Abstract:

Failures in a distributed system (such as node crashes) can give rise to unwanted computations referred to as orphans. Orphans can interfere with executions of other programs, thus giving rise to unpredictable behaviour. In this paper a graph model of computation is utilised to discuss the orphan phenomenon in a rigorous manner. In particular, conditions are derived for interference-free executions of programs. In a distributed system consisting of 'clients' and 'servers', where clients interact with servers by making use of remote procedure calls, various techniques for treating orphans under 'at least once' and 'exactly once' call semantics are next investigated.


On the Treatment of Orphans in a Distributed System

By

S.K. Shrivastava

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 188
August, 1983

© 1983 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
SHRIVASTAVA, Santosh Kumar


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1983.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 188.)

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 188.

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.687

Suggested keywords
ATOMIC ACTIONS
DISTRIBUTED SYSTEMS
NETWORK PROTOCOLS
RELIABILITY

Abstract
Failures in a distributed system (such as node crashes) can give rise to unwanted computations referred to as orphans. Orphans can interfere with executions of other programs, thus giving rise to unpredictable behaviour. In this paper a graph model of computation is utilised to discuss the orphan phenomenon in a rigorous manner. In particular, conditions are derived for interference-free executions of programs. In a distributed system consisting of 'clients' and 'servers', where clients interact with servers by making use of remote procedure calls, various techniques for treating orphans under 'at least once' and 'exactly once' call semantics are next investigated.

About the author
Dr. S.K. Shrivastava joined the Computing Laboratory at the University of Newcastle upon Tyne in August 1975 where he is a lecturer.
ON THE TREATMENT OF ORPHANS IN A DISTRIBUTED SYSTEM

S.K. Shrivastava
Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne, NEI 7RU, England

Abstract
Failures in a distributed system (such as node crashes) can give rise to unwanted computations referred to as orphans. Orphans can interfere with executions of other programs, thus giving rise to unpredictable behaviour. In this paper a graph model of computation is utilized to discuss the orphan phenomenon in a rigorous manner. In particular, conditions are derived for interference-free executions of programs. In a distributed system consisting of 'clients' and 'servers', where clients interact with servers by making use of remote procedure calls, various techniques for treating orphans under 'at least once' and 'exactly once' call semantics are next investigated.

1. Introduction
Orphans are unwanted computations of a distributed system that get scheduled due to the occurrence of various undesirable events such as message duplication and node crashes [1,2]. We will consider a distributed system consisting of 'clients' and 'servers' where clients interact with servers by making use of remote procedure calls. In such a system we will investigate various situations that can give rise to orphans and discuss techniques of dealing with them. Let us first informally examine the orphan phenomenon.

Figure 1 depicts a message exchange for a call between a client and a server. We assume that the data communications layer, which provides 'send' and 'receive' operations, uses (in the interest of fault tolerance) a message retry facility whenever a loss of message is suspected. This could occasionally give rise to multiple executions at a server when only one execution was intended. In figure 1 above, if the client accepts a result message pertaining to 'work2' then all the other messages reaching the client will be rejected, and in particular, 'work1' will become an orphan execution. Orphans could also be created due to node crashes, as illustrated by figure 2.

Figure 2 Crash Orphan

The client crashes just after sending a message to the server. If after recovering from the crash, the client repeats the call, then depending upon which 'result' message is accepted, either 'work1' or 'work2' will be an orphan. In both of the cases illustrated by the two figures, orphans do not pose any serious problems if the work being performed by a server has the idempotency property (i.e., repeated executions equivalent to a single one). Since in general this can not be guaranteed, techniques must be devised for treating orphans. (Incidentally, even for idempotent calls, as we will see, the test for orphan treatment can not be ruled out entirely). In the next section we will introduce a graphical notation for describing events occurring in a distributed system and will use that notation to describe remote procedure calls and to discuss the orphan phenomenon in a rigorous manner. In particular, we will discuss implementation requirements for calls with 'at least once' and 'exactly once' semantics.

2. Event Graphs
Rather than employing diagrams as used in the previous section, we will use a more concise notation for describing the events of interest to us. We assume that the computation of a process can be viewed as a sequence of events.

A directed acyclic graph whose vertices and arcs are interpreted as events and precedence relations between events respectively, is called an event graph or occurrence graph (1). If there is an arc from event $a_i$ to $a_j$, or a (directed)
path from \( a_1 \) to \( a_2 \), then we say that \( a_1 \) precedes or "happens before" \( (a_2 \) succeeds or "happens after") \( a_2, a_3 \), and we will use the notation \( a_1 \rightarrow a_2 \) to indicate this fact. An event graph is shown in figure 3(a) where \( a_1 \rightarrow a_2 \) has been used to represent an event. Two events \( a_1 \) and \( a_2 \) are said to be concurrent if \( a_1 \rightarrow a_2 \) and \( a_2 \rightarrow a_3 \). In figure 3(a), we have for example, \( a_1 \rightarrow a_2, a_3 \rightarrow a_4 \), and \( a_5 \rightarrow a_6 \).

Figure 3 Event Graphs

We say that an event \( a_1 \) happens immediately after \( a_1 \) (is immediate successor of \( a_1 \) or \( a_1 \) is immediate predecessor of \( a_1 \)) if \( a_1 \rightarrow a_2 \) and for no \( a_3 \) \( a_3 \rightarrow a_4 \). The notation \( a_1 \rightarrow a_2 \) will be used to indicate the immediate predecessor/successor relationship between events. In figure 3(a) for example, \( a_5 \rightarrow a_6 \) and \( a_6 \rightarrow a_7 \).

We will use event graphs as a means of describing a computation, and assume that there exists a most detailed description of the computation termed the basic graph. Events of the basic graph - basic events - represent "atomic" (indivisible) occurrences. A basic graph can be converted into another graph describing the computation in not so much a detail by "collapsing" a set of events into one composite event (3). For example, figure 3(b) represents a graph converted from the graph of figure 3(a), where the composite event \( A \) has been formed out of events \( a_1, a_2, a_3, a_4, a_5, a_6, a_7 \). The collapsing operation will next be defined.

In an arbitrary event graph, a subgraph generated by a subset of events \( E \) is defined as that subset together with all the arcs which have both end points in \( E \). Thus, in figure 3(a), \( A \) is a subgraph generated by the set \( \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\} \) of events. For a subgraph \( A \), the following two sets of events are of interest:

- \( A' \) of immediate successors of \( A \)
- the set \( A' \) of immediate predecessors of \( A \)

A subgraph \( A \) of an event graph can be collapsed by replacing the whole subgraph by a single composite event labelled \( A \) and by connecting \( A \) to \( A \) and \( A \) to \( A' \) (see figure 3(b)). For any acyclic event graph, we will allow only those subgraphs to be collapsed such that the resulting new graph is also acyclic. This rule allows us to view composite events as atomic occurrences (3). As an example, in figure 4(a), composite events \( A_1 \) and \( A_2 \) cannot be viewed as atomic since neither \( A_1 \rightarrow A_2 \) nor \( A_2 \rightarrow A_1 \) are concurrent; whilst the collapsing shown in figure 4(b) is permissible since the resulting graph is acyclic. Note that in figure 4(a), it is possible to regard \( A_1 \) only as atomic. Thus viewing a subgraph as an atomic occurrence also depends on what other subgraphs have been chosen for collapsing.

Figure 4(a) Invalid Collapsing

Figure 4(b) Valid Collapsing

Remarks:

Remark 1: If two subgraphs are collapsible then we say that the computations being described by them do not interfere (and therefore these computations can be viewed as atomic occurrences or simply, atomic).

Remark 2: The above atomicity criterion is frequently referred to as "atomicity with respect to concurrency" and should not be confused with another atomicity criterion referred to as "atomicity with respect to failures" (which states that a computation either terminates normally or is aborted with no side effects being produced). It is the former criterion that will be used here.

Remark 3: An execution of a single "conventional" sequential program gives rise to a computation that is atomic (as there is no other computation interfering with it). This need not be true for a sequential program making use of several procedures by means of remote procedure calls (due to the possibility of orphans). Conditions under which atomicity can be maintained in the presence of orphans will be investigated in subsequent sections.

1. Remote Procedure calls and Orphans

We will use event graphs to describe computations of a distributed system. Since our primary interest is with orphans of remote procedure calls, we will concern ourselves with the description of sequential client programs consisting simply of remote procedure calls. Such a computation is composed out of the following types of basic events:

(i) The call event \( r \): The effect of this event, which occurs in a client, is to send one or more call messages to the appropriate server.

(ii) The return event \( r \): This client event represents the successful termination of a remote call, and occurs as soon as a result message from the server is received.

(iii) The time-out event \( r \): This client event signals an abnormal termination of a remote call (as such its occurrence implies that the return event \( r \) will not take place).
(iv) The work event \( w' \): This server event takes place after the receipt of a call message from a client, and represents some work done by a server. (The overall work done by a server for a call can include a number of such work events).

(v) The send event \( 's' \): This server event represents the sending of results by a server through a message to a client.

(vi) The crash event \( 'c' \): This event (which occurs spontaneously) represents the crash of a node followed by its recovery. No processing takes place during a crash; further, the node is reset to some initial state. Let \( a_0, a_1, \ldots \) be the basic events at a node, and let \( a_c \) be a crash event. Then the following predicates states

\[
\forall a_i, 1 \leq k \leq (a_1, a_k, v a_k, a_0)
\]

that no event of a node occurs concurrently with a crash. We will also assume that consecutive crashes of a node are equivalent to a single crash; \( cr \). We will use the following notation to identify various events:

\[ c_{ij} \] : \( i \)th call event on node \( j \). The first superscript is used to uniquely identify a call event among the set of all call events. The events \( r, t, s \) and \( w \) can only occur as a result of an occurrence of an event \( c \), so we will use a notation that will identify to which call event these events 'belong', \( t^{(i)}(c_{ij}) \): a time out (return) event for \( i \)th call on node \( j \).

\[ s_{ik} \] : \( k \)th send result event on node \( k \) for call event \( i \).

(Where convenient we will drop the superscripts).

To start with we will use event graphs to describe a simple remote call. A call can terminate in one of the three nodes, namely normal termination; abnormal termination; or crashed termination.

The following assumptions will be made:

(i) The 'maximally concurrent' invocation scheme is being utilised. In this scheme, a server consists of a group of work processes and each request from the client is assigned to a worker process which after performing the work, sends the results through a message. (Note that a single call event can result in multiple requests).

(ii) The work at a server involves access to some global data held at the server. So basic work events of a server cannot occur concurrently.

(iii) 'Send result' events of a server are concurrent.

(iv) After a crash of a server's node, the server is initialised to a state ready to receive further work requests (if the server was in the middle of a computation, then the pre crash computation is not resumed).

(v) Work performed by a server (some arbitrary sequential computation) for a single request can be regarded as an atomic occurrence with respect to other executions of the work (the notation \( W^j_k \) will stand for \( j \)th repetition of work performed by server on node \( k \) for call \( i \)). Subsequently we will examine what is required to guarantee this atomicity. The work performed is however not atomic with respect to a crash of the server (only basic events are atomic w.r.t. a crash). So, we will interpret the relation \( W, r \) or as "the execution of work \( W \) is interrupted by a crash" (whether the crash occurred during or just after the execution need not concern us).

3.1 Normal Termination

A normally terminated call is composed out of events \( c, r \) (at a client) and one or more intervening occurrences of \( W, s \) and zero or more occurrences of \( cr \) (all at a server). If, for some \( W^k, W^r \), \( i \), \( k \) then the corresponding \( W^k \) event does not take place. The return event occurs

\[
\begin{align*}
& W^k \quad i \quad j \quad W^k \quad i \quad j \quad W^k \quad i \quad j \quad W^k \quad i \quad j \quad W^k \quad i \quad j \quad W^k \quad i \quad j \\
\end{align*}
\]

Figure 5 A normally terminated call

Definition: The call event \( c_{ij} \) of a normally completed call can give rise to an orphan event on node \( k \). If there is an event \( W^k \), such that there is no \( W^k \), \( c_{ij} \), then there is an event \( W^k \), \( r \), \( n \), such that \( W^k \), \( c_{ij} \), \( r \), \( n \), \( W^k \) is then termed an orphan event.

If a work event of a server is not followed by the corresponding return event and also there is no interfering crash of the server then such an event will be an orphan. If figure 5 for example, \( W^k \) and \( W^r \) are orphans but \( W^i \) and \( W^j \) are not orphans because of the crash event \( c_{ik} \).

Remark 1: 'Opening up' an event \( W \) to view a more detailed description could reveal orphans on other nodes (\( W \) might itself contain remote calls).

Remark 2: The following general behaviour of a client and a server is being modelled: the client keeps on sending requests to the server until a result message is obtained. The server on the other hand receives a request, assigns it to a worker process which performs the work and sends...
the results. Requests arrive at the server in an arbitrary order and are executed in an arbitrary order. Messages can get lost. In Figure 5, the send result messages resulting from events $a_i$ and $a_j$ either are lost or arrive at the client after the message resulting from $a_k$ (in which case they are not accepted).

**Lemma 1:** In an event graph describing a single normally completed call, the subgraph $W^{ik}$ generated by the events of the server can be collapsed into a single event labelled $W^{ik}$, and the new graph can be further collapsed into a single event.

**Proof:** The proof follows from the following three observations:

(i) all the events of subgraph $W^{ik}$ are atomic (this is because of our assumption that $W^{ik}$ are atomic);

(ii) if $E'$ is the set of events of subgraph $W^{ik}$, then

$$\forall a_i \in E': (c^{ij} \leftarrow e_1)$$

and

(iii) $\forall a_j \in E': (a_j \rightarrow r^{ij} \leftarrow e_2 || r^{ij})$.

Hence the relation $c^{ij} \leftarrow W^{ik} \rightarrow r^{ij}$ will hold, making it possible to collapse the three events into a single one.

3.2 Abnormal and Crashed Termination

A single, abnormally terminated call is composed of event types $c$, $t$, and optionally $s$ and $cr$ (at a server). At the client, $c^{ij} \leftarrow r^{ij}$ holds while at the server, there could be zero or more occurrences of work events, send events and crashes.

**Figure 6: An abnormally terminated call**

**Definition:** An abnormally terminated call generates an orphan on node $k$ if there is a $W^{ik}$ such that there is no $c^{ik}$, $W^{ik}$, or $W^{jk}$. The event graph of Figure 6 describes an abnormally terminated call where $W^{ik}$ and $W^{jk}$ are orphans.

**Lemma 2:** In an event graph describing a single abnormally terminated call, the subgraph $W^{ik}$ generated by the events of the server can be collapsed into a single event $W^{ik}$ and the resulting new graph can be further collapsed into a single event.

**Proof:** Similar to that of Lemma 1.

An abnormal termination occurs when the client (after possibly several attempts) does not receive any result message from the server and therefore 'times out'. A crashed termination (crashed call) is similar to an abnormal termination except that the client crashes after making a call $(c^{ij} \rightarrow r^{ij})$ thus giving rise to the possibility of orphans.

**Lemma 3:** In an event graph describing a single crashed call, the subgraph $W^{ik}$ generated by the events of the server can be collapsed into a single event $W^{ik}$, and the new graph further collapsed into a single event.

3.3 Multiple calls

So far we have considered execution of a single call. We will now consider the implications of a client making a sequence of calls to a server. For any two calls, Figure 7 shows all possible combinations of events at a client between two call events.

(i) $c^{ij}$ follows a normally terminated call.

(ii) a crash event (cr) occurs between an abnormally terminated call and the event $c^{ij}$.

(iii) a new call is made after an abnormally terminated call.

(iv) a crash occurs after a normally completed call but before a new call is made.

**Figure 7: Possible events at a client between two calls**

We have already assumed that a server event $W^{ik}$ occurs atomically with other events $W^{ik}$, $W^{jk}$, $W^{ik}$. We therefore need some way of ensuring that the subgraph $W^{ik}$ and $W^{jk}$ generated by the server's events for calls $c^{ij}$ and $c^{jk}$ respectively are both collapsible to single events. This can be guaranteed if all the events of call $c^{ij}$ precede those of $c^{jk}$. This is stated formally in the theorem that follows:

**Theorem 1:** An event graph describing a sequence of calls of a client can be collapsed into a single event if condition $R_1$ is satisfied between call events.

$$R_1: c^{ij} \rightarrow c^{jk} = W^{ik} \leftarrow (W^{ik} \rightarrow W^{jk})$$

where $E_n$ and $E_k$ are sets of events generating subgraphs $W^{ik}$ and $W^{jk}$ respectively.
Proof: Let \( C^1 \) be the first call event of the sequence and \( C^1 \) the call graph for this call. From Rule 1, we find that all the events are not in \( C^1 \). So the call graph \( C^1 \) must succeed the events of \( C^1 \). Hence, the call graph \( C^1 \) can be collapsed into a single event, \( C^1 \) (this follows from Lemma 1 or 2 or 3). Now we have one of the two situations: (1) either \( C^1 \) or (ii) \( C^1 \) or \( C^1 \) (see Figure 7) for case (i), the two events can be collapsed into a single one. For case (ii), we note that excluding \( C^1 \), all the events that are not in the call graph \( C^1 \) must succeed the events of \( C^1 \). Hence, the call graph \( C^1 \) can be collapsed into a single event \( C^1 \), and \( C^1 \) or \( C^1 \) can be further collapsed into a single event. Such collapsing can be carried out successively to reduce the entire graph to a single event.

We next investigate the condition necessary to make \( W_j \), the computation performed by a server (as a result of receiving a request from its client) atomic w.r.t. some other \( W_i \). Such computations can be described as sequences involving basic work events of the server and remote calls (see figure 8). Condition R2 of Theorem 2 states the requirement for \( W_i \) and \( W_j \) to be atomic (note that R2 says nothing about events not local to a server).

![Figure 8: Computation of a server](image)

Theorem 2: Let \( W_i \) and \( W_j \) be two subgraphs describing two computations of a server and each subgraph meet condition R1. Then \( W_i \) and \( W_j \) are both collapseable to single events if condition R2 holds:

R2([\( W_i, j: (W_i, j: W_i, i, V, W_i, Z)] V (W_i, j: W_j, j, V, W_j, Z)])

Proof: We note that all the call events of \( W_i \) are concurrent with those of \( W_j \). As such all the subgraphs representing calls can be collapsed individually (Theorem 1). The new graph can be further collapsed to single events since R2 guarantees that this collapsing will not lead to a cycle.

Conditions R1 and R2 lead to implementation rules IL1 and IL2 respectively for a server.

IL1: Once the execution of a call starts, computations belonging to prior calls (if any) must never be allowed to be scheduled (i.e., all the orphan of prior calls must precede this call).

IL2: Local computations performed by worker processes of a server as result of a single call must not interfere with each other.

If all the servers follow rules IL1 and IL2, an execution of any sequential program containing remote calls (which can be nested) will lead to a computation in which all the calls (including the nested ones) can be viewed as atomic occurrences, despite the presence of orphans.

4. Orphan Treatment and Call Semantics

4.1 From a semantic point of view, the presence of orphans in a computation could prove to be undesirable. For example, suppose the work requested by a client involves the server making an increment operation, then clearly more than one execution would be erroneous. On the other hand, work can be tolerated if a server's operation is idempotent (e.g., a disc write operation). We identify two types of call Semantics: (i) at least once, and (ii) exactly once.

At least Once Semantics:

As the name suggests, a normally terminated call implies that the server has performed the requested work one or more times. It can be seen that we have implicitly assumed this semantics in the previous discussion. This semantics is 'least demanding' since presence of orphans is tolerated. Such calls are of limited use however - they can only be used meaningfully where a server's work is idempotent. Since disc read and write operations have this property, many distributed systems containing file servers have opted for this call semantics [4]. In the following, the idempotency property will be assumed for such calls.

Exactly Once Semantics:

A normally terminated call results in only one execution at a server. If a call gives rise to nested calls then they must also be 'exactly once' calls. This definition implies that no orphan execution takes place. Exactly once semantics represents the natural interpretation of a procedure call, being appropriate in the general case when a server's work need not be idempotent.

4.2 Orphan Treatment

In this section we will discuss ways of meeting requirements IL1 and IL2, bearing in mind any additional requirements posed by a particular call semantics. Orphans can be prevented, tolerated, or killed. As we will see, no single measure is adequate by itself and any well engineered system will employ a mixture of these techniques. As the name suggests, preventive measures are intended to ensure that orphans simply do not occur, whilst tolerating orphans essentially implies 'learning to live with them' (as in 'at least once' calls). Finally, orphan killing measures imply the need for systemically detecting orphans and then aborting those computations. We will discuss the relevance of these measures for normally com-
completed calls, abnormally completed calls and crashed calls.

4.2.1 Normal ‘at least once’ calls

Let us first consider an elementary call: server's work is local, that is, does not contain any remote calls and atomic, e.g. a disc write operation. We see that by definition condition I(2) is ensured and it is only required to maintain I(1). A rather ingenious way of meeting this requirement is to design a protocol which guarantees that, for any call, there is no wait, such that if, that is, the return event is the last event of a call and all the orphans precede it (the ‘last of many’ semantics [2]). Implementing ‘last of many’ semantics certainly is a possibility for elementary calls.

Consider instead the general case where a server's work is some arbitrary computation, including calls on other servers. We will certainly need some means of meeting requirement I(1). Also for such calls it is very difficult to implement ‘last of many’ semantics. (Recall that messages may be delivered and work scheduled in any order.) So that we require some technique for meeting requirement I(1). We thus see that tolerating orphans for general ‘at least once’ calls is not really an attractive proposition and that it is far better to use preventive and/or killing measures. In this case ‘at least once’ calls should be abandoned in favour of ‘exactly once’ calls. Henceforth we will assume that ‘at least once’ calls are used only for performing (indispensable) elementary work.

4.2.2 Normal ‘Exactly once’ calls

The requirement for exactly once calls can be best illustrated by an example. Figure 9 shows an event graph depicting a normal call. Let us assume that any results produced by are destroyed (undone) as a result of the crash. Then the call of figure 9 can certainly be taken to be of an ‘exactly once’ variety. In general however, such an assumption about any work done cannot be made (for example might involve calls on other nodes of include disc write operations that survive crashes and so forth). We are therefore led to the conclusion, that ‘exactly once’ calls need not incorporate means of tolerating crashes of servers: So, a crash or a server in the middle of a call should result in an abnormal termination.

Fortunately, it is relatively straightforward to prevent orphans in the absence of crashes. A simple sequence numbering scheme where the server has always been performed the same number will do as the server will be in a position to discard duplicates. After a crash, a server should be in a position to discard pre-crash requests. This requires that after a crash, a server should be initialized to accept messages with sequence numbers greater than those of pre-crash messages. The remote procedure call mechanism described in [5, 6] meets this requirement by making use of loosely synchronized node clocks for generating sequence numbers.

4.2.3 Abnormal Termination of Calls

A time out event results in an abnormal termination (the client receives an exceptional return) and the fate of any orphans is determined by the call semantics and the subsequent actions of the client. We will use angular brackets to enclose the exception handler for a time out. Consider now the two possible situations:

(i) \[ \text{call} (...) <t \text{: retry}(n) \text{<t : } \text{"kill orphans"}>>; \]

(ii) \[ \text{call} (...) <t : \text{"kill orphans"}>>; ----- \]

The first situations represents the case where the client retries the call (a maximum of times) in order to get a normal return. For ‘at least once’ calls, no special measure is necessary (other than those mentioned already), so long as a retry succeeds. In the case of ‘exactly once’ calls, the interpretation should be ‘if a retry succeeds then only one execution has taken place’. What should happen when all the retries fail i.e., terminate abnormally? We require that any orphans should be killed before they start interfering with subsequent calls. We have indicated this by a comment. The second case is just a special case of this first one. The client 'does some thing else' after an abnormal return; so, as before, orphans must be killed.

Let us return to the first case, and consider a rather appealing solution for 'exactly once' calls. We know that during retries, further orphans could be created which for 'exactly once' calls must be destroyed. This is rather cumbersome, so an alternative is to adopt an orphan adoption is illustrated in figure 10(a), where 'rt' represents a retry call event. If a server has already performed the work, then rather than

![Figure 10](attachment:figure10.png)

* Ideally, killing should be such that non-complete calls produce no side effects.

---

* Suitable crash recovery techniques can certainly be employed to provide such a guarantee; however such techniques are better employed within the framework of atomic transactions to be discussed in section 4.4.
aborting it, it is 'adopted' for the retry call on the other hand, if no orphan exists (figure 10(b)) then an rt event generates a work event at the server.

4.2.4 Crashed Calls

We consider three possible cases: (i) client crashes and then the call is retried; (ii) client crashes, and then 'does something else'; and (iii) client crashes and remains crashed. As far as orphan treatment is concerned, the first two cases resemble the cases we have just discussed in the previous subsection. However, there are some implementation problems peculiar to crashes. A crash has the effect that most (if not all) of the state information regarding a computation is lost. This means that arranging retries across client crashes is usually not possible, (or considered practical) unless state information for a call, such as a unique call identifier, is maintained on stable storage (a storage medium assumed to be crash-proof, such as dual disc (S)). This being a costly proposition, it is preferable to treat case (i) as equivalent to case (ii) and to rely on one or more orphan killing techniques. The third situation can be dealt with if servers can detect client crashes and take necessary actions (see the next section).

4.3 Orphan Killing

Orphan killing techniques have been discussed in the cited references [1, 2] so we will not elaborate them in detail. For the sake of completeness, we will discuss them briefly and mention a technique that has not been described before.

We need the facility for killing orphans under two circumstances: after an abnormal termination and after a crash. For the former, a server node needs an abort facility as a server is required so that a client can abort orphans (if any). Let us assume a special server primitive 'abort-call' which can be invoked by a client. Then a typical usage is shown below:

---
call (-) let :call (abort-call)let :wait(t)---
The call returns abnormally, so 'abort-call' is invoked (also through a remote call). What if 'abort-call' also returns abnormally? A reasonably safe assumption is that the server has crashed - but we do not know whether this crash has created orphans on some other nodes. A sensible strategy is to retry a few times and if the situation persists, either to ignore the exception and continue (so we are prepared to tolerate occasional orphans) or if this is not acceptable, to rely on some orphan killing technique that will guarantee that orphans will be exterminated within some duration, say T (in which case, the computation should be delayed for T seconds).

Such orphan killing techniques will be described shortly. The 'abort-call' operation should be idempotent and could involve a server making further such calls. A server should make sure that all the orphans have been aborted and no further activity from that call is allowed to take place.

A more powerful abort facility is needed when a client node crashes, since all outstanding calls from that node need attention. Orphan killing can either be initiated by a client node or by a server node. In the former, a list of nodes to which calls have been made is maintained on a stable storage by a client. The crash recovery procedure of a client then involves making abort requests to all the nodes on the list. There are three shortcomings of this approach: first, a stable storage is needed (this is a minor criticism); secondly if a server node is down (or cannot be reached due to some link failure) then orphan killing can not be completed until the node comes up; and lastly, if the client node does not recover, then orphans never get killed. The major advantage is that in the absence of crashes, very little overheads are generated.

Server initiated orphan killing techniques apparently have not received any attention in the literature, yet a particularly attractive method can be devised. We assume that every node maintains a 'down' list which shows which nodes have crashed. Whenever a node detects that some node, not already on its down list has crashed, then an entry is made and any calls on that node are aborted. Let every node update its list every 'd' secs. Then, when a node crashes, it will be noticed by every other 'up' node within the duration d + cmax, where 'cmax' represents the maximum error by which the interval 'd' as calculated by a clock can vary. Whenever a node crashes, it is guaranteed that its orphans will be dead by the time d + cmax + t, where t is the maximum time taken by a node to kill orphans.

Thus all that is required is that crash recovery time of a node be made greater than d + cmax + t. The price paid is that this is the cost of maintaining the down list (however such a list can be of benefit to other higher level protocols(7) and also on some local area networks it is relatively easy to maintain such a list(8)).

A solution based on synchronised clocks is also possible whereby a time limit is assigned to each call. At a server's node calls whose time limits have expired are killed by some special process. Choosing the correct time limit is however difficult; the main advantage is that once the limit is chosen, no further action is necessary.

It appears that no single technique is satisfactory by itself and combination of a few might be required. Little or no practical experience on orphan killing techniques exists (the author is not aware of any implementations) and there is a real need for experimental work in this area.

4.4 Orphans and Robust Atomic Actions

In this subsection we will examine the relevance of orphan killing techniques within the framework of robust atomic actions (atomic trans-
actions). It is often desirable that a program be made atomic with respect to certain failures, such as node crashes. We will refer to such a program as a robust atomic action. Construction of such programs in a distributed system has received wide attention [6] and we know that various reliability measures are necessary (including a termination protocol known as the two phase commit protocol) to guarantee the property that either a program terminates normally or the program is aborted with no results being produced. Given the provision of robust actions, the question arises as to whether there is a need for orphan killing techniques for such programs. It is possible to rely entirely on the reliability measures of robust actions (and thereby do away with the need for killing orphans) if the following rules are observed? (i) if a call returns abnormally, the action is aborted; (ii) a crash (of clients or servers) results in the abortion of the action. By these means an action is aborted whenever there is the possibility of orphans being created. However, if it is required that robust actions should complete normally in the presence of abnormal returns and crashes, then orphan killing techniques will generally be required. A slightly different approach is also possible whereby the remote procedure call mechanism is designed specifically with robust atomic actions in mind [10]. It is necessary for a client to record on stable storage certain state information regarding the call being made. Both the client and server make use of this information to ensure that orphans are eventually aborted.

5. Concluding Remarks

Independent failure modes inherent in a distributed system pose reliability problems not normally encountered in a centralised system. Thus the apparently simple task of implementing a procedure call abstraction is fraught with reliability problems that are surprisingly intricate. One such problem is that of orphans - unwanted computations that get scheduled due to failures such as lost messages and node crashes. We made use of a graph model of computation to describe distributed computations containing remote procedure calls. Using such a formalism orphans phenomenon was discussed rigorously and necessary conditions for interference free executions of programs were stated and proved. Orphan treatment issues were then discussed for normal, abnormal and crashed calls with 'at least once' and 'exactly once' semantics. The current popularity of remote procedure calls in existing distributed systems is largely based on the fact that the underlying protocol can be made quite simple. However, effective orphan killing techniques must be devised before remote procedure calls can become a general and efficient tool for distributed programming. As observed in section 4.1, there is much scope for experimental work in evaluating various orphan killing techniques that have been proposed in the literature.

6. Acknowledgements

Part of the work reported here was carried out while the author was a visiting scientist at the School of Automation, Indian Institute of Science, Bangalore, India.

References