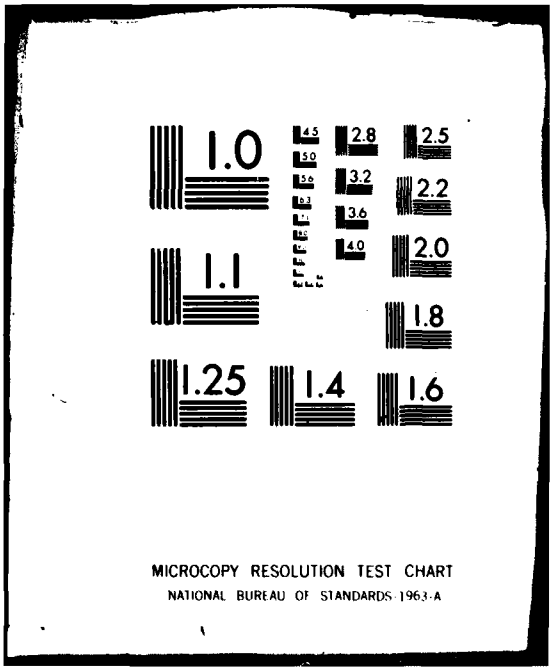
AD-A093 463 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
CHECKPOINTING AND ERROR RECOVERY IN DISTRIBUTED SYSTEMS,(U)
SEP 80 J A MCDERMID
UNCLASSIFIED RSRE-MEMO-3271 DRIC-BR-76154 NL

END
DATE FILMED
2-81
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A093463

(18) DRIC (19) BR-76154

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3271

(6)

TITLE: CHECKPOINTING AND ERROR RECOVERY IN DISTRIBUTED SYSTEMS

AUTHOR: (10) J. A. McDermid

DATE: (11) September 1980 (12) 27

(14) RSRE-MEMO-3271

SUMMARY

This paper discusses some of the problems of producing fault tolerant distributed computer systems, in particular those of software error recovery. It shows how checkpoints may be used in error recovery, it defines the information that checkpoints must contain, and discusses alternate strategies for checkpointing. It describes models of error recovery and extends an existing recovery protocol to cater for certain types of checkpoint inconsistencies. The paper defines protocols for systematically generating checkpoints so that they can be used by the recovery protocols. It also defines a protocol for discarding checkpoints when they are no longer "of use", which prevents the set of checkpoints growing indefinitely. The paper concludes by considering some of the problems of implementing the protocols.

Accession For
NTIS GRA&I X
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Special
A

This memorandum is for advance information. It is not necessarily to be regarded as a final or official statement by Procurement Executive, Ministry of Defence

Copyright
C
Controller HMSO London
1980

Fig -A-

401121

0) Introduction

0.0) Background

This paper discusses mechanisms which allow the software in a distributed computer system to continue working despite the failure of some of the hardware components of the system. Many of the mechanisms would also be applicable to recovery from software faults, but this aspect of fault tolerance will not be discussed here. The background to this research is described in [1].

The mechanisms described are presented in a general way as they are not specific to one particular system. However in order to elaborate on some of the points reference is made to systems on which the author is working. Reference is made to the FLEX computer system [2], a novel, high level language oriented, multi computer system which incorporates a very high speed packet switching communication system known as COMFLEX [3]. The mechanisms described are most likely to be appropriate to large, real time, computer systems running a static set of communicating processes. The process organisation referred to in this connection is known as POSER [4]. POSER enforces a message passing communication discipline, and prohibits shared data areas, but is otherwise similar to MASCOT [5].

The mechanisms discussed aim to provide what is known as backward error recovery (BER) [1], [6]. BER involves forgetting work which may have been performed erroneously due to a fault, and then repeating the work (hopefully correctly) after the faulty component has been removed from the system. With this form of error recovery some work may be repeated but, in principle, none should be lost. It is beyond the scope of this paper to discuss the merits and demerits of alternative recovery strategies, but it is worth pointing out that it is possible for an operating system to perform BER "automatically", thus providing a "recoverable virtual machine" to application processes. This will often be a desirable way of building a system.

Definitions of important terms, such as fault, can be found in [1] and [6]. The definitions used stem from those given in [7] which regard a fault as the mechanistic cause of an error, which is a detectable, incorrect, state and which, if uncorrected, will lead to the failure of the system. We cannot prevent faults and errors, but we do aim to prevent them from leading to a system failure.

0.1) Requirements

The fundamental reason for checkpointing in a fault - tolerant computer system is to preserve enough information about the execution of the system so that the system can be restarted after a fault has occurred. In other words we are storing the information necessary to enable backward error recovery to be performed. The technique chosen for

checkpointing must satisfy a number of subsidiary requirements, the most salient of which are:

1) The system must be restarted, from a globally consistent state, with the minimum repetition of work consistent with purging the fault and consequent errors from the system.
2) The amount of information which has to be stored must be minimised. This can be achieved by:
   storing only "long - lived" data, and ensuring that the amount of it stored is minimal;
   minimising the frequency of checkpointing.
3) It must be possible to restart one part of the system which has been affected by a fault without unnecessarily disturbing the rest of the system.

These subsidiary requirements are not "absolute", in fact, requirements one and two are to some extent contradictory, but they must in some measure be met to ensure that the checkpointing technique chosen is practicable. The other constraints on checkpointing are described as appropriate below.

0.2) Global Checkpoints

Perhaps the most obvious way of checkpointing is to take copies of the state of the entire system from time to time. This is often done in practice for long running programs on mainframe computers, to reduce the amount of work lost in the event of a system crash. The checkpointing technique is normally to periodically halt the system and copy a complete store image, or the part of it containing variables, onto a more permanent medium such as a disc. This technique can, in principle, be extended to a distributed computer system simply by copying the store images of all the programs in the system onto disc at the same time. We refer to this type of checkpointing as global checkpointing. Obviously global checkpointing violates our third requirement. Requirement one can be met by checkpointing sufficiently frequently. Requirement two is almost certainly not met, as copies of nonessential (non long - lived) data will almost inevitably be made.

There are practical problems associated with taking global checkpoints in a distributed system. Primarily these are:

1) Synchronisation between the different parts of the system when taking the checkpoint. This either requires synchronised/central clocks and prearranged checkpointing times, or a protocol to achieve synchronisation. The clocking arrangements reduce the independence of the processors and could introduce undesirable common mode failures. The synchronisation protocol would be difficult to implement and would be very costly in time, particularly if the computer system was physically widely distributed.
2) The whole system will stop for a period of time whilst the

checkpoint is being taken, which may well be unacceptable in a real time system.

These problems render global checkpointing infeasible in a practical distributed computer system, and it will not be considered further.

0.3) Distributed Checkpointing

Any other checkpointing technique must involve copying the states of individual components in the system from time to time - this technique is known as distributed checkpointing. These checkpoints will be taken largely independently, thus the most fundamental difficulty is to ensure that we can construct an internally consistent system state from these component states. By internally consistent we mean that the set of states is one which could validly have existed at some time whilst the system was executing correctly.

That consistency is a problem may easily be shown by example: consider the simple case of two processes running in parallel, and passing messages to each other. The interactions are shown (in the time domain) in figure 0.0, together with a possible set of checkpoints. It is clear that any attempt to establish a consistent pair of checkpoints for the two processes will lead to the trivial pair (the initialisation condition) being established, as no other pair could validly have existed at the same time.

For distributed checkpointing to be valuable we must ensure that consistent, and non - trivial, sets of states can be established in error recovery. We can achieve this by taking checkpoints at the times of interaction with other components. This may result in more information being stored than is necessary to provide the speed of recovery required by a given application system, but it provides a good solution to the third requirement as will be shown later. In the remainder of this paper we shall assume that checkpoints are taken at every interaction, unless otherwise stated.

We need to record information about causes and effects in order to determine which set of checkpoints represents a consistent system state. We will refer to this information as causal relations or dependencies.

Note that the internal system state being consistent does not imply that this state accurately reflects the outside world (environment), indeed it is to be expected that a discrepancy between the world and the systems view of it will arise during error recovery.

Note also that, since we are dealing with distributed systems, we must define the mechanisms for generating and maintaining checkpoints, and for error recovery so that they are capable of distributed execution. We must also assume that subsets of the set of checkpoints may be stored in different places and define our mechanisms accordingly.

0.4) Choice of Components to Checkpoint

The components which we need to checkpoint are those which we can replace or reallocate in order to overcome faults. We must ensure that the set of components which we can replace or reallocate is such that the desired fault coverage is achieved. Obviously our decomposition must be applied hierarchically, going down to the lowest level replaceable or reallocatable module, but initially we will only consider the highest levels of the hierarchy.

At the highest level the hardware decomposes into the major independent subsystems between which work can be redistributed, and which are largely independent from the point of view of faults. In a homogeneous multicomputer system these subsystems would be the individual computers.

The software decomposes into units which can be independently loaded and run, such as the processes in a POSER system.

The hierarchic nature of systems is considered in more detail in the next section as it significantly affects error recovery algorithms.

0.5) Hierarchic checkpoints and recovery

Our checkpointing and error recovery strategies must take account of the hierarchic structure of systems. If an error cannot be corrected at one level then the problem is passed onto the parent component which will try to deal with it, and so on. Thus the checkpoint for a particular component cannot have any use outside the scope defined by its parent component. We can thus see that in general we can, and indeed should, store checkpoint information for one component so that it is only accessible through its parent.

This strategy is satisfactory at all but the highest level - i.e. that of the system itself. Here the parent component is the environment, but it was one of our basic requirements that the system should perform error recovery without outside intervention [1], thus our general rule breaks down. The practical solution to this problem is to keep multiple copies of checkpoints within the system, and rely on the low probability of common mode or simultaneous faults affecting the separate copies to give us good error coverage.

This analysis also throws some light on how we should design our recovery mechanisms. We say that the system has a single fault, and single error at level N, if all faults and errors at lower levels are included in one component at level N. We can always find such a level, even if the whole system has to be regarded as faulty (see fig. 0.1). It is generally much easier to perform recovery if we have to deal with only one faulty component, rather than many. We combine recovery mechanisms based on the assumption that only one component (at

that level) is faulty, with the hierarchical strategy described above. It is likely that the error recovery mechanism will fail at levels where there are multiple errors, but we can expect that recovery will be successful where only one component is faulty. Simplicity is a primary aim in designing fault tolerance mechanisms, thus the use of a hierarchical, single fault, single error, recovery strategy seems desirable.

Unfortunately this strategy will not be adequate at the system level. In order to increase the fault coverage we would like to produce recovery mechanisms at this level which are capable of handling multiple errors and possibly multiple faults. This makes the highest level recovery algorithms more complicated than those at lower levels.

In the software which comprises the user processes, the hierarchical structure is applicable and of use. We can regard each block in the program (assuming a block - structured HLL is used) as being one level in the hierarchy. Thus the recovery information for one block in the language need have no meaning outside the level immediately above (more strictly, the next one for which error handling mechansims are defined). We see therefore that the recovery information for one block need only be accessible through the block above, which defines how this information should be stored. Similarly we can see that error recovery should be performed on a block by block basis, and that the aim of the error recovery mechanisms should be to complete the action of the block correctly. This recovery can be performed in many ways, and the means of achieving it does not concern us here. It is worth noting, however, that this type of recovery mechansim is widespread: The Recovery Block technique [8] developed at Newcastle University implements exactly this form of hierarchical recovery; The exception handling mechanisms provided in the FLEX instruction set will support this type of recovery; The exception handling mechanisms provided in ADA are also of this hierarchical nature.

0.6) Checkpoint Structures

The checkpoints for a component in a system comprises a copy of the component state (i.e. sufficient information to characterise the condition of the component at that time) together with information defining the causal relations with other components, and a unique identifier for the checkpoint. The checkpoint structure is best illustrated by considering an example. We describe here the checkpoints for POSER processes and channels.

POSER processes and channels are defined so that a process or channel activation corresponds to a procedure call, execution and return. The interactions with other processes and channels are by means of the parameters and results of the procedure which are propagated through the system by a message passing method. Because of the way these processes interact a checkpoint is required immediately prior to each activation.

There is long lived data associated with each process and channel which is preserved, unchanged, from one activation to the next.

The checkpoints for each process and channel conceptually comprise:

A copy of the long - lived data associated with the process or channel (known as the Own Data).
A copy of all parameters.
Sequence number.
Dependency information.

The first two items are effectively the process or channel state. The remaining information defines the causal dependencies between this and the other components in the system. It is convenient to show these dependencies diagrammatically in order to describe the checkpoint structures. Often the state information and that concerning the actions on the states (the process and channel activations) are shown independently, thus we obtain graphs corresponding to individual activations as shown in figures 0.2 and 0.3.8 These are elemental Occurrence Graphs as we shall see in the next section.

1) Models of Error Recovery

1.0) Definition of Occurrence Graphs

Occurrence Graphs (O.G.s) were developed to model the dynamic behaviour of systems, and are described for systems in which it is necessary to perform error recovery in [9]. O.G.s are "generated" as the system executes and they bear a strong relationship to Petri Nets [10] which can be used to model the static structure of systems. We are interested in Occurrence Graphs as they can be used to model the checkpoints which are taken as the system executes, and the manipulation of checkpoints during error recovery. We will initially describe O.Gs in the abstract, then show how they map on to the checkpoint structures described above.

Occurrence Graphs, as we use them, contain two node types:

Places - these represent the state of a system component at a particular time.
Events - these represent the activities which occur in the system (such as process activations) which cause changes in component states.

Diagrammatically places are represented by circles and events by vertical lines. The arcs of the graph are directed and represent mechanistic causality (at this level of abstraction), hence O.G.s are acyclic and are often referred to as DAGs - Directed Acyclic Graphs. The arcs are represented diagrammatically by curved lines bearing arrows pointing from the causal node to the caused node. We can now see that the graphs in figures 0.2 and 0.3 are the O.G.s which are generated by a process activation and by a channel activation respectively. Note that one event and a number of places correspond to one checkpoint. The checkpoints are the information which we store, and the O.G. is simply a model of these checkpoints and their causal relationships. However we will tend to use both these terms to describe the stored checkpoints, choosing the term which emphasises the attribute in which we are interested at any given time.

Places can only have one incoming arc and one outgoing arc, as only one event can generate a place (can have caused the corresponding state) and only one (different) event can be (partially) caused by it. Events can have multiple incoming and multiple outgoing arcs (the event has been caused by the conjunction of a number of states and causes changes to a number of states). The places hold enough information to allow the event which they have caused to be repeated - e.g. to allow POSER processes and channels to be restarted.

Merlin and Randell [9] introduce additional graph elements and markings which make their model capable of describing systems where not all the component states are checkpointed (at the level in the system with which we are concerned) and give a general treatment of error recovery in distributed

systems. By considering only systems where every state is checkpointed, such as is being done with the POSER implementation on FLEX, we are able to employ a simpler model, and to produce some more useful results. It is possible with this simple model to extend Merlin and Randells analysis to show the efficacy of our recovery algorithms.

1.1) Error Recovery - the Chase Protocols

Merlin and Randell [9] define a set of protocols which they term the "Chase Protocols", and prove that these guarantee error recovery, in the following sense. The protocols construct an O.G. which corresponds to a consistent system state, from a graph which corresponds to a system state which is inconsistent after an error or some errors have occurred. The set of states defined by this new Occurrence Graph may be used to restart the corresponding components thereby providing error recovery.

The set of graph elements used in our model is a strict subset of that used by Merlin and Randell (with some relabelling) thus, because of the nature of the protcols and the proof, their proof holds for our formalism also. However the general model yields little information about the amount of work which will be lost in error recovery, or even whether or not there exists a non - trivial set of places to which we can recover (known as a recovery line). Our simpler model gives more information in both these areas, hence it is worth pursuing the analysis here.

We must first introduce some simple notions of graph marking and define our simplified Chase Protocol. When places and events are generated (checkpoints are taken) they are believed to be correct and are marked "live" and "active". When a place has been consumed (has generated an event), or when an event is complete (it has generated all the consequent places) the appropriate graph element is marked "inactive". If a place or event is believed to be errorneous, or it is to be removed from the graph for some other reason (i.e. a checkpoint is discarded), it is marked "dead". Dead graph elements cannot become live, and are ignored. Live, inactive, places may be returned to the active state to cause a "new" event if the one which they originally caused becomes dead - e.g. the state contained in a process checkpoint may be used to restart the appropriate process from a time before it was believed to be faulty.

When a component is thought to be faulty the graph elements which are believed to be erroneous (because they were generated after the fault occurred) are marked invalid, then will become dead. In practice we will only mark events invalid; if a place is thought to be incorrect, then we must repeat the event which generated it, hence we directly mark the causal event invalid.

The Chase Protocols work by sending messages between those parts of the system responsible for maintaining the O.G. (e.g.

part of the kernel in a recoverable virtual machine). For the sake of simplicity of expression the protocols are described in somewhat loose terms, for example "messages sent down tne incoming arcs to an event" is used for "the kernel responsible for maintaining the event sends messages to those kernels which are responsible for the places which are immediate causes of the event" and so on. It is hoped that this lack of pedantry clarifies, rather than obscures, the definition.

The algorithm may be stated as follows:

```
IF an event is marked invalid
    THEN
        the event sends fail messages on all its outgoing
        arcs;
        the event sends fail messages down its incoming
        arcs;
        the event is made dead
FI;

IF a live place receives a fail message on an outgoing arc
    THEN
        the place is made active
FI;

IF a live place receives a fail message on an incoming arc

    THEN
        the place sends a fail message on its outgoing
        arc;
        the place is made dead
FI;

IF a live event receives a fail message on an incoming arc

    THEN
        the event sends fail messages on all its outgoing
        arcs;
        the event sends fail messages down its incoming
        arcs (except that which sent the fail message);
        the event becomes dead
FI;
```

The incoming arcs to dead elements are effectively removed so it is unnecessary to define the algorithm for a dead graph element receiving a fail message.

We wish to know that this algorithm does yield a recovery line and that it does ensure that the requirements of section 0.1 are met. We approach this by attempting to find what work will have to be "repeated" after executing this algorithm. The work of the components which lead to the generation of the now deleted graph elements has effectively been discarded. This work will not necessarily be repeated, as it may have taken place erroneously anyway (in a sense, the control flow in the system was wrong). Thus we see that the concept of what work is "repeated" is rather ill - founded. The best we can do is

define what work we did without deriving any benefit from it, i.e. that corresponding to the deleted portion of the graph, which we term the work lost.

It is obvious from the definition of the algorithm that the work lost is defined by the set:

$$S(e) = \bigcup_{i=0} S_i(e)$$

where: e is the invalid element, $S_0(e) = \{e\}$, and $S_i(e)$ is the set of the elements of the O.G. having an incoming arc in the O.G from some element of $S_{i-1}(e)$. This set represents the work which e directly or indirectly caused i.e. that for which it is a partial cause, which is what we would intuitively expect. (S(e) - e is often described as the "causal future" of e). This set is a strict subset of that derived by Merlin and Randell. We may now define the recovery line as:

$$R(e) = \bigcup_{i=0} R_i(e)$$

where $R_i(e)$ is the set of places having ingoing arcs into $S_{i-1}(e)$. This means that we may restart the system by restarting the component associated with the invalidated event using the places which had arcs into that event in the original O.G. (the other components will eventually be restarted if appropriate as e was their partial cause).

Merlin and Randell demonstrate other properties of O.G.s and the Chase Protocol, particulary concerning termination of the algorithm and the operation of the algorithm in the presence of multiple invalid elements. As indicated earlier the proofs for the general O.G. hold for our formalism (as it is simply a special case). The proofs of the remaining properties do not yield any further information if they are performed for our formalism so they will not be pursued further. We do however quote, in a somewhat imprecise form, two results which are important:

> Termination of the algorithm - the nature of the algorithm is such that, even if the parts of the system which are not affected by the error are still running normally, the set S(e) will eventually be complete (no longer growing) provided that the fail messages propagate faster than new graph elements are generated. Thus the algorithm will effectively terminate and allow us to recover from the effects of the error.
>
> Multiple invalid elements (implying multiple errors, and perhaps faults) - in the presence of multiple invalid elements a composite recovery line will be established, and the work lost is the union of the sets corresponding to each of the elements when considered separately. All the other properties are still preserved in the presence

of multiple errors.

Returning to our requirements in section 0.1 we see that the amount of work lost (requirement one) is the theoretical minimum and the set of states from which we are recovering is consistent from the point of view of causality. Since the work lost is only that for which the invalid element was a partial cause, we can restart unconnected parts of the system independently i.e. requirement three is met.

It is likely that in many cases less information could be stored than this graph structure and algorithm requires - i.e. we can afford to lose more than the theoretical minimum amount of work without the system behaving in a manner which is unacceptable. However if we interpret the second requirement as having to be absolutely constrained by the first, then we can use the above analysis to define what constitutes the "minimum amount of information". It should be noted that all the proofs above rest implicitly on the assumption that the graph represents all causal dependencies. Thus we can see that it is essential to produce checkpoints (generate places) on the reception of all messages at all processes otherwise the graph structure will violate this constraint. This demonstrates the necessity of the checkpointing rule described in section 0.3 and the proof of the prototcol shows that it is optimal in terms of the work lost.

1.2) Restart Protocols

The Chase Protocols as described above give an elegant, theoretical model of error recovery mechanisms in a distributed system. However certain implicit assumptions contained in the protocols make them difficult to implement directly. We now extend the Chase Protocls to produce what we call the Restart Protocols which are intended to be more readily implementable. We also show that the manipulations which the Restart Protocols perform on the occurrence graph are equivalent to those of the Chase Protocols.

One limitation of the Chase Protocols is that they assume that the O.G. is built correctly, which we cannot guarantee if there are faults in the system. In particular different components in the system may have a different view of the system state, either simply due to message delays, or, for example, because an acknowledgement to a message gets lost. The degree of discrepancy can be bounded (to a high degree of confidence) if we check that the O.G. is built consistent with the static connectivity of the system, that is with the static definition of the components and their interconnections. The Restart Protocols are defined to operate with a graph built in this manner.

We now define the "Restart Protocols" which will lead to the desired recovery with an O.G. built as described above:

IF an event is to be repeated (the system component restarted)

```
    THEN
        fail messages are sent down all potential outgoing
        arcs (i.e.  we use the static, not dynamic,
        connectivity);
        the event sends messages down all potential
        incoming arcs;
        the event is made dead
FI;

IF a live place receives a message from an event which it
        caused
    THEN
        the place is made active
FI;

IF a live place receives a fail message from an event
        which caused it
    THEN
        the place sends a fail message down its potential
        outgoing arcs;
        the place becomes dead
FI;

IF a live event receives a fail message from a place which
        caused it
    THEN
        the event sends fail messages down all its
        potential outgoing arcs;
        the event sends messages down all potential
        incoming arcs (except that which sent the fail
        message);
        the event becomes dead
FI;
```

This protocol obviously performs the equivalent graph manipulations to the Chase Protocols since the dynamic (actual) connectivity must be less than or equal to the static (potential) connectivity, and it copes with all possible inconsistencies in the graph (assuming that the checks in graph construction prevent any structures which are impossible given the static connectivity).

These protocols are obviously effective regardless of the scale of the component which fails i.e. they work if simply one process is restarted, if a whole processor is restarted or even if the whole system is restarted after a major failure. Since the Chase Protocols are equivalent to the Restart Protocols in terms of their graph manipulations, their properties also apply to the Restart Protocols.

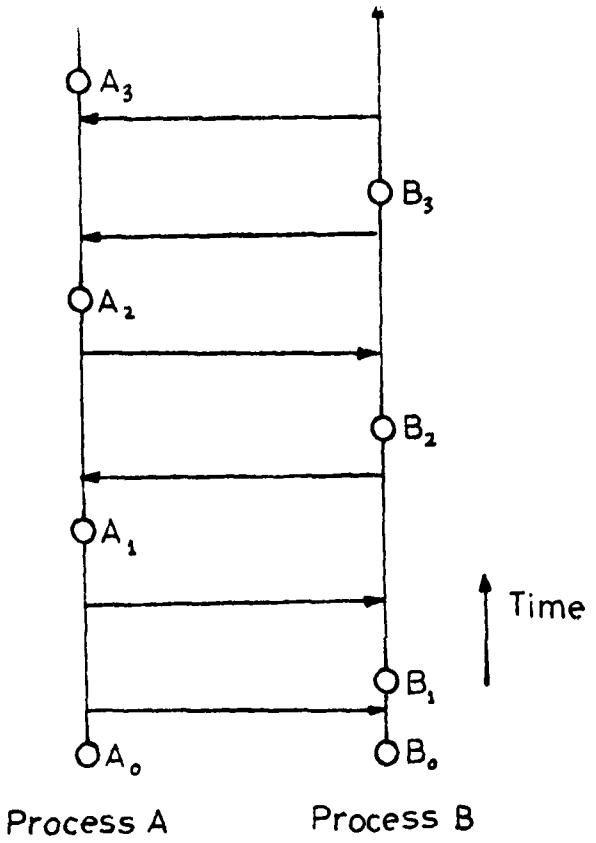
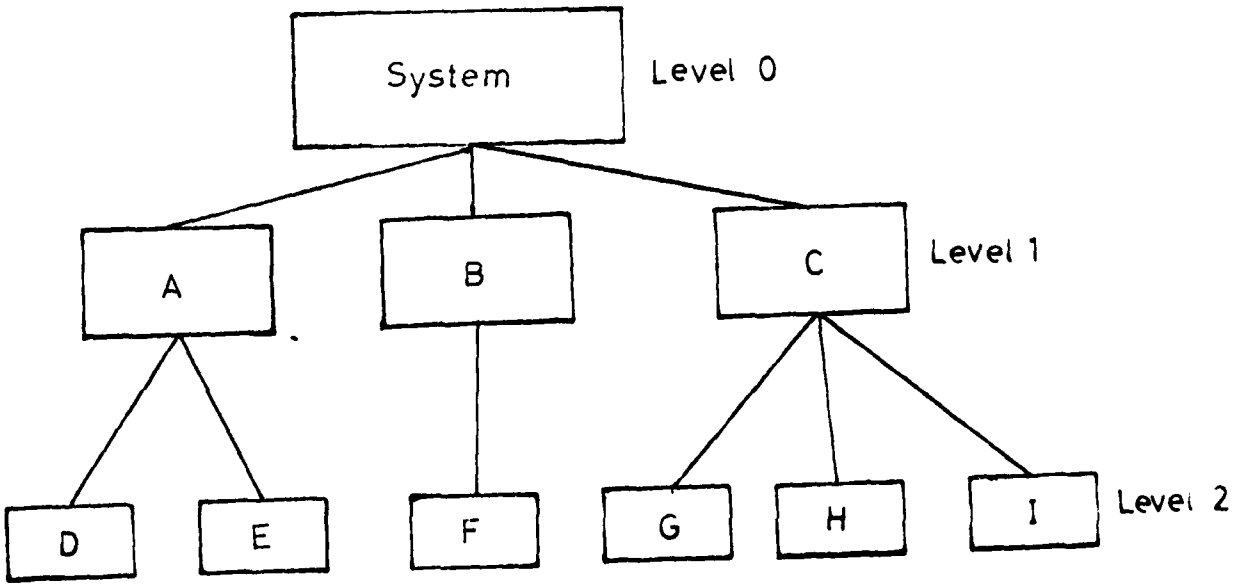


Fig. 0.0 Checkpoint Consistency.

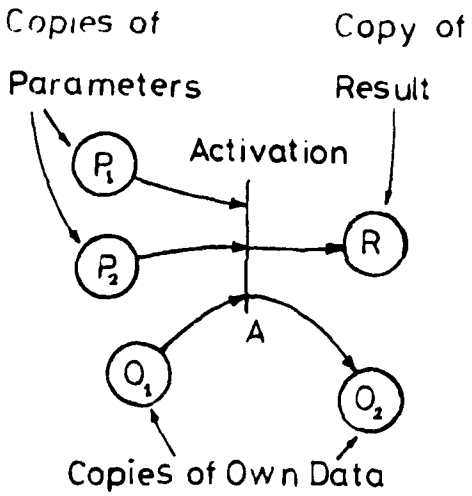


D&E are components of A.

A fault in G with errors in G&I gives a single fault & error in C.

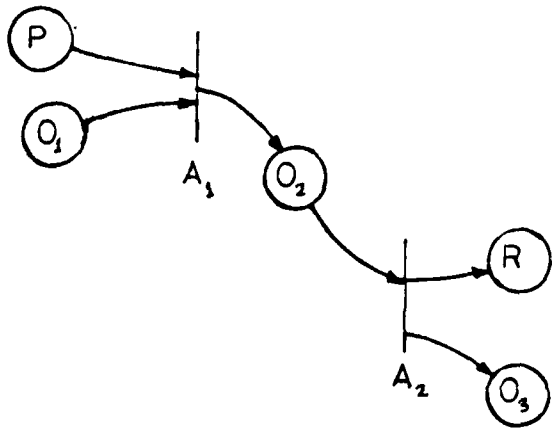
Faults and errors in D&H give a single fault & error in the system.

Fig. 0.1 Fault Hierarchies.



P_1, P_2, O_1 & A form the checkpoint for one process activation.

Fig. 0.2 Process Checkpoint.



P, O_1 & A_1 form the checkpoint for an incoming message, and O_2 & A_2 that for an outgoing one.

Fig. 0.3 Channel Checkpoint.

2) The Generation and Maintenance of Checkpoints

2.0) Introduction

The above analysis derived several properties of O.G.s which we should also like our checkpoint structures to have. We describe here techniques for generating checkpoint structures and maintaining them so that they do have these properties. We first consider simplifications which can be made to the graph without altering its properties, then we describe protocols for reliably producing checkpoints and for deleting checkpoints when they are of no further use. Finally we consider where and how checkpoints should be stored.

Most of the mechanisms described here are generally applicable, but they are presented in POSER terms as POSER represents a convenient basis for description. The optimisations described are specific to graphs generated by the execution of a system composed of POSER processes and channels.

2.1) Graph Optimisations

O.G.s which represent the execution of a system of POSER processes and channels will have some structure, by virtue of the form of the processes and channels. We can make use of this structure to simplify the O.G. and hence to reduce the number and complexity of checkpoints which we have to store. We perform these optimisations statically, so that we are able to generate the optimised graph directly (rather than having to simplify the full graph as we are building it).

The O.G. corresponding to a single process activation is shown in figure 0.2. Because there are always multiple in-paths to, and multiple out-paths from, an event it is not possible to reduce (represent in a simpler fashion) this graph. In practice we can optimise this structure by collapsing the event onto the place for the Own Data, and treating this pair as one object. This artifice reduces transput overheads in generating checkpoints, and speeds up some of the graph manipulations.

A graph showing some typical channel activations is shown in figure 2.0. Let us consider the data associated with message 1. Since the channel does not transform the data, it is evident that the data in this message is stored many times: in places Pk, Rk+2, Ok+1 and Ok+2 (as Ok+1 and Ok+2 contain copies of the long lived data in the channel, which must include the messages). We can obviously reduce the amount of copying done by replacing the data part of place Rk+2 with a reference to that in Pk. With a little more ingenuity we can further reduce the number of copies by modifying Ok+1 and Ok+2 or Pk. This reduces the number of copies of the message to one, but still leaves us with some undesirable causal relationships.

Since the message content is not modified on passage

through the channel, nor does the passage of the message [illegible] in the channel (except, perhaps, to [illegible] the recording of the channel states shows causal dependencies which are unimportant. That these dependencies are also undesirable is shown by considering the regeneration of messages after an error. If message one has to be regenerated then message two will also be regenerated, although these messages are strictly independent. Therefore we would like to remove these causal dependencies, but this would mean that the internal channel state is not recorded.

However if we can constrain the channel operation suitably (e.g. to FIFO) we can simply reconstruct the channel state by re - inputting the messages. This will not in general yield the same channel state as had existed before the error, but will produce one which is functionally equivalent - i.e. we are applying forward error recovery techniques. Thus we can reduce the graph for the channel to one place per message unless the desired mode of channel operation would make it unacceptably time consuming to reconstruct the channel state during error recovery.

The place representing the message in the channel contains the same information as the places representing parameters to the processes, so we may further optimise the graph by amalgamating the appropriate places. This can most readily be accomplished by abolishing the places associated with the process activation, and making the event refer directly to the places for the channels.

For the purpose of this discussion we assume that these graph optimisations have been made.

2.2) Checkpoint Generation

The primary concern in generating checkpoints is to ensure that they are produced consistently. It is obviously impossible to ensure absolute consistency when we are dealing with distributed checkpointing in a system where errors may occur (e.g. messages may go astray). What we are aiming to do is to reduce to an acceptable minimum the degree of inconsistency which can occur. We thus define protocols for the production of checkpoints which use handshaking techniques to improve the probability of error detection.

The protocol executed for the channel is:

```
C input C
IF message received
    THEN
        generate live, active checkpoint;
        C a single place C
        acknowledge message;
        Make channel eligible for scheduling
FI;

C output C
```

```
IF the channel has been run without an exception being
        raised
    THEN
        deliver message;
        wait for acknowledgement;
        complete checkpoint for that message (make
        inactive)
FI;
```

These threads can not be run simultaneously, but there is no restriction on the order in which they are run. These threads effectively form a "shell" in which the channel runs. The most likely inconsistencies (between different parts of the graph) which can occur with this protocol are due to an acknowledgement for a message going astray. This will result in different parts of the O.G. giving incompatible histories for the parts of the system which they represent. Clearly this protocol must rely on a lower level set of protocols (for ensuring reliable data transmission) to reduce the probability of this occurring. It should be noted however that the Restart Protocol is capable of ensuring that a consistent set of checkpoints is reconstructed from a graph which is constructed in this manner (see section 1.2).

The protocol for processes is similar, but it has to deal with the multiple in and out paths which are associated with the process. When the system is initialised a skeletal checkpoint consisting of a place for the intialisation Own Data, and an empty data structure for holding the event and its links to the casusal places is created. When messages arrive these links are filled in until the process is schedulable. A similar process occurs after the process has run building up the links to the caused places. The protocol is:

```
C input C
WHILE NOT eligible for scheduling
    DO
        IF message received
            THEN
                add link to causal channel checkpoint;
                acknowledge message
        FI
    OD;

C output C
IF process run without exception being raised
    THEN
        create skeletal (live and active) checkpoint for
        next activation
        WHILE messages to send
            DO
                deliver message;
                await acknowledgement;
                add link to caused channel checkpoint
            OD;
        complete checkpoint (make inactive)
```

```
ELSE
    perform error recovery
FI;
```

These threads will be run alternately unless the process does not terminate normally, but there is scope for speeding up the operation by use of parallel processing. As for the channel this protocol is essentially the shell in which the process runs. It should be clear from considering the sequence of operations concerned in creating a checkpoint how the protocols for the channel and process interact, and that they do produce checkpoint structures which are equivalent to the O.Gs described above.

2.3) Maintenance of Checkpoints

2.3.0) Principles

If nothing were done to prevent it, O.G.s would continue to grow indefinitely (assuming that the system continued to run). Thus one primary aim of checkpoint maintenance is to ensure that the set of checkpoints stored in the system does not grow too large (for the storage space available for them). This can, in principle, be achieved by destroying checkpoints (removing places and events from the O.G.) when they are no longer required - the difficulty is deciding when they are no longer required. The problem is twofold. First, no matter how cautious one is about removing checkpoints, one can always synthesize an error which is sufficiently devious, or latent, that one of the deleted checkpoints was essential for recovery. Hence we need to establish a criterion for removing checkpoints which will be "reasonably good", although we know that it is not failsafe. In section 2.3.1 we consider how the criterion for being deleted varies with assumptions about error latency. Second, it is difficult to devise algorithms for removing checkpoints which are amenable to distributed execution (although it is relatively easy to do so for centralised execution).

Directly removing checkpoints is not the only ploy for preventing their indefinite growth. An attractive alternative is to collapse the checkpoints into "larger" ones, and to modify their connectivity so that their causal relationships are preserved. The checkpoints so created correspond to larger (conceptual) components than the processes and channels. The collapsing will, in general, allow some checkpoints to be removed. This technique can be applied repeatedly, thereby continually thinning the graph. This hierarchical structuring of the graph corresponds quite closely to the idea of "boxes" introduced in [11]. This method alone, or a combination of the two methods, would yield fine error recovery from small and recent faults, and more gross recovery from larger and more highly latent faults. With this method the difficulty arises in finding an algorithm for collapsing the checkpoints in a causally consistent way, which is suitable for distributed execution.

The second primary aim of checkpoint maintenance is to preserve those checkpoints which are still required. This aspect of maintenance involves ensuring that the multiple copies of the checkpoints are built consistently (with each other), and recovering from the corruption of, or loss of, one of the copies of the checkpoints.

The fundamental mechanism for ensuring consistent building of checkpoints is to serialise the critical operations (e.g. irrevocably changing pointers) so that, at any one time, at least two of the three copies agree. This ensures that recovery from a fault, occurring at any time, is straightforward.

The recovery from corruption to, or loss of, a copy of the checkpoints involves building a new copy of the remaining, intact sets of checkpoints. Since the sets of checkpoints are always growing, care must be taken to see that the recovery terminates, and that the new copy does become properly integrated with the old ones. This could, most simply, be achieved by stopping the appropiate part of the application system until the new copy was made. This will not normally be acceptable due to the cessation in service it would cause. A more practical algorithm is to build a copy until it is "nearly complete", then to integrate completion with one of the critical operations. This solution would lead to a relatively short break in service.

In practice, if the number of checkpoints kept for any one process is small, acceptable recovery times may be achieved simply by creating a new set of checkpoints (from scratch) - this set will very soon be in step with the other two. This algorithm is by far the most desirable (being the simplest) if it is acceptable.

2.3.1) Deletion of Checkpoints

We will now consider the deletion of checkpoints in more detail. We start by making some unrealistically restrictive assumptions about error latency, then proceed to derive algorithms which are satisfactory under more reasonable restrictions.

Under the assumptions:

1) The O.G. is constructed consistently - i.e. we can guarantee that no messages get lost.
2) All faults are detected (by means of the errors which they generate) within the process (channel) activation in which they occurred.

we can delete checkpoints as soon as they become inactive (strictly this marking is no longer necessary). This is possible because we know that, when we make a checkpoint inactive, the event associated with that checkpoint, and all the causal events, have been completed correctly. Thus we cannot wish to repeat the event, hence we no longer require

the checkpoint. With these assumptions the maintenance of the checkpoints becomes essentially trivial.

If one simply removes either of these two constraints then the potential error latency ceases to be bounded, hence we need to relax these assumptions in a more subtle way. We can not realistically assume that errors will be detected within the process activation in which they arose, but it is quite reasonable to assume that a large proportion of the errors will be detected by processes which receive the outputs of the erroneous process, either by input validity checks, or by the fact that they fail in execution. Thus we can be reasonably sure that a process has been executed correctly, (and hence we no longer need to keep its checkpoints), if all the processes which use its outputs have run successfully.

Obviously this algorithm is not fail safe, but it is practicable. The protocol can be extended to cater for more highly latent errors by increasing the length of the chain of processes (N) which have to be executed before we decide that the causal process has been run correctly. It is difficult to know how the error coverage varies with N, but the cost of executing the protocol will increase more than linearly with N. Intuitively the best approach is to keep N small (or even unity) and to back up the restart protocol with a cold restart mechanism.

For N = 1 the protocol is:

```
C for a process C
IF the process is executed without error
    THEN
        send thank you messages down all incoming arcs to
        the corresponding event
    ELSE
        perform error recovery
FI;

REPEAT
        receive thank you message
    UNTIL
        message received from all outgoing arcs
TAEPER; C closing delimiter for REPEAT C

delete process checkpoint and corresponding channel
        checkpoints;

C for a channel C
pass thank you messages
```

This protocol may be extended for N > 1, by including N in the message, decrementing it on passing "through" each process, and only acting on the message when N is reduced to 1. No further modifications are required. This protocol implicitly relies on the ability to perform error recovery to correct inconsistencies in the O.G. without re - executing processes.

2.3.2) Collapsing Graphs

The graph collapsing (or reductions) which we have to perform are identical (in purely graph theoretic terms) to those used in program analysis [12]. These algorithms are normally performed in a centralised way, but could probably be adapted for distributed execution. It is intended that these algorithms will be studied as they could yield very efficient and elegant implementations of the checkpointing protocols. However this problem has not yet been tackled, so it will not be pursued further here.

2.4) Checkpoint Storage

The arguments and analysis developed above have given us enough information to allow us to decide where the different classes of checkpoint should be stored. We assume here that processes and channels will be written in a block structured HLL, and that a dynamic storage allocation mechanism (a "heap") is available.

Checkpoints corresponding to program blocks within the user processes do not need to have any meaning outside the next superior block for which an exception handler is supplied. It is intended that processes will be restarted from the beginning of an activation in the event of errors which effect a complete computer, and which therefore require processes to be reloaded on another machine in the process of error recovery. Thus these checkpoints need not be stored outside the machine on which the process (or channel) is running. The checkpoints may therefore be stored on the heap with references to them in the appropriate stack frame. The scope rules are such that the checkpoints will be put to garbage at the appropriate time. Obviously other implementations which are functionally equivalent maybe devised. This destruction of checkpoints is equivalent to a very simple reduction of the O.G. - simply collapsing the lowest level of detail into one node at the level above. The mechanisms as described here correspond quite closely to those used in early implementations of Recovery Blocks; - a later implementation makes use of hardware assistance [12], [13].

The process and channel level checkpoints must be accessible from machines other than those on which the process (or channel) was originally running, in order that their work may be continued elsewhere if necessary. This information is the highest level recovery information in the system and must be replicated as explained above. It is essential therefore, that multiple copies of this information be stored on globally (assuming a homogeneous system) accessible, semi - permanent media. One way of doing this is to use three (or more) physically separate magnetic discs, each containing a complete set of checkpoints. The discs must be accessible from any computer in the system. In FLEX this would be achieved by having three separable filestores on separate discs. COMFLEX would ensure the global accessibility of the checkpoints. The

roots for these checkpoints would have to be held on disc in known, named, locations so that they could be found even after a major crash. The "critical operations" alluded to above would primarily be manipulations to these roots. A crash whilst they were being changed could lead to (one copy of the) checkpoints being irrecoverably lost, hence the need for the serialisation of critical operations.

2.5) Practical Implementation of the Protocols

The above descriptions have not defined (in much detail) the interactions between the protocols, and other problems of their implementation. It is believed that these protocols can be succesfully integrated to form an "error recovery kernel" (for want of a better term) for use in a distributed computer system. Most of the protocols can be integrated simply by putting them in a hierarchy inside a shell in which the process or channel runs. The protocols for generating and removing checkpoints are, however, strongly bound up with the simple data transport mechanisms and require careful consideration. Implicit here also are the "timeouts" necessary to detect that an expected message has not arrived which have to integrated into the error handling scheme.

These problems of implementation are being tackled by building an experimental system on a multi - computer simulation running on RSREs prototype FLEX system. If this experiment proves successful (within the capability of a simulation to validate these protocols) then the experiment will be extended to a more practical implementation. At this stage it should be possible to see the difficulties introduced by true (rather than simulated) parallelism, and the handling of real peripherals.

3) Conclusions

This report has described a number of protocols which are useful in the provision of error recovery mechanisms in distributed systems. The protocols are intended to be fairly readily implementable, although it is unlikely that they could be implemented solely from the information presented here due to the lack of detail. Also because of the way in which the protocols interact their form, as implemented, differs significantly from the abstract form shown here.

The protocols as described do not cater for certain eventualities, such as the creation and deletion of processes. The theoretical foundation, based on the Occurrence Graphs, is rich enough to describe systems which allow processes to be created and destroyed, and there seems to be no theoretical difficulty in extending the protocols to deal with this. Such extensions would, however, require some care in implementation if problems of resource allocation etc. are to be avoided. The protocols are, however, quite adequate for essentailly static systems such as those built using MASCOT or POSER.

These protocols are not sufficient in themselves to allow a fault tolerant computer system to be produced. In particular mechanisms for fault and error detection, and for the reallocation of resources (fault recovery) must be provided. Little has been said about these mechanisms as they depend very heavily on the hardware architecture and on the application system.

A recovery kernel must be built on an almost "bare" machine if it is to be really effective, which poses certain problems. It is dubious that an efficient operating system could readily be built within the restrictive POSER discipline. Thus we must either run application programs fairly directly on the kernel and tolerate the limitations which this imposes, or do a considerable amount of work to build an operating system which incorporates fairly explicit recovery mechanisms.

The protocols for generation and maintenance of the checkpoints have been implemented on the simulation, and the recovery mechanisms will shortly be added. Some effort will then be spent in evaluating these mechanisms with a "typical" apllication system. Further reports will present the results of the evaluation.

4) References

[1] "Fault Tolerant Computing", J A McDermid, RSRE Memo. 3197, 1979.

[2] "An Intrduction to the FLEX Computer System", J M Foster, I F Currie, C I Moir, J A McDermid, P W Edwards, J D Morison, C H Pygott, RSRE Report 79016, 1979.

[3] "COMFLEX - a high speed packet switch for inter - computer communicarion", J A McDermid, Proc. of Eurocomp 78, 1978.

[4] "POSER - a Process Organisation to Simplify Error Recovery", J A McDermid, RSRE Memo. 3249, 1980.

[5] "MASCOT - A Modular Approach to Software Constrcution Operation and Test", RRE Tech. Note 778, H R Simpson, K Jackson, 1975.

[6] "Reliable Computing Systems", B Randell, P C Lee, P A Treleaven, in Lecture Notes in Computer Science No. 60: Operating Systems - An Advanced Course, G Goos, J Hartmanis Eds., Springer Verlag, 1978.

[7] "Software Reliability: The role of programmed exception handling", M P Melliar -Smith, B Randell, Proc. ACM Conf. on Language Design for reliable software, 1977.

[8] "Recovery Blocks in action: a system supporting high reliability", T Anderson, P Kerr, Proc. Int. Conf. on Software Engineering, 1976.

[9] "Consistent State Restoration in Distributed Systems", P M Merlin, B Randell, Report No. TR113, Computer Lboratory, University of Newcastle upon Tyne, 1977.

[10] "General Net Theory", C A Petri, in Computing System Design, University of Newcastle upon Tyne, 1977.

[11] "A Formal Model of Atomicity in Asynchronous Systems", E Best, B Randell, Report No. TR 130, Computer Laboratory, University of Newcastle upon Tyne, 1978.

[12] "Graph Theory leads to Program Visibility", B D Bramson, S J Goodenough, RSRE Report 80004 (to appear).

[13] "Concurrent Pascal with backward error recovery: Inplementation", S K Shrivastava, Software Practice and Experience Vol. 9, No. 12, 1979.

[14] "A recovery cache for the PDP11", P A Lee, N Ghani, K Heron, Report No. TR134, Computer Lboratory, University of Newcastle upon Tyne, 1979.

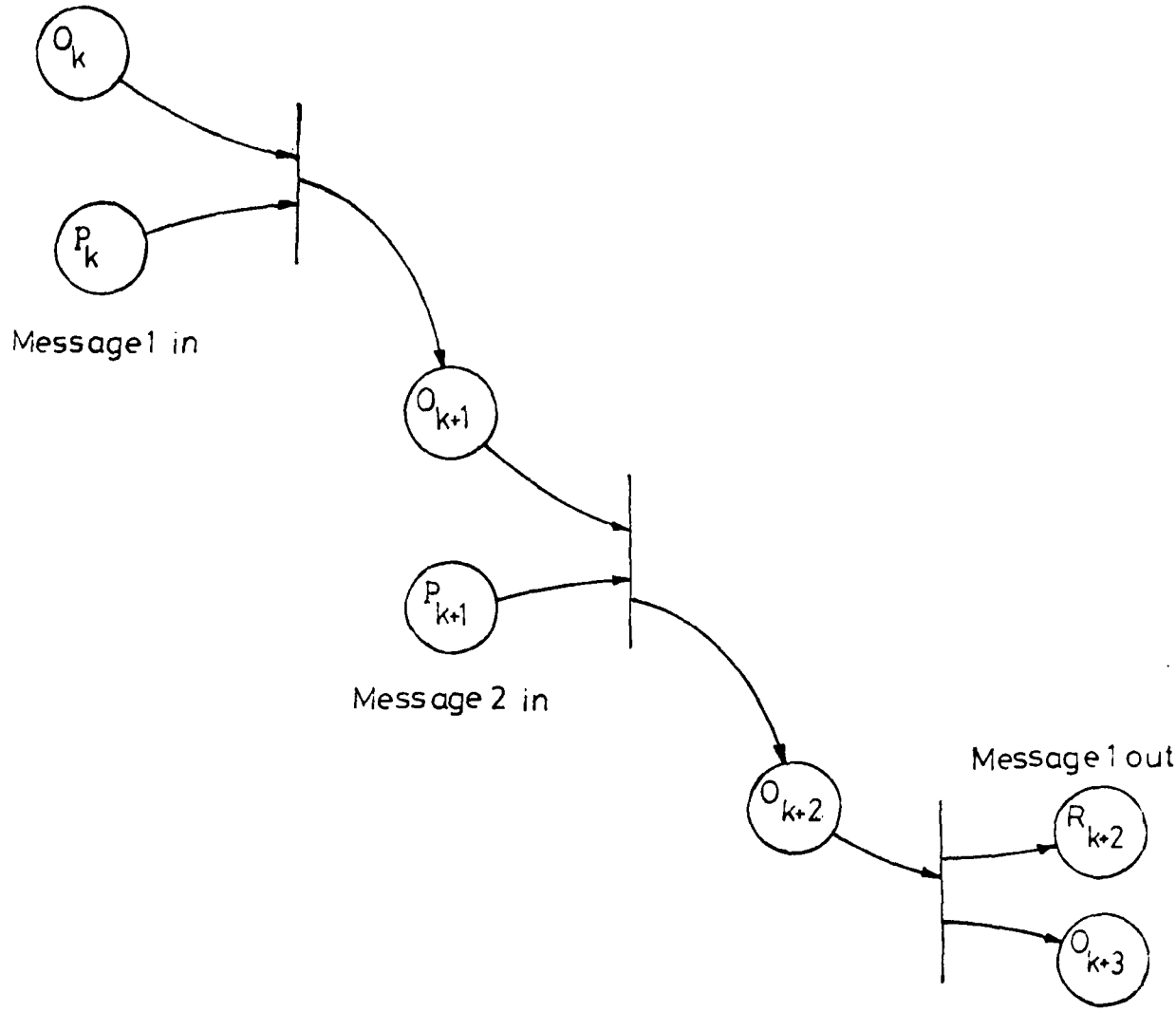


Fig. 2.0 Typical Channel Activations.