emptyset \). This is consistent with the idea that defining abstraction levels is a way of structuring actions by proceeding bottom-up, so that no simple actions can be "missed out" or "counted twice".

**Definition 2.6** Let \( L \) be an abstraction level and \( A \in L \). The expansion of \( L \) with respect to \( A \), denoted by \( L/A \), is the set \( (L\setminus A) \cup \{A\} \) is a subactivity of \( A \).

It is easily proved that \( L/A \) is an abstraction level itself.

**Definition 2.7** Let \( G = (Act, \rightarrow, AbsH) \) be a sog and \( L \) an abstraction level of \( AbsH \). The view of \( G \) at \( L \), denoted by \( G@L \), is the og \( (L, \rightarrow_L) \), where the relation \( \rightarrow_L \) is defined by:

\[
A \rightarrow_L B \text{ iff } A, B \in L, A \not\equiv B \text{ and } a \rightarrow b \text{ for some } a \in A \text{ and } b \in B
\]

(often the subscript \( L \) is omitted). Note that \( G@L \) is indeed an og, in that \( \rightarrow_L \) is, as required by Definition 2.1, irreflexive. Intuitively, \( G@L \) is obtained from the og \( (Act, \rightarrow) \) by collapsing the simple actions of each activity \( A \in L \) into the node \( A \) of \( G@L \), and simplifying any redundant edges resulting. Likewise, \( G@L/A \) is the view obtained from \( G@L \) by "opening up" \( A \).

**Definition 2.8** A sog is said to satisfy atomicity if none of its views contains a cycle.

[BR] also gives the formal definitions - which we omit - of interference between activities and of atomic occurrence of an activity. Intuitively, \( B \) interferes with \( A \) if \( B \) occurs strictly after part of \( A \) and strictly before another part of \( A \); conversely, \( A \) occurs atomically if it is not interfered with by any other activity. The atomicity of a sog and the atomicity of its activities are related by the

**Theorem 2.1** A sog satisfies atomicity if and only if all of its activities occur atomically.
Example 2.1 An og $G_1$ is shown in Fig. 2.1 (left), together with a possible abstraction hierarchy and three abstraction levels, $L_1 L_2$ and $L_3$ (centre). $G_1$ and the abstraction hierarchy define a sog $G$ which satisfies atomicity; accordingly, none of the views shown on the right is cyclic.

An extra link in $G_1$, however, can make $G$ a sog that does not satisfy atomicity (Fig. 2.2). Notice that the view $G@L_2$ contains now a cycle $A\rightarrow B\rightarrow A$, which implies that $B$ interferes with $A$.

Fig. 2.1 - A structured occurrence graph that satisfies atomicity

Fig. 2.2 - A structured occurrence graph that does not satisfy atomicity

3. A general execution model

This section informally introduces an rpc “execution model” as the basis for the formalization given later. Rather then an rpc protocol, the execution model is a set of properties (tagged in bold for further reference) that are general enough to be true for every conceivable rpc protocol. Also for the sake of generality, we assume an at-least-once rather than exactly-once semantics (cf. S2 below), for the treatment of the former easily specializes to the latter but not conversely.

From a physical point of view, an rpc system is composed out of sites. (S1) Sites perform computations made up of actions with a local effect and of rpcs. (S2) An rpc may result in more than one server computations being activated on the site to which it is made. (A1) On the site on which it is invoked, an rpc actually consists of two distinct actions: an invocation and a return; the latter may be normal, abnormal or crashed. About the order in which computation actions occur,
three assumptions are made. The first applies within a site: (PO1) computations are sequential, and the return from an rpc is the next (local) action a computation performs after the invocation of that rpc. The second two apply across sites: (PO2) the invocation of an rpc precedes any remote computation serving it, and (PO3) the normal return of an rpc is the result of the completion of one of the remote computations that serve the rpc.

As required by transparency (cf. Nested rpcs, Introduction), computations serving an rpc are allowed to invoke rpcs themselves; this suggests that recursion should somehow turn up in the formalization.

3.1 Peeking under the model

We have sought to make our model "general" so that the formalization need not be fraught with irrelevant details. Moreover, we have omitted to state our fault assumptions. As it turns out, the universe described by S1 and S2, the classification A1, and the causal relationships disguised under the PO precedence constraints are all that needs to be included in our model. Nevertheless, part of what has been omitted may be inferred by so to speak - "peeking" under the model, and in this section we shall do so as a help to intuition.

When an rpc is invoked, the rpc protocol causes a message requesting the execution of the called procedure to be sent to the called site. If the caller site uses a retry mechanism or the communication system duplicates data, the called site may receive several copies of a request message. This explains how an rpc may give rise to multiple server computations (cf. S2).

A computation that issues an rpc is suspended until the rpc returns (cf. PO1). A normal return may be thought of as the consequence of a reply from the remote site (cf. PO3), and an abnormal return as that of a timeout expiring before a reply is received; in either case, after the return, the suspended computation is resumed. It is also possible that after issuing an rpc but before returning from it the computation is disrupted and then, after a while, resumed (in an unspecified state). Such a resumption is considered to follow a "crashed" return, by analogy with the resumption following a return proper.

The introduction of crashed returns only aims at making the model more general by catering for possible rpc protocols that use some form of crash recovery. However, the treatment of crash recovery - which is generally performed in the context of at-most-once semantics - is beyond the scope of this paper. Crash recovery need only concern us insofar as, by effectively extending computation lives beyond crashes, it may give rise to interference with orphans. Indeed, without crash recovery there would be no computations which orphans of pre-crash calls could interfere with.

Finally, note that it is assumed that individual computations may fail: this is more general and yet not more difficult to treat than the usual assumption that a crash affects an entire site.

4. Extended activation trees and interference

4.1 Basic definitions

The first step of our formalization is to assume three sets: S, the sites, P, the computations, and C, the rpcs. We further presuppose the functions onsite: P→S, tosite: C→S, rcalls: P→C*, caller: C→P and a relation serves⊆P×C. Below we introduce their intended meanings and the constraints such meanings entail.

The local part of computation p is performed at the site onsite(p). The rpc c is made to the remote site tosite(c).

If computation p sequentially invokes the rpcs c₁,...,cₙ (n≥0), then we let rcalls(p) = <c₁,...,cₙ> and caller(cᵢ) = p (0 ≤ i ≤ n). The function caller is well-defined, for an rpc can be invoked by one computation only. If rcalls(p) = <> (<> is the empty string), p is said to be a bottom (i.e. completely local) computation.
\( p \text{ serves } c \) means that computation \( p \) has been activated as a result of rpc \( c \); hence the requirement that if \( p \text{ serves } c \), then \( \text{onsite}(p) = \text{ onsite}(c) \). For a given \( c \), under the at-least-once semantics assumed, there may be zero, one or more \( ps \) such that \( p \text{ serves } c \); in the former case \( c \) is said to be \textit{barren}. On the other hand, for a given \( p \) there can be either one or no \( c \) such that \( p \text{ serves } c \); in the latter case \( p \) is referred to as a \textit{top} computation. The properties of \textit{serve} are illustrated in fig. 4.1.

![Fig. 4.1 - Relation serves and function caller](image)

### 4.2Extended activation trees

The above functions provide some information concerning a computation (although not its full characterization). As a means of portraying this information, we now introduce the notion of extended activation tree (eact) of a computation. The eact \( T[p] \) of a computation \( p \) has root \( p \) and, from left to right, a child \( T[c] \) for each rpc \( c \) in \textit{rcalls}(p); \( T[c] \) has root \( c \) and a child \( T[q] \) for each computation \( q \) such that \( q \text{ serves } c \), where \( T[q] \) is the eact for \( q \). It is required - this is in fact an additional constraint on \textit{serve} and \textit{rcalls} - that all nodes of an eact should be distinct. Note that eact leaves are either bottom computations or barren rpcs.

Thus, the definition of eact is, as expected, recursive and entails that only top eacts, i.e. eacts of top computations, are not subtrees of any other eact. Note that eacts are partially ordered trees: a left-to-right order has only been defined among the children of a computation node \( p \), so as to reflect the order in which \( p \) makes its rpcs, but not among the children of an rpc \( c \); this is consistent with the choice of introducing \textit{rcalls} as a string-valued function and \textit{serve} as a relation.

**Example 4.1** The eact in fig. 4.2 describes top computation \( p_1 \), which runs on site \( A \) and sequentially performs the rpcs \( c_1, c_2 \) and \( c_3 \) to site \( B \); of these, \( c_3 \) is barren (e.g. because of a message loss), whereas \( c_2 \) and \( c_1 \) are served on \( B \) by \( p_4 \) and by \( p_2 \) and \( p_3 \) respectively; \( p_4 \) performs an rpc \( c_4 \) back to site \( A \), where \( p_5 \) is activated as a result.

### 4.3Local vs. remote procedure calls

Eacts obviously aim at extending the well-known concept of activation tree (act) for local, conventional procedure calls (lpc) [ASU]; thus, a comparison between eacts and acts may be useful to highlight the differences between rpcs and lpcs. The discussion below is rather informal: the reader is referred to Section 2 for the definitions of atomicity and interference freeness, and to Section 5 for the formal treatment.

#### 4.3.1 Lpcs

In the absence of crashes, a lpc \( c \) gives rise to exactly one (entirely local) computation \( p_c \) and returns only on termination of \( p_c \), which does not therefore survive the return of its originating call. As pointed out in [Neil], the same can be assumed even in the presence of crashes, for the crash sweeps away both \( p_c \) and the caller of \( c \). Two consequences follow. First, computations need not be represented explicitly in an act, except for the main program at the root (corresponding to the top computations of an eact). Second, lpcs give rise to a single thread of control. In more detail, at any time instant there is an \textit{activated call path}, starting at the root of the act, such that the only
computation being executed serves the last call in the path. Meanwhile (NI1) each computation for a call higher up in the path is waiting for the next call in the path to return, and (NI2) each call lying to the left of the path has returned and the computation for it has terminated. (The activated path is in practice recorded in the call stack).

4.3.2 Rpcs

Let us now turn to rpcs and eacts. According to assumption S2, an rpc may have multiple server computations, which so far nothing in our model prevents from being concurrent. It is also possible that a computation p returns from an rpc c to site A and proceeds, while a computation p' set up on A to serve c still survives thus becoming an orphan. In practice this may happen under two circumstances: (i) when c's return is abnormal or crashed, or (ii) - provided that c has multiple server computations - when c's normal return is due to the termination of a server p"other than p'. Therefore an rpc may give rise to several threads of control: at any time a top eact, unlike an act, may have several paths with rpcs awaiting return and several (sub)computations executing.

However, for the sake of transparency, we stipulate that a top computation is acceptable only in the following two cases.

(i) If, like its local counterpart, it gives rise to a single thread of control, viz., more specifically, if it satisfies NI1 and NI2, and moreover (NI3) the additional threads due to the at-least-one assumption are not concurrent.

(ii) If it produces the same results as some top computation satisfying (i), i.e. if any subcomputations carried out concurrently, within different threads of control, have the same effect as though they were not concurrent.

Intuition suggests that concurrent subcomputations are harmless so long as they do not reach the same site, where they might access the same data objects in such a way that interference arises.

(iii) may be rephrased in the light of the concepts introduced in Section 2: a top computation is acceptable if its structured occurrence graph satisfies atomicity or, equivalently (Theorem 2.1), if its subcomputations do not interfere with each other. Hence a methodology for a formal treatment: define the sog of the top computation, and devise correctness criteria that ensure that it satisfies atomicity. We shall accomplish these two tasks in Section 5 and 6 respectively.
4.4 A classification of interference

Given a top computation with its exact, suppose that two (sub)computations $p_1$ and $p_2$ interfere. We may distinguish three cases, depending on which of the NI requirements is violated.

I1 Closed inter-call interference: $p_1$ and $p_2$ lie on the same exact path. Assume that $p_1$ is an ancestor of $p_2$ (otherwise the dual reasoning applies): for $p_1$ and $p_2$ to be concurrent, $p_1$ must have returned from the call that gave rise to $p_2$, so $p_2$ is obviously an orphan.

I2 Open inter-call interference: $p_1$ and $p_2$ have a computation node $p$ as least common ancestor in the exact. Owing to the meaning of the left-to-right position of the children of a computation node, the orphan among $p_1$ and $p_2$ is the one that has the leftmost path to $p$, for it originates from a less recent rpc of $p$.

I3 Intra-call interference: $p_1$ and $p_2$ have a call node as least common ancestor in the exact.

These definitions and the concepts previously introduced are illustrated in the

Example 4.2 The computation of Example 4.1 might follow this scenario (before $c_3$ is invoked): $p_1$ makes rpc $c_1$ to site $A$, where $p_2$ and $p_3$ are started; because of a timeout expiry $c_1$ returns abnormally and $p_1$ proceeds by making rpc $c_2$ to site $B$, thus activating $p_4$; the latter makes $c_4$ to $A$, where $p_5$ is activated, but shortly after $c_4$ returns abnormally and $p_4$ resumes; meanwhile $c_2$ has also returned abnormally and $p_1$ has resumed. At this point $p_1$, $p_2$, $p_3$, $p_4$ and $p_5$ might well be executing simultaneously, thus producing several interference patterns, as summarized in Table 4.1. The scenario is described in Fig. 4.3 by the computation exact (from Fig. 4.2) and a temporal diagram (dashed lines indicate waiting for return, $\tau$ an abnormal return).

![Fig. 4.3 - Interference scenario](image)

5. A formal treatment of rpc atomicity: histories

5.1 Preliminaries

As noted in Section 4, in order to study rpc atomicity we need to characterize a top computation $p_t$ by a structured occurrence graph, which we shall term the history of $p_t$ and denote by $H[p_t]$. Being a structured occurrence graph, $H[p_t]$ must consist of a set of simple actions $Act[p_t]$ endowed with an abstraction hierarchy and an irreflexive precedence relation $\rightarrow$. These are defined in the next
<table>
<thead>
<tr>
<th>Interference type</th>
<th>Interfering computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra-call</td>
<td>(p₂,p₃)</td>
</tr>
<tr>
<td>open inter-call</td>
<td>(p₂,p₄) p₂ is an orphan</td>
</tr>
<tr>
<td>closed inter-call</td>
<td>(p₃,p₄) p₃ is an orphan</td>
</tr>
<tr>
<td></td>
<td>(p₅,p₅) p₅ is an orphan</td>
</tr>
</tbody>
</table>

Table 4.1 - Interference patterns

three subsections and then illustrated by Example 5.1. To convince himself that our formalization is indeed adequate, the reader need only compare it to our informal execution model (the correspondence between informal notions introduced in Sections 3.1 and formal definitions is emphasized by labelling both with the tags A, S and PO).

5.1.1 The set of simple actions

Given a top computation \( p_t \), for each computation \( p \) in the eact \( T[p] \) we assume a finite non-empty set of simple actions \( s\_actions[p] \). \( Acf[p] \), the set of simple actions of \( H[p_t] \), is defined as the union of the sets \( s\_actions[p] \).

Since \( s\_actions[p] \) is finite, it is countable; thus, when convenient (cf. Figs. 5.1, 5.2 and 5.3), its elements can be represented as \( w[p,n] \) for \( 1 \leq n \leq |s\_actions[p]| \). Each \( a \in s\_actions[p] \) is intended to represent a simple action of computation \( p \), and is therefore associated with an attribute \( a.t \), which can take one of the following values:

- \( LOC \) if \( a \) is an action whose effect is local to \( onsite(p) \);
- \( CALL \) if \( a \) is the invocation of an rpc;
- \( NORM \) if \( a \) is the normal return of an rpc;
- \( ABN \) if \( a \) is the abnormal return of anrpc;
- \( CRSH \) if \( a \) is the crashed return of an rpc.

We further assume the requirement:

\[ A1 \quad \text{For each rpc } c \text{ in } \text{rcalls}(p), \text{ there are in } s\_actions[p] \text{ two simple actions } i[c] \text{ and } r[c], \text{ such that } i[c].t=\text{CALL} \text{ and either } r[c].t=\text{NORM} \text{ or } r[c].t=\text{ABN} \text{ or } r[c].t=\text{CRSH}. \text{ All of the other elements of } s\_actions[p] \text{ have attribute } LOC. \]

5.1.2 Structuring the action set

We now proceed to introducing the second component of the history \( H[p_t] \), viz. an abstraction hierarchy on \( Acf[p] \). This is achieved by recursively defining the indexed activities \( w[p] \) and \( rpc[c] \):

\[ S1 \quad w[p] = \{ a \in s\_actions[p] | a.t=LOC \} \cup \bigcup_{c \in \text{rcalls}(p)} \text{rpc}[c] \]

\[ S2 \quad \text{rpc}[c] = \{ i[c], r[c] \} \cup \bigcup_{p \text{ serves } c} w[p] \]

According to these definitions, \( w[p] \) comprises all the simple actions - both local and remote - performed by computation \( p \); \( rpc[c] \) comprises the invocation and return of remote call \( c \) together with the simple actions performed by the computations serving \( c \).

Thus, the hierarchy tree of a history \( H[p_t] \) can be obtained from the eact \( T[p_t] \) by the rules:
H1 each computation node \( p \) is renamed \( w[p] \); each call node \( c \) is renamed \( rpc[c] \);
H2 for each \( a \in \_\_\_\_\_\_\_\_\_\_\text{actions}[p] \), a leaf \( a \) is attached to \( w[p] \) if \( a=t=LOC \), to \( rpc[c] \) if \( a=i[c] \) or \( a=r[c] \); we convene that \( i[c] \) shall be the leftmost child of \( rpc[c] \) and \( r[c] \) the rightmost.

Example 5.1 The temporal diagram in Fig. 5.1 on the left depicts the behaviour of top computation \( p_0 \). After some local activity, \( p_0 \) makes the remote call \( c_1 \), which gives rise on the called site to computation \( p_1 \). Immediately after returning successfully from \( c_1 \), \( p_0 \) makes the remote call \( c_2 \), which gives rise on the called site to computation \( p_2 \). Finally, \( p_0 \) returns successfully from \( c_2 \) and, after some local activity, terminates. The event \( T[p_0] \) is shown in Fig. 5.1, centre. The hierarchy tree of the history \( H[p_0] \), which is shown in Fig. 5.1 on the right, is obtained from \( T[p_0] \) by applying the above rules H1 and H2, and assuming that each vertical line in the temporal diagram corresponds to a single simple action.

![Diagram](image)

Fig. 5.1 - A computation scenario, the event and the hierarchy tree

5.1.3 The precedence relation
As noted in Section 2, the relation \( \Rightarrow \) of the sog \( H[p] \) should be determined by \( \text{PO} \), the set of precedence constraints entailed by the rpc protocol, and \( \text{DO} \), the particular schedule that \( p \) enforces on simple actions accessing the same data objects.

As for \( \text{PO} \), a set of precedence constraints that any rpc protocol should imply has already been identified as part of our informal execution model. The formalization is:

**PO1**
(i) The restriction of \( \Rightarrow \) to the set \( \_\_\_\_\_\_\_\_\_\_\text{actions}[p] \) is a total order.
(ii) If \( p = \text{caller}(c) \), then \( i[c] \Rightarrow r[c] \) and there is no \( a \in \_\_\_\_\_\_\_\_\_\_\text{actions}[p] \) such that \( i[c] \Rightarrow a \Rightarrow r[c] \).

**PO2** If \( p \) serves \( c \), then \( i[c] \Rightarrow a \) for all \( a \in w[p] \).

**PO3** If \( r[c] . t = NORM \), then there is a \( p \) such that \( p \) serves \( c \) and \( a \Rightarrow r[c] \) for all \( a \in \_\_\_\_\_\_\_\_\_\_\text{actions}[p] \).

For each set \( \_\_\_\_\_\_\_\_\_\_\_\text{actions}[p] \) it is assumed that the total order postulated by **PO1** agrees with the numbering \( w[p,n] \).

It is worth while to note that **PO1** implies that \( \Rightarrow \) is an asymmetric relation. The proof is by contradiction: if for some \( a \) and \( b \) both \( a \Rightarrow b \) and \( b \Rightarrow a \) held, \( a \Rightarrow a \) would follow, owing to the transitivity of \( \Rightarrow \) and contradicting **PO1** (i). Thus, **PO1** forbids cycles to occur in histories (but not in their views, see next section).
Fig. 5.2 depicts the precedence constraints entailed by **PO1-PO3**, and the position of the constrained simple actions in the hierarchy tree. Note how triangles are used to represent hierarchy (sub)trees - e.g. the triangle with vertex \( rpc[c] \) is the hierarchy subtree with root \( rpc[c] \). Dashed lines represent paths in the hierarchy tree.

**PO1:** \( i[c] \Rightarrow r[c] \)  
**PO2:** \( i[c] \Rightarrow a \)  
**PO3:** \( \text{if } r[c].t = \text{NORM} \text{ then } \exists p \text{ serves } c \land \text{ if } b \in s\_actions[p] \text{ then } b \Rightarrow r[c] \)

Fig. 5.2 - Precedences due to PO1, PO2 and PO3

According to Section 2, to determine \( \Rightarrow \) fully, after the PO constraints have been applied, a sog \( H[p] \) should be completed with the DO precedences, viz. precedences between simple actions that reference the same objects and are not constrained by PO. Note that this requires a knowledge of the semantics of simple actions in addition to the knowledge of their schedule.

### 5.2 Interference

Let \( H[p] \) be a history and \( L_p \) (resp. \( L_c \)) an abstraction level containing the activity \( w[p] \) (resp. \( rpc(c) \)). The PO conditions imply that in the view \( H[p]_L@L_p/u[p] \) (resp. \( H[p]_L@L_c/rpc(c) \)), where \( w[p] \) (\( rpc(c) \)) has been "opened up", precedences such as those shown in Fig. 5.3 should hold. This matches well the intuition behind the model of Section 3.1. Histories containing only precedences of this kind might be shown to satisfy atomicity. However, the question must be raised whether the graph edges introduced by the neglected DO precedences may jeopardize the atomicity of histories by introducing cycles in views like those of Fig. 5.3. We expect this to be possible, given that so far the formalization, faithful to the informal execution model, has omitted taking into account DO. The following example shows a history that satisfies PO1-PO3 and yet, because of the DO precedences, does not satisfy atomicity.

![Fig. 5.3 - Precedences within views of a history](image-url)
The abstraction hierarchy tree $T[p_1]$ with the abstraction levels $L1$ and $L3$ ($L0$ is the least abstract level, comprising all of the simple actions).

**Fig. 5.4 - A call history**
Example 5.2  The history \( H[p_1] \) in Fig. 5.4 describes in greater detail the scenario of Example 4.2. (Note the correspondence between the abstraction hierarchy and the first two subtrees of the eact in Fig. 4.2). It has been assumed that in addition to the edges due to PO1-PO3, the graph \( H[p_1] @ L0 \) contains edges due to DO, e.g. \( w[p_3,1] \rightarrow w[p_2,1] \) and \( w[p_2,1] \rightarrow w[p_3,2] \). All the more abstract views shown below \( H[p_1] @ L0 \) contain cycles, thus - as expected - \( H[p_1] \) does not satisfy atomicity; in particular, whenever two computations \( p_i \) and \( p_j \) listed as interfering in Table 4.1 are present at level \( L \), they are contained in a cycle in the view \( H[p_1] @ L \).

Histories would enable us to formalize the concept of orphan, and the three kinds of interference identified in Section 4.4. However, as this would require some of the definitions omitted in our short overview on ogs, we have chosen not to do so. Suffice it to say that, if a history satisfies atomicity, then, by virtue of Theorem 2.1, all of its activities \( w[p] \) and \( rpc[e] \) occur atomically and do not interfere with each other.

6. Correctness criteria for rpc atomicity

We begin this section by introducing the "lies-to-the-left" relation over the hierarchy trees of structured occurrence graphs. This relation is then used to define the notion of monotonicity for ogs, and monotonicity is shown to be a sufficient condition for a sog to satisfy atomicity, i.e. for its activities to be interference-free.

Next we strengthen the condition PO1 of Section 5.1.3 so as to constrain rpc histories to be monotonic and, consequently, free from interference. This provides us with a sufficient condition for an rpc protocol to be correct with respect to interference: if the protocol enforces the new PO1, it will produce interference-free rpc histories. Finally, we discuss the practical significance of PO1.

6.1 The lies-to-the-left relation

Consider a generic sog \( G \). Since simple and compound actions of \( G \) are nodes of its hierarchy tree, we may speak of path (from the root), ancestor and descendant of an action.

The relative position of any two (simple or compound) actions \( A \) and \( B \) as nodes of the hierarchy tree must fall into one of three cases: either the path to \( A \) lies to the left of that to \( B \) (denoted \( A \gg B \)), or vice versa \( (B \gg A) \), or \( A \) and \( B \) are on the same path. The following properties are direct consequences of the definition of \( \gg \).

**Property 6.1**  The restriction of \( \gg \) to an abstraction level is a total relation.

**Proof.** Let \( L \) be an abstraction level of a sog \( G \). We must show that if \( A \in L \), \( B \in L \) and \( A \gg B \), then either \( A \gg B \) or \( B \gg A \) holds. This immediately follows from the Definition 2.5 of abstraction level: a cut of the hierarchy tree of \( G \) cannot intercept nodes that lie on the same path. □

**Property 6.2**  \( \gg \) is a transitive relation. □

**Property 6.3**  If \( A \) and \( B \) are activities of a hierarchy tree, then \( A \gg B \) implies \( a \gg b \) for all simple actions \( a \in A \) and \( b \in B \). □

According to Definition 2.3, the \( \gg \) relation over a sog hierarchy tree, i.e. the left-to-right ordering of the tree nodes, is irrelevant in that it does not affect the abstraction hierarchy defined by the tree. However, for our purposes it is convenient to link the relations \( \gg \) and \( \Rightarrow \).

6.2 Monotonicity and atomicity

**Definition 6.1**  A sog \( G = (Act, \rightarrow, AbsH) \) is said to be monotonic if, for every two simple actions \( a \in Act \) and \( b \in Act \), \( a \gg b \) implies \( \neg(b \Rightarrow a) \).
Theorem 6.4  A monotonic sog satisfies atomicity.

Proof. We must prove that if \( G \) is monotonic, then for no abstraction level \( L \) the og \( G@L \) is cyclic. By contradiction, assume that \( G@L \) is cyclic; then it must contain a cycle \( A \Rightarrow B \Rightarrow A \) with \( A \neq B \), for all its cycles were of the form \( A \Rightarrow A \), the og \( G@L \) would have to contain a self-loop \( A \Rightarrow L \), which contradicts the irreflexivity postulated by Definition 2.1. By the Definition 2.7 of view, this implies that for some \( a_1 \in A, b_1 \in B, b_2 \in B \) and \( a_2 \in A, a_1 \Rightarrow b_1 \) and \( b_2 \Rightarrow a_2 \) hold. On the other hand, since \( A \in L \) and \( B \in L \), by Property 6.1 either \( A \gg B \) or \( B \gg A \) must hold, whence either \( a_2 \gg b_2 \) or \( b_2 \gg a_2 \) must hold, by Property 6.3. In either case the monotonicity hypothesis is contradicted: by \( b_2 \gg a_2 \) if \( a_2 \gg b_2 \), by \( a_1 \gg b_1 \) if \( b_1 \gg a_1 \).

6.3 Implementing monotonicity

Hereafter it will be assumed that the rpc protocol is capable of enforcing a stronger version of the condition PO1 given in Section 5.1.3.:  

PO1 \( \) If \( \text{onsite}(p_1) = \text{onsite}(p_2), a_1 \in s\_\text{actions}[p_1], a_2 \in s\_\text{actions}[p_2], \) and \( a_1 \gg a_2, \) then \( a_1 \Rightarrow a_2. \)

According to Section 2, \( \Rightarrow \) should be the transitive relation satisfying PO1-PO3 and, in addition, as required by DO, containing either \( (a, b) \) or \( (b, a) \) for each pair of simple actions \( a \) and \( b \) that reference the same object. However, as any such \( a \) and \( b \) must occur on the same site, the new PO1 suffices to ensure what required by DO. Thus, in contrast with the conclusions of Section 5.1.3, PO1-PO3 are now enough to determine \( \Rightarrow \) completely. In the sequel \( \Rightarrow \) will be assumed to be the smallest transitive relation satisfying PO1, PO2, and PO3.

6.3.1 Well-definedness

In this subsection we show that the new \( \Rightarrow \) is well-defined, in that it satisfies the old PO1 of Section 5.1.3 and is therefore in accordance with the intended execution model of Section 3. We show first the

Lemma 6.5 \( \) If \( H[p_1] = (Acd[p_1], \Rightarrow, AbsH) \) is a history, \( a_1 \in \text{Act}[p_1], a_2 \in \text{Act}[p_2], \) and \( a_1 \Rightarrow a_2, \) then \( a_1 \Rightarrow a_2 \) implies \( a_1 \gg a_2. \)

Proof. The hypothesis \( a_1 \Rightarrow a_2 \) means that there exists a chain \( a_1 = a_0 \Rightarrow a_1 \Rightarrow ... \Rightarrow a_n = a_2 \) such that each \( a_i \Rightarrow a_{i+1} \) \( (1 \leq i < n) \) directly follows from the application of one of PO1-PO3. The proof that, given such a chain, \( a_1 \gg a_2 \) holds, is by induction on \( n. \)

Basis. Assume \( n = 1. \) If \( a_1 \Rightarrow a_2 \) is implied by PO1, it is PO1 itself that entails \( a_1 \gg a_2. \) If \( a_1 \Rightarrow a_2 \) is implied by PO2 or PO3, \( a_1 \gg a_2 \) follows from the rule H2, Section 5.1.2, for constructing the hierarchy tree. The reader may refer to Fig. 5.2 to satisfy himself that if \( a_1 \Rightarrow a_2 \) follows from PO2 or PO3, then \( a_1 \gg a_2 \) holds (take \( a_1 \Rightarrow [c] \) and \( a_2 \gg a \) for PO2, and \( a_1 = b \) and \( a_2 = r[c] \) for PO3).

Induction. Easily follows from Property 6.2 (transitivity of \( \gg. \))

We may now prove that histories satisfy the condition PO1 of Section 5.1.3.

Theorem 6.6 \( \) Let \( H[p_i] \) be a history and \( p \) one of its computations. Then the restriction of \( \Rightarrow \) to the set \( s\_\text{actions}[p] \) is a total order. Moreover, if \( p = \text{caller}(c), \) then \( i[c] \Rightarrow r[c] \) holds, while \( i[c] \Rightarrow a \Rightarrow r[c], \) with \( a \in s\_\text{actions}[p], \) does not.

Proof. First we prove that the restriction of \( \Rightarrow \) to \( s\_\text{actions}[p] \) is a total order. Let \( a \in s\_\text{actions}[p], \) \( b \in s\_\text{actions}[p], \) and \( a \gg b. \) If \( a \gg b, \) then \( a \Rightarrow b \) and \( \neg(b \Rightarrow a) \) must follow - \( a \gg b \) by PO1, \( \neg(b \gg a) \) because by Lemma 6.5 \( b \gg a \) would imply \( b \gg a, \) but by Property 6.1 \( a \gg b \) and \( b \gg a \) cannot be both valid. Likewise, if \( b \gg a, \) then \( b \gg a \) and \( \neg(a \gg b) \) must hold. In conclusion either \( a \gg b \) or \( b \gg a \) holds.

As for the second part of the thesis, rule H2, Section 5.1.2, for constructing the hierarchy tree entails that \( i[c] \gg r[c] \) holds and \( i[c] \gg a \gg r[c], \) with \( a \in s\_\text{actions}[p], \) does not. Thus \( i[c] \gg r[c] \) follows
from \(i(c) \Rightarrow r(c)\) by PO1; \(i(c) \Rightarrow a \Rightarrow r(c)\), with \(a \in s\_actions[p]\), cannot hold or, by Lemma 6.5, \(i(c) \Rightarrow a \Rightarrow r(c)\) would also hold.

6.3.2 Correctness

The histories considered in this section can be proved to be monotonic and hence, by Theorem 6.4, to satisfy atomicity. This is tantamount to proving correct any rpc protocol satisfying PO1 (in the revised version), PO2 and PO3.

Theorem 6.7 Histories are monotonic.

Proof. By contradiction, assume that a history \(H[p_t]\) is not monotonic, i.e. that for two simple actions \(a\) and \(b\) both \(a \Rightarrow b\) and \(b \Rightarrow a\) hold. Then, by Lemma 6.5, also \(b \Rightarrow a\) should hold, contradicting Property 6.1, which requires that the restriction of \(\Rightarrow\) to simple actions (the least abstract level) should be a total relation.

6.4 The intuitive meaning of PO1

Although the new form given to PO1 is elegant and permits concise proofs, its practical implications may be obscure. The intuitive meaning of PO1 can be best appreciated from the three properties below, which may replace it as correctness criteria.

PO11 If \(onsite(p1) = onsite(p2)\) and \(...w[p1] \Rightarrow rpc[c2] ... w[p2]\) is the path to \(w[p2]\), then \(a2 \Rightarrow r[c2]\) holds for all \(a2 \in s\_actions[p2]\).

PO12 If \(onsite(p1) = onsite(p2), w[p1] \Rightarrow w[p2]\) and the least common ancestor of \(w[p1]\) and \(w[p2]\) is \(w[p]\), then \(a1 \Rightarrow a2\) for all \(a1 \in s\_actions[p1]\) and \(a2 \in s\_actions[p2]\).

PO13 If \(onsite(p1) = onsite(p2), w[p1] \Rightarrow w[p2]\) and the least common ancestor of \(w[p1]\) and \(w[p2]\) is \(rpc[c]\), then \(a1 \Rightarrow a2\) for all \(a1 \in s\_actions[p1]\) and \(a2 \in s\_actions[p2]\).

The above labels and formulations emphasize that PO11-PO13 orderly correspond and cope with the interference patterns 11-13 identified in Section 4.4. The following formulations should be even more intuitive.

PO11 Any work performed on behalf of an rpc on the site where the rpc has been invoked must precede the rpc return (whether normal, abnormal or crashed).

PO12 If a client invokes rpc \(c1\) before rpc \(c2\), any work made on a site on behalf of \(c1\) must precede any work made on the same site on behalf of \(c2\).

PO13 If computations \(p1\) and \(p2\) serve the same rpc, either any work that is part of \(p1\) precedes any work that is part of \(p2\) on every site, or vice versa.

The reader should recall the convention stipulated in Section 4.2 for exacts and therefore extended to hierarchy trees: the children of a computation node are its remote calls, arranged from left to right in the order in which they are invoked. This explains how the two variants of PO12 are related. Note also that the first variant of PO13 provides an interpretation of the left-to-right position of the children of a call node: if \(w[p1]\) and \(w[p2]\) are children of \(rpc[c]\) and \(w[p1] \Rightarrow w[p2]\), then any work that is part of \(p1\) will precede any work that is part of \(p2\) on every site.

7. The implementation of interference prevention

Crashcounts, extermination and expiration are three of the best-known strategies for interference prevention through orphan-killing. Their relative merits are compared in [Ne1], where they were proposed for the Emissary rpc protocol. Expiration and crashcounts have also been adopted for the Rajdoot [PS] rpc protocol.

Restricting ourselves to exactly-once semantics, the one supported by the noted rpc protocols, we shall discuss how the above techniques can be employed to ensure PO11 and PO12. Specifically, extermination and crashcounts cope with orphans left behind by crashes, expiration with orphans
of calls that have returned abnormally because of a timeout expiration. Note that, in accordance with [Nel] and [PS], in this section we make the assumption that a crash affects an entire site rather than individual computations.

7.1 Crashcounts

A site's crashcount records the number of times the site has crashed. Let \( c \) be a generic rpc with each path \( p \), \( c_1 \), \( p_1 \), ... \( c_n \), \( p_n \), \( c \); in [PS] the rpc-path of \( c \) is defined as the list \( \langle (s_1, \kappa_1), (s_1, \kappa_1), ..., (s_n, \kappa_n) \rangle \), where \( s_i = \text{onsite}(p_i) \) and \( \kappa_i \) is the crashcount of \( s_i \). The rpc protocol of [PS] employs crashcounts as follows. The message requesting a computation to be started on behalf of a call contains the rpc-path of the call. Each site uses the rpc-paths received to maintain a table recording the highest known crashcount of the other sites in the network; whenever the crashcount for site \( s \) has to be updated, any outstanding computation for a call whose rpc-path contained \( s \) is aborted. When a request message is received, a computation is started only if no site in the rpc-path has a crashcount lower than that known locally.

The two eacts of Fig. 7.1 illustrate the position of the computations \( p_1 \) and \( p_2 \) in terms of which POI1 and POI2 are formulated.

With reference to Fig. 7.1a, assume that the site \( s \) of \( p_1 \) and \( p_2 \) crashes before \( p_1 \) has returned from \( c_2 \), and that \( p_1 \) is later restarted by the crash recovery procedure. Owing to the crashcount mechanism, no action of \( p_2 \) can take place after the recovery: if the request message that would give rise to \( p_2 \) on \( s \) is received after \( s \) recovers, it will be ignored because it carries an obsolete crashcount for \( s \).† This ensures that if \( r[c_2] \) is crashed, \( a_2 \Rightarrow r[c_2] \) holds for all \( a_2 \in s\_actions[p_2] \), as required by POI1.

As for POI2, with reference to Fig. 7.1b, assume that the site \( s \) of \( p \) crashes and recovers after the invocation of \( c_1 \) and before that of \( c_2 \). Because of the crashcount mechanism, on the site of \( p_1 \) and \( p_2 \) no action of \( p_1 \) can take place after \( p_2 \) has started. Indeed if the request message that should give rise to \( p_1 \) is received after \( p_2 \) has started, it will be ignored because it carries an obsolete crashcount for \( s \). If, on the other hand, the request message that initiates \( p_2 \) is received while \( p_1 \) is still in progress, the latter will be aborted because the message carries a higher crashcount for \( s \) than that registered by \( p_1 \). Thus \( a_1 \Rightarrow a_2 \) for all \( a_1 \in s\_actions[p_1] \) and \( a_2 \in s\_actions[p_2] \), as required by POI2.

7.2 Extermination

Extermination is performed as part of crash recovery; before a computation is resumed after a crash, the rpc protocol will exterminate all the orphan computations left by pre-crash rpcs around the network. Extermination alone does not suffice to guarantee POI1 and POI2, for it is not effective against delayed request messages that were originated by pre-crash rpcs but reach the called site after extermination has completed. For this reason extermination is generally coupled with a crashcount-based mechanism (e.g. in [Nel]). Its advantage is that it minimizes the waste of resource caused by orphans.

7.3 Expiration

Expiration consists in assigning each (non-top) computation a deadline after which the computation will be aborted if still ongoing. Deadlines and rpc timeouts should be chosen in such a way that when an rpc returns abnormally any descendant rpc will have already returned abnormally and any descendant computation will have terminated. Thus, rpcs that return abnormally behave, with respect to interference, like lpcs (cf. N11 and N12, Section 4), and hence satisfy POI1 and POI2.

† Of course, if \( p_2 \) existed before the crash, the recovery procedure should avoid restarting it.
8. Concluding remarks

The independent failure modes introduced by distribution pose reliability problems not normally encountered in centralized systems. Thus the task of implementing the procedure call abstraction in a distributed environment is fraught with considerable difficulties. An important problem is that posed by interference between unwanted computations, the so-called orphans, and the intended one. We have made use of the occurrence graphs computational model to describe distributed programs containing remote procedure calls. Using this model, interference has been characterized rigorously, and a condition that ensures the interference-free execution of programs has been stated and proved. Another original contribution of this paper is the notion of monotonicity for occurrence graphs; we feel that, as in our case, it might often lead to simpler proofs.

Research in this area does not abound. A predecessor of this paper is [Shr], where interference was studied in an informal setting, and correctness criteria similar to our PO11-PO13 were derived. Orphans and interference have been studied in [Nel], with an informal approach and a strong emphasis on implementation. Orphan detection techniques are presented in [Wal] and studied formally in [Gor]; their results are difficult to compare with ours because their objective is to provide at-most-once semantics.

If directly implemented, our correctness criteria might provide an interesting alternative to expiration. The latter is based on prevention: computations expire before they are made orphans by the abnormal return of the call they are serving. However, if time delays are difficult to estimate, it may be more convenient to resort to repression, i.e. to let orphan computations live until they would otherwise actually cause interference. Basically, this can be achieved (i) by incorporating into the rpc request message some encoding of the rpc's exact path, and (ii) by maintaining a database associating every ongoing computation with its exact path (inferred from the request message that originated the computation). Thus, the called site would be able to decide whether starting a computation in response to a request message might violate PO11-PO13, and, if so, whether PO11-PO13 are to be enforced by ignoring the request or by suppressing some ongoing computation. Critical design issues are obviously who should make such decisions and manage the database at the called site, and how to avoid the database becoming a performance bottleneck. Further investigation is required, but a preliminary analysis seems to indicate that, given a

† This notion of path extends that of [PS].
suitable target environment, this repression-based approach could compare favourably with expiration.

Acknowledgements
This work was supported partly by the RSRE of the UK Ministry of Defence. The authors would like to thank Luigi Mancini for many helpful discussions.

References