Distributed predicate detection in a faulty environment

J. Roger Mitchell and Vijay K. Garg

TR-PDS-1997-001 **April 1997**

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Distributed predicate detection in a faulty environment

J. Roger Mitchell* Vijay K. Garg†

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Department of Electrical and Computer Engineering
The University of Texas at Austin,
Austin, TX 78712-1084

April 30, 1997

Abstract

There has been very little research in predicate detection for distributed systems in a faulty environment. In this paper we create an algorithm for detecting conjunctive process and channel predicates which is able to operate in the presence of process crash, channel (or link) severance, and message loss failures. Our approach is truly distributed, a requirement for the algorithm to remain live under these classes of failures. We also do not use control messages unlike previous work.

1 Introduction

Detecting a global predicate is a fundamental problem in distributed systems. However, very little research has been done for detecting these predicates in a faulty environment.

An algorithm which can provide predicate detection amid failures can be useful for monitoring applications. An example of predicates worth monitoring for are "bad states," that is, states we want to make sure the system doesn't fall into. Some examples of such predicates are: no leader, no token, or all processes and channels inactive. Consider an application which requires that a token be passed among a set of processes. A process, channel, or message failure can cause the token to be lost. When loss occurs it is necessary to regenerate the token.

In this paper we present an algorithm which can detect such predicates in spite of process crashes, channel severances, and message losses. We assume an asynchronous environment and so, accurate knowledge of failures is not possible [6]. Therefore, we use inputs which are suspicions of failures. The user supplies these suspicions as input to the algorithm using a method such as [17], and makes the suspicions as accurate as necessary. The use of suspicions does not invalidate our approach. For example, when detecting a bad state, we can tell if the predicate *could* be true. In any case, if failure has occurred, we guarantee detection of the predicate.

*supported in part by a Virginia & Ernest Cockrell fellowship

†supported in part by the NSF Grants ECS-9414780, CCR-9520540, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

2 Previous work

Previous work in predicate detection in a non-faulty environment has been numerous. Chandy and Lamport's method [4] for detecting a global predicate involves periodically taking a global snapshot of the state of the system. Snapshots can only be used to detect stable predicates (those that remain true once they become true). Cooper and Marzullo [5] present a method for detecting general predicates which include unstable predicates but their method requires examination of all global states. Manabe and Imase [14] give a method for detecting predicates including channel predicates, using a replay approach. Garg and Waldecker [10] present an on-line method for detecting unstable conjunctive local predicates reducing the state space search. Their method uses a centralized checker process. In [7], Garg et. al. give a centralized method for detecting unstable channel predicates as well as local predicates. In [9], Garg and Chase present a distributed version of this algorithm which employs a token. In [11], Hurfin, et. al. give a distributed method for detecting unstable local predicates using only application messages. Here we present a method for detecting a conjunction of stable and unstable local and channel predicates which is also distributed. We do not use extra control messages.

For predicate detection in a faulty distributed environment, previous work has either not considered general conditions or has required many control messages. Li and McMillin, in [13], give a method for deadlock detection. Other work has been done in termination detection in a faulty environment [19, 18]. Both these assume perfect failure suspicion to ensure that their algorithms detect termination correctly. In Meta [1], an add on to the Isis system, combinations of process conditions can be detected, but this approach requires that any change to a monitored variable be broadcast to all other monitoring processes. This results in many control messages. The method given in this paper provides detection of more general global predicates in a faulty environment. We use suspicions of failures rather than assuming perfect failure detection. The distributed nature of our approach is important in a faulty environment since reliance on a central checker process or token could result in failure of the algorithm.

In the following table we compare our research with previous efforts for detection of predicates which occur across all N processes and for which each process sends m application messages during the monitoring period.

Although we avoid adding control messages in our approach, we do add detection information to application messages. One drawback of our approach is the amount of extra information added for detection of channel predicates. After presentation of the algorithm, we discuss how to reduce this. For example, detection of "no token" can reduce the overall overhead to $O(N)$ additional information as can channel predicates based on aggregate numbers of messages.

In the next section we give our model. We then present ideas necessary to understand our algorithm. Finally we present the algorithm followed by correctness, complexity and a discussion.

3 Model

The environment we consider is a message-passing system of distributed processes without a global clock or shared memory. Every process is connected to every other process by a channel. A

Research	Predicates detectable	Fault requiremts	Additional messages	Message overhead
Garg, Waldecker [10]	process conjunctions	No faults	$O(mN)$	$O(N)$
Garg, et. al. [7]	Process/channel conjunctions	No faults	$O(mN)$	$O(mN)$
Garg, Chase [9]	Process/channel conjunctions	No faults	$O(mN)$	$O(N)$
Hurfin, et. al. [11]	Process conjunctions	No faults	0	$O(N)$
Venkatesan [19]	Termination detection	Perfect detection	-	$O(N)$
Tseng [18]	Termination detection	Perfect detection	-	$O(N)$
Meta [1]	Process	Via Isis	$O(cN^2)^*$	-
Our approach	Process/channel conjunctions	Suspicions	0	$O(m'N)^{**}$

$*$ c = the number of times on average a local predicate changes during monitoring.
$**$ $m' \ll m$ and in many cases the overhead can be reduced to $O(N)$.

Table 1: Comparison of predicate detection algorithms.

channel is a uni-directional connection between any two processes through which messages can be passed. For example, channel (i, j) is the connection through which process i, P_i, sends messages to process j, P_j. The run of P_i is a succession of states, s_i's: $s_{i,1}, s_{i,2}, ...s_{i,l}$, with implied events between successive states. We use $\prec$ and $\preceq$ to represent the ordering of these states. Therefore, $\forall j, k : j < k : s_{i,j} \prec s_{i,k}$ and $\forall j, k : j \leq k : s_{i,j} \preceq s_{i,k}$. We use the relation, $\rightarrow$, to order states in the same manner as Lamport's happened before relation on events [12]. Therefore, if s and t are process states, then $s \rightarrow t$ is the smallest relation satisfying 1) $s \prec t$, 2) s is the state immediately preceding the send of a message and t is the state immediately following the receive of that message, or 3) $\exists u : s \rightarrow u \wedge u \rightarrow t$. A global state, G, is defined as a set of local states such that G contains one state from every process. The state of P_i in G is represented by $G[i]$. Note that G also defines the states of the channels between processes. We also use the $\prec$ relation on global states. $H \prec G \iff \forall i : H[i] \preceq G[i] \wedge \exists j : H[j] \prec G[j]$. A consistent global state is a global state such that for all $s, t \in G : s \not\rightarrow t$. That is, all s, t in G are *concurrent*. For the remainder of the paper, we will only consider consistent global states and represent them by G.

We make the following assumptions about the environment.

- A1: A run is infinite and a process sends messages to every process infinitely often.
- A2: Processes operate correctly until failing by ceasing all activity.

- A3: Channels operate correctly until failing by severing the link; they do not spuriously alter or create messages. All messages sent into a failed channel are lost.

- A4: Messages only fail by disappearing from the channel. They are never altered.

4 Efficiencies of conjunctive predicates

Here we define and give examples of predicates detectable by our algorithm and how detection of such predicates can be efficient. These predicates are made up of conditions on processes and in the channels. That is, local (or process) and channel predicates. A local predicate is defined as any boolean formula on a process. We use LP_i to represent a local predicate at P_i and $LP_i(s)$ to mean that the predicate is true at state s on P_i. A channel predicate is any boolean function on the state of the channel. The channel state is defined as the difference between the number of messages sent into the channel and the number of messages received from it. At states s on P_i and t on P_j the state of channel (i, j) is the set of messages sent before s minus the set of messages received before t. We use CP_{ij} to represent a predicate for the channel from P_i to P_j. $CP_{ij}(s, t)$ means that the predicate for the channel between P_i and P_j is true for the state s at P_i and t at P_j.

As shorthand for predicates true on a global state G, we use $LP_i(G)$ and $CP_{ij}(G)$ to mean $LP_i(G[i])$ and $CP_{ij}(G[i], G[j])$. Our base algorithm detects conjunctions of local and channel predicates. The algorithm uses suspicions to remove either the local process or channel from consideration, while suspected messages are considered to be absent from the channel. The detection of conjunctions can be used to detect any global predicate which can be written as a boolean expression of local and channel predicates [7]. Our detection algorithm is therefore used to detect conjunctive predicates as follows.

Definition 4.1 *A general conjunctive predicate is true in a non-faulty environment iff:*

$$\exists G : \bigwedge_i LP_i(G) \ \wedge \bigwedge_{i,j,i \neq j} CP_{ij}(G)$$

Many useful predicates are of the form given in the above equation. For example, consider predicates which are "bad states." These are states we want to make sure the system doesn't fall into. Some examples of such predicates are: no leader, no token, or all processes and channels idle. For the global predicate "no leader", $LP_i()$ would be "process not leader" and $CP_{ij}()$ would be TRUE. Therefore, if all processes report they are not leaders on a consistent global state G, then the predicate is true. In addition if a process becomes suspected on G it is removed from consideration. Without considering suspicions, a monitoring algorithm would not be able to detect the case where a leader fails. For "no token," channel predicates would be used as well. "All processes and channels idle" could be used to detect that a task has terminated or failed.

The algorithm we present for detection of a global predicate adds predicate information to application messages without the use of control messages. Our scheme sends information needed for detection as part of application messages. That is, information is sent about a process's *causal past*. Figure 1 highlights the causal past of a state of P_2. With the proper state information included in messages, P_2 can maintain information about any state or event in its causal past. Figure 2 shows the last global state G for which P_2 at t has knowledge. Note that this global state is also consistent and that including the proper information with messages, P_2 can have knowledge of all global states H which precede G. So, if a global predicate is true for some consistent global state and enough application messages are exchanged, eventually a point will be reached in P_2's execution when this global state will be in its causal past. Therefore, a global predicate of the form in definition 4.1 is detectable by processes merely by piggybacking information on application messages. Note that with the continued exchange of messages, eventually all processes which do not fail will reach a point at which the predicate can be detected. This fact is important given that the environment we are considering is faulty. That is, any process can detect the predicate; it is distributed rather than relying on a central checker process or a token, either of which, if failed, would cause the algorithm to fail. Only [11] have considered such a completely distributed detection scheme, however, their approach did not detect channel predicates (or consider a faulty environment).

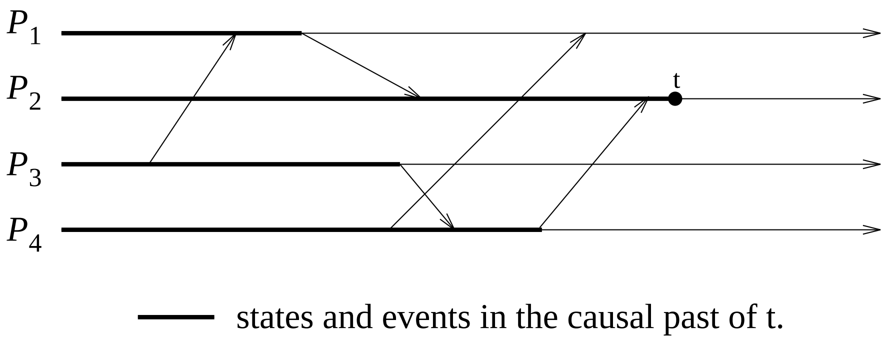

Figure 1: The causal past of P_2 at state t.

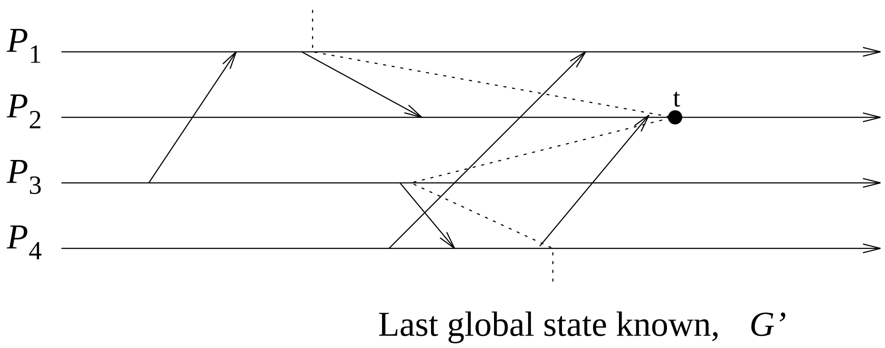

Figure 2: The latest global state known by P_2 at state t.

It is important to understand the nature of the predicates that our algorithm can detect. Conjunctive predicates allow us to reduce the number of global states to consider from $O(m^n)$ to $O(mn)$. This is first because a conjunction across a set of processes is equivalent to finding a state at every process in this set which could have occurred at the same time as all the other process states. That is, the conjunction of local predicates must lie on a consistent global state. Secondly, by detecting conjunctions, we can efficiently remove states from consideration. For example, this is much easier than finding any combinations of conjuncts and disjuncts across the system.

To give further insight, consider a predicate "All processes are in state blue." This predicate is

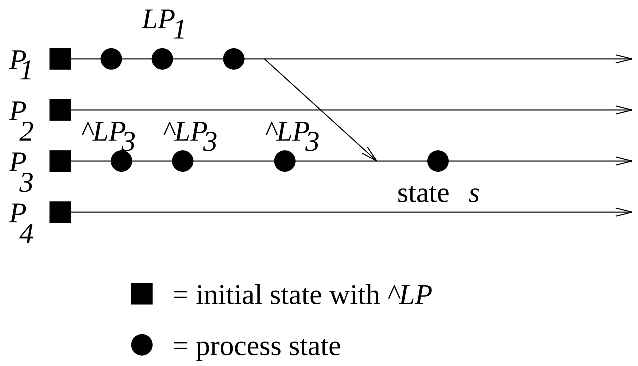

Figure 3: A four process computation with process predicates

a conjunction across all processes and is of the form $\bigwedge_{i \in P} LP_i$. Consider a second predicate "Two processes are in state blue." This predicate requires only a conjunct across two processes $\bigwedge_{i \in Q} LP_i$, where Q is any subset of P of size two. However, this second predicate is actually a disjunction of conjunctions which can be written as $\bigvee_{i,j \in P}(LP_i \wedge LP_j)$. Looking at figure 3, consider the first predicate. Because all states preceding s at P_3 have LP_3 false, we can eliminate from consideration all states on P_1 which causally precede s. This is not true for the second predicate. The state on P_1 for which LP_1 is true must still be considered because it might satisfy the predicate with a state from P_2 or P_4. This fact makes detection of disjunctions less efficient.

The property described above for conjunctive predicates allows us to develop an efficient algorithm for their detection. This property can be employed as follows. Given a consistent global state, if any local state of the global state does not have LP true, then *advance* the state of this process past the next send or receive event. Now, advance any other processes, if necessary, to create a new consistent global state.

The channel predicates we detect also allow us to efficiently eliminate many channel states from consideration. We restrict channel predicates to those that are *monotonic* predicates [7].

Definition 4.2 Send monotonic channel predicate —
CP_{ij} is send monotonic iff $\neg CP_{ij}(s,t) \Rightarrow \forall u : s \prec u : \neg CP_{ij}(u,t)$

Definition 4.3 Receive monotonic channel predicate —
CP_{ij} is receive monotonic iff $\neg CP_{ij}(s,t) \Rightarrow \forall u : t \prec u : \neg CP_{ij}(s,u)$

These types of channel predicates allow an algorithm to eliminate consideration of particular global states when trying to detect a global predicate. For example, suppose all local predicates are satisfied on G and channel predicates are receive monotonic. If $\neg CP_{ij}(G)$, a new global state G' can be created where $\forall k : k \neq i : G[k] = G'[k]$ and $G[i] \prec G'[i]$. Therefore, the earlier global state, G, has been removed from consideration. An example of a receive monotonic predicate is: "at least k messages in channel." Our algorithm also handles predicates which are sometimes receive monotonic and sometimes send monotonic. These predicates are called dynamically monotonic.

Definition 4.4 *(***Dynamically monotonic channel predicate***)*
CP_{ij} is dynamically monotonic iff $\neg CP_{ij}(s,t) \Rightarrow (\forall u : s \prec u : \neg CP_{ij}(u,t)) \vee (\forall u : t \prec u : \neg CP_{ij}(s,u))$

Examples of these predicates include: "no marker," and "channels empty." We note that monotonic channel predicates represent the set of "useful" channel predicates. Example of non-monotonic predicates include: "even number of messages in channel."

5 Predicate detection in an error-free environment

In this section we demonstrate how to implement an algorithm which detects predicates in the form of definition 4.1 without using control messages. As shown earlier in figure 1, if enough information is passed with application messages, a process can maintain all information about its causal past. Also, in the previous section we showed how elimination of states from consideration was possible when detecting conjunctions across a set of processes. We use both of these ideas in detecting a global predicate.

First we will discuss how our algorithm detects a conjunction of local predicates, then how it detects a conjunction of channel predicates. This follows the approach of our algorithm which first attempts to find a consistent global state, G, for which $\forall i : LP_i(G)$. Once such a G is found, the algorithm checks to see if $\forall i, j : i \neq j : CP_{ij}(G)$.

To find a consistent global state which satisfies the local predicates, we use the reasoning of figure 2. A process can know of any state s that happened before its current state for which any P_i had $LP_i(s)$ = TRUE. To do this, every P_i can include in all outgoing messages, an indication of which states had the predicate true. There is a way to avoid sending all states however. As shown in figure 4 any states between the same send and receive events on a process are concurrent to the same set of states. In the figure, s, t, u, v are all concurrent to state w. Because of this,

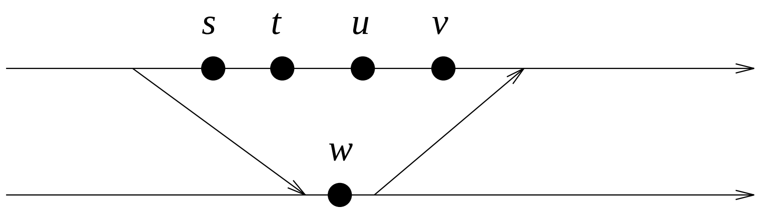

Figure 4: Concurrency of states between the same set of send/receive events.

a process which has multiple states for which $LP()$ is true between a succession of send/receive events only keeps track of one. To do so, each process keeps track of the total number of send and receive events it has performed. This is done using a vector clock [15], VC, where $\mathsf{VC}[i]$ is the vector clock entry for P_i. $\mathsf{VC}[i]$ is only incremented for sends and receives of messages on P_i. Therefore, for the VC on P_i, $\mathsf{VC}[i]$ is the number of sends and receives performed by P_i, and $\mathsf{VC}[j]$ is the number of sends and receives performed by P_j in P_i's causal past. All $\mathsf{VC}[i]$ uniquely identify the set of states between successive sends/receives. For our purposes here, $\mathsf{VC}[i]$ will represent the state immediately following the ith send or receive. Our algorithm will assume that if $LP_i(\mathsf{VC}[i])$ = TRUE, then the local predicate was true for the state represented by $\mathsf{VC}[i]$. If necessary, it is not a difficult matter to include a little more information to pinpoint the exact state. Because of our use of $\mathsf{VC}[i]$ and because $\mathsf{VC}[i]$ is incremented before every send, VC always represents a set of concurrent states, that is, a consistent global state.

We can further reduce the amount of information sent with messages. We do this using a global state named First_state. First_state[i], like VC[i] represents the set of states on P_i following the ith send or receive (as for VC). All states on P_i which precede First_state[i] are states which have been eliminated from consideration using the ideas presented in section 4. The VC on P_i represents the most recent global state of which P_i knows, and First_state is either equivalent to or preceding VC. A process need only maintain process and channel information about the computation between First_state and VC inclusive. When sending information about local predicates, we will show that it is only necessary to include First_state and VC.

Figure 5 shows an example of First_state and VC maintained by P_3. In addition to maintaining

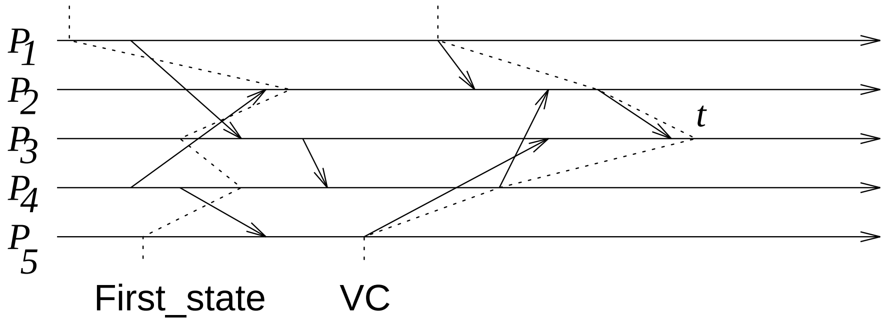

Figure 5: The global states maintained by a process: VC and First_state.

global states, our algorithm needs to know for which local states on these global states the local predicate was true. Therefore, for both global states, VC and First_state, two fields are maintained for each entry of the vectors. The first is val, which is incremented with every send and receive. VC[i].val $= 4$ represents the state immediately following the 4th send/receive event at P_i. The second field is LP which is TRUE or FALSE depending on the evaluation of the local predicate for the state specified by val. LP can take a third value which we will introduce when discussing channel predicates. First_state[i].val and First_state[i].LP work the same as VC. Since every global state which precedes First_state has been eliminated from consideration (because they cannot satisfy the global predicate), our algorithm checks for the condition $\forall j$: (First_state[j].LP $=$ TRUE). When this is satisfied First_state is the first global state for which the conjunction of local predicates is true. This is similar to the approach used in [11].

5.1 Maintaining First_state

In order to perform detection of a predicate, each process must maintain and update data in addition to First_state and VC. As part of the procedure for detection, P_i adds its VC to a list of global states whenever its local predicate becomes true. This list is called State_list and contains global states which may satisfy the predicate. Since VC represents a consistent global state, State_list contains only consistent global states. First_state begins with the initial consistent global state consisting of process initial states (that is, val for every entry equal to 0). First_state is sent with every message and combined by the receiver with its own First_state to create a new one. This operation creates a new First_state by taking the maximum val for each entry from the local and received First_states. LP for each entry is taken from the vector which had the maximum val. For identical vals the logical

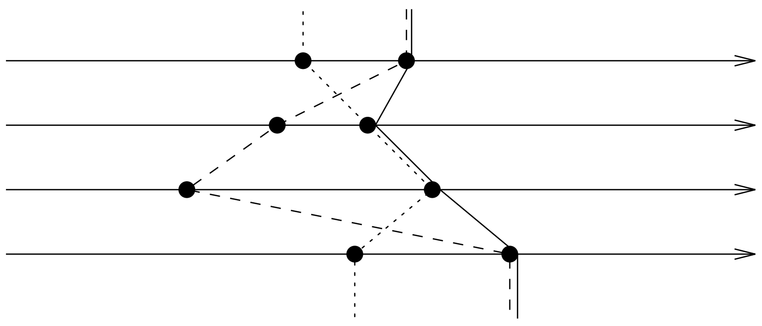

Figure 6: Combining two consistent global states to obtain the latest global state.

OR of LPs is used. This combination occurs because two processes have First_states before which all local states cannot be used to satisfy the predicate. This is key to our algorithm. We also note that every new First_state created is a consistent global state. Figure 6 gives the intuition behind combining two consistent global states. The created global state is a consistent global state. This follows the reasoning behind combinations of consistent cuts as presented by Mattern in [15].

Following the update of First_state when receiving a message, every VC in State_list which precedes the new First_state is removed (these cannot satisfy the predicate since First_state represents the first global state which possibly can). Finally, if State_list is not empty (this is a FIFO queue), the global state which is at the top of the list is combined with First_state. Again, we do this because State_list contains all global states for which the local predicate on this process was true. Any global state removed from this list could not satisfy the predicate. Therefore, after these removals, the top global state is the earliest which may satisfy the predicate and should be combined with First_state. If, following removals, State_list is empty, then this means there are no earlier states than VC[i] that can possibly satisfy the predicate, therefore, First_state is set to VC, a consistent global state.

The final step when receiving is to check if $\forall i$: First_state[i].LP = TRUE. If so, First_state is a global state which satisfies the conjunction of local predicates.

5.2 The states of channels associated with First_state

Up to this point we have described the method used to track consistent global states for local predicates. To extend this to include the states of the channels for the consistent global states is fairly simple. Every process is considering one global state: First_state. The states of the channels for First_state are determined by those messages sent before but received after the global state. That is, any message, m, sent from P_i to P_j before the state First_state[i] and not received on or before the state First_state[j] is in channel (i, j) for First_state. If a message, m, was sent before First_state[i] any process which knows of the state First_state[i] can know of m. Our algorithm keeps track of all messages, discarding a message when its receive event precedes First_state. Then, only those messages remaining which were sent before First_state are in channels.

Recall that for our channel predicates we detect only monotonic channel predicates. A monotonic channel predicate, when false, can be made true by either sending more messages or receiving more messages. This is necessary or the algorithm will not know how to move First_state forward if a channel predicate is not true. After determining that First_state satisfies the conjunction of local predicates, if some channel does not satisfy the channel predicate, then based on monotonicity, the state of the sender or receiver on the channel can be advanced. We do this by setting First_state[l].LP = CHAN_FALSE, where P_l is the process which needs to advance. So, the LP field can take values TRUE, FALSE, and CHAN_FALSE. The setting of CHAN_FALSE is used by the algorithm to advance the cut when a channel predicate is not true.

Message information is maintained in Msg_list at each process. For every message, m in Msg_list, a record is maintained with the following entries: *m.source, m.dest*– the source and destination, *m.send*– the state number preceding the send of m, *m.rcvd*– the state number preceding the receive of m, and *m.profile*– any information necessary for detection of the channel predicate. Examples of *m.profile* are message type, size, etc. For channel predicates which check for numbers of messages, *m.profile* is empty.

5.3 The algorithm for error-free environments

We now present our algorithm for detecting conjunctions of local and channel predicates in an error-free environment. The variables with their initial values and functions used by the algorithm follow.

- VC, First_state – array[1...n] of structure:
  integer val - the numerical value of a state; initially 0.
  boolean LP - TRUE if $LP()$ for state val;
  FALSE if unknown or $\neg LP()$ for state val;
  CHAN_FALSE if $LP()$ for state val, but $CP()$ has shown the state to not satisfy global the predicate.
  Initially FALSE.

- State_list – FIFO set of VCs; initially empty.

- Msg_list – set of messages sent in the causal past and not received before First_state; initially empty.

- Msgs_in_channels – set of messages sent before but received after First_state; initially empty.

- max(vector X, vector Y) – function which takes two arrays as inputs and returns: for all i:
  val = max(X[i].val, Y[i].val),

$$\mathsf{LP} = \begin{cases} \mathrm{X}[i].\mathsf{LP} & \text{if } \mathrm{X}[i].\mathsf{val} > \mathrm{Y}[i].\mathsf{val} \\ \mathrm{Y}[i].\mathsf{LP} & \text{if } \mathrm{Y}[i].\mathsf{val} > \mathrm{X}[i].\mathsf{val} \\ \max(\mathrm{X}[i].\mathsf{LP}, \mathrm{Y}[i].\mathsf{LP}) & \text{if } \mathrm{Y}[i].\mathsf{val} = \mathrm{X}[i].\mathsf{val} \end{cases}$$

  where for max(LP,LP), CHAN_FALSE > TRUE > FALSE.

The algorithm is given in figure 7. A description of statements in the algorithm follows.

- 1-2: The local predicate has become true for the first time in this state. If First_state[k].val = VC[k].val then the state for P_k in First_state should indicate that the predicate was true. So,

```
When LP_k(VC[k])= TRUE for the first time for VC[k]:
    if (First_state[k].val = VC[k].val) then
        First_state[k].LP ← TRUE;
    VC' ← VC; VC'[k].LP ← TRUE;
    add VC' to State_list;

When sending msg m:
    m.send ← VC[k].val;
    add record m to Msg_list;
    if (State_list = NULL) then First_state ← VC;
    VC[k].val ← VC[k].val + 1;
    send VC, First_state, and Msg_list as part of message;

When receiving msg m:
    Receive VC_m, First_state_m, and Msg_list_m from msg;
    m.rcvd ← VC[k].val;
    VC ← max(VC, VC_m);
    VC[k].val ← VC[k].val + 1;

    First_state ← max(First_state, First_state_m);
    State_list ← {VC' ∈ State_list | (VC'[k].val > First_state[k].val) ∨
                 (VC'[k].val = First_state[k].val ∧ First_state[k].LP ≠ CHAN_FALSE)};
    if (State_list = NULL) then First_state ← VC;
    else First_state ← max(First_state, top entry of State_list);

    Msg_list ← {m' ∈ Msg_list ∪ Msg_list_m |
            First_state[m'.dest].val < m'.rcvd};

    if (for all j: (First_state[j].LP) then
        Msgs_in_channels ← {m' ∈ Msg_list | m'.send < First_state[m'.source].val};
        if ((proc ← check_channel_pred(Msgs_in_channels)) = 0) then issue detected;
        else First_state[proc].LP ← CHAN_FALSE;
```

Figure 7: The algorithm for detecting conjunctions of predicates at P_k in an error-free environment.

set First_state[k].LP = TRUE. (Recall that VC is the earliest consistent global state containing the current state.)

- 3-4: Add VC as a possible global state to State_list since the current state had the local predicate true. This operation simply allows us to place the earliest consistent global state which includes the current state in State_list.
- 5-6: Add a record for the message to be sent, and assign $m.send$ as the state before the send.
- 7: Advance First_state to VC, the last consistent global state known by P_k, if State_list = NULL and this process is not suspected. State_list will only equal NULL at this point if the last receive cleared out State_list. So, this assignment simply keeps First_state up to date if multiple sends have occurred since then.
- 8: Increment state number before sending so VC always represents a consistent global state.
- 11: Assign the received state of m as the state before the receive.
- 12-13: Update VC.
- 14: Update the receiver with information about the first possible global state from the sender. That is, if the sender's first possible global state is later than the receiver's, the receiver accepts the later states to form a new global state. Note that this operation always creates a consistent global state, since taking the intersection of two consistent global states (to create the latest possible) always creates a consistent global state.
- 15: Since First_state represents the earliest possible global state which can satisfy the predicate, remove any global states in State_list which occur before it. Also remove VC that is the same as First_state if a channel predicate for P_k has been found false and requires that P_k advance.
- 16: If State_list is empty, then P_k does not have any earlier states than VC which can be used to create a global state which satisfies the predicate. Therefore, set the first possible global state to VC, which is the earliest consistent global state which includes the current state.
- 17: If State_list is not empty, combine the top entry with the current First_state. Again, the resultant global state will be consistent, since all global states in State_list and First_state are consistent. Note that if the state from the top entry is used (because it is later than First_state), the value of First_state[x].LP is set to FALSE, except for First_state[k].LP, due to the assignment of VCs to State_list. This operation reflects what P_k knows about the predicate for a particular state.
- 18: Update Msg_list and remove any messages that were received before First_state; these messages are no longer needed.
- 19: If for all processes, the local predicate is true in First_state, then proceed.
- 20: Put all messages in Msg_list that were sent before First_state in the list Msgs_in_channels.

```
int check_channel_pred(list Msgs)
    list Cur_channel;
    for all i, j, i ≠ j do
        Cur_channel ← {m ∈ Msgs | m.source = i ∧ m.dest = j};
        if channel predicate is FALSE for Cur_channel then
            if sending more messages may make channel predicate true then return i;
            if receiving more messages may make channel predicate true then return j;
    return 0;
```

Figure 8: The function *check_channel_pred()* for P_k.

- 21: Call the function to check channel predicates with the Msgs_in_channels list. If it returns 0, then for all channels, the predicate is true. Otherwise, the function returns either the send or receive process of a channel that did not satisfy the predicate.
- 22: Since the global predicate was not satisfied, set First_state[*proc*].LP = CHAN_FALSE for the 'proc' value returned. This will force the global state to advance to a later state of 'proc'.

The function *check_channel_pred()* is given in figure 8. This function checks every channel. If the predicate is not true for some channel, the function determines whether the sending or receiving process id should be the return value, based on the channel predicate. If all channel predicates are true, 0 is returned.

In the following sections we present the algorithm for faulty environments. We prove correctness there.

6 Predicate detection in a faulty environment

Here we consider failures and how they affect predicate detection. If a process or channel fails before detection of the predicate, our predicate detection algorithm in the previous section will never detect that the remaining set of unfailed processes and channels have satisfied the predicate. We want to detect predicates like "request no longer being serviced," "no token," "greater than 90% loading at processes," and "too many processes suspected." Our algorithm allows us to do this.

Since failures are impossible to detect accurately in an asynchronous environment [6], suspicions of failures must be used in the real world. Let SSP, SSC, and SSM be suspicion sets for processes, channels, and messages. We use SSP_G to uniquely identify SSP on global state G. Similarly for SSC_G and SSM_G. To expand the observation of the system, we create the function $f()$ which operates on the failure sets. These can be used, for example, to limit the number of failures allowed during predicate detection to avoid trivial detection.In addition, we use $SP(G,i)$, $SC(G,(i,j))$, and $SM(G,m)$ to mean that P_i, channel (ij), and m are suspected on G, respectively. Next we use the following idea: if a process or channel is failed, we simply exclude it from our detection of the conjunctive predicate and instead only consider live processes and channels. That is, we shrink the number of processes and channels we consider.

Now we can define:

Definition 6.1 *A global predicate is true in a faulty environment iff:*

$$\exists G : f(SSP_G, SSC_G, SSM_G) \wedge \quad \bigwedge_i [LP_i(G) \vee SP(G,i)] \wedge \\ \bigwedge_{i,j,i\neq j} [CP_{ij}(G) \vee SC(G,(i,j))]$$

For shorter notation of our long expression above, we will use:

$$\text{process\_conditions}(G) \equiv \bigwedge_i [LP_i(G) \vee SP(G,i)]$$

$$\text{channel\_conditions}(G) \equiv \bigwedge_{i,j,i\neq j} [CP_{ij}(G) \vee SC(G,(i,j))]$$

and:

$$\text{suspicion\_conditions}(G) \equiv f(SSP_G, SSC_G, SSM_G)$$

Because of message suspicions, we must also modify our definition of $CP()$. Similar to above, we say,

Definition 6.2 *$CP_{ij}(G)$ is TRUE when messages can fail if the predicate for channel (i,j) is true when all $m \in SM_G$ are considered to have never been sent.*

As an example of the use of this equation, suppose we wanted to detect the condition, "No token in system and less than k processes and channels suspected." We would set $\forall i : LP_i() =$ "Token not held," $\forall i,j : CP_{ij}() =$ "Token not in channel," and $f() =$ "Less than k processes and channels suspected."

Our algorithm for error-free systems was completely distributed. This was deliberately done to avoid single failures from corrupting the algorithm when extending it to tolerate failures. Note that the algorithm did not rely on a single checker process as in [10, 7], or a token or leader among a set of processes as in [8]. These approaches all have single points of failure and therefore cannot be extended to operate in a faulty environment. Our approach allows for the correct processes to continue to check for predicates. This will become clear in this section. There are several other considerations which we discuss as well.

When detecting a predicate across a set of processes where failures are possible, it is not enough to use the algorithm for error-free environments from the last section. Suppose we wish to detect "All processes are in state blue." It's obvious that one process may fail before ever reaching the blue state and the predicate will not be detected even though all other processes may reach this state. Changing this predicate to "All processes are in state blue and suspect no more than one process," does not correct the problem. This is because including the conjunct "suspecting no more than one process" does not exclude a failed process from having to detect the local predicate. A similar problem occurs for channel failures. Detecting "All channels are empty" would be impossible to detect for any channel or message which failed. In the first case, any messages sent into the channel would never be received. In the second, the lost message would always appear to be in the channel. The result for both is that the channel would never be detected to be empty, even though all messages sent into the failed channel and the lost message may have been retransmitted and received. Figure 9 shows the scenario for a lost message.

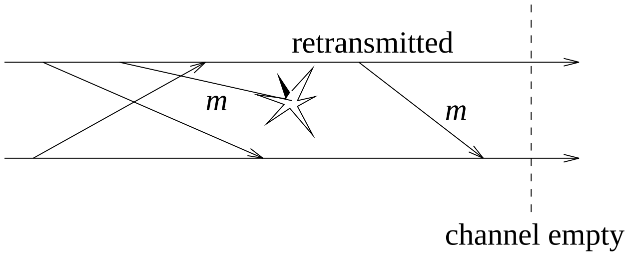

Figure 9: A lost message if not removed from consideration, will not detect an empty channel.

Because of the above problems, we first modify the algorithm from the previous section so that suspected processes and channels are excluded from consideration. Earlier we gave predicates which are useful to detect in this approach: "no leader," "no token," or "all processes and channels inactive." Because we exclude suspected processes, these predicates effectively monitor that failures have not put a computation in jeopardy of making progress. In this case, it would be because of the possibility of failure of the leader, loss of a token, or failure of critical processes. A process cannot, however, simply exclude processes and channels from consideration which it *currently* suspects. This is because First_state lags a process's current view of the computation, VC. To solve this problem, we require that suspicion information include the state at which suspicion started for both processes and channels. For example, if after state s, P_j suspects that process P_i has failed, s would be part of the process suspicion tag. Our algorithm would then not consider the suspicion of P_i until $s \preceq$ First_state$[j]$. This is important because the global predicate may have become true before First_state$[j] = s$, so we don't want a suspicion to mask this unnecessarily. Figure 10 demonstrates this. For message suspicion it is sufficient to track the message number without the state in which it was suspected. This is because the suspicion will only be used once First_state moves past the send event of the suspected message.

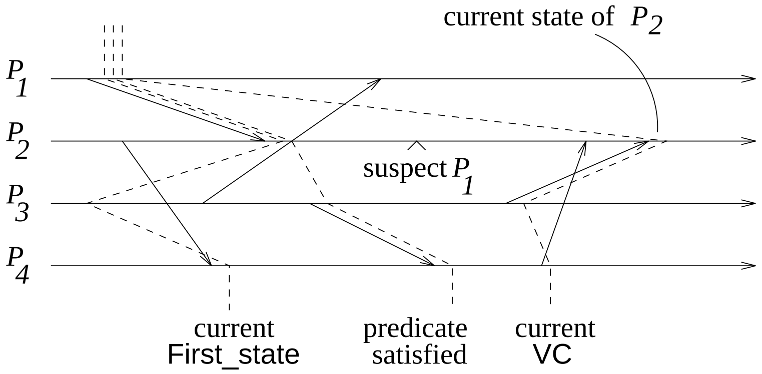

Figure 10: First_state and VC at P_2.

Next we must ensure that suspicion information is available for all suspicions that occurred before First_state. To do so, we require that suspicion information be causally spread. We note then that if P_j begins suspecting P_i before the state First_state$[j]$ as maintained by P_k, then SP(First_state,i) will be true at P_k before First_state$[j]$. Figure 11 illustrates this. Therefore, P_k will have SP(First_state,i) when evaluating the global state. The approach for CP$(G, (i, j))$ is similar.

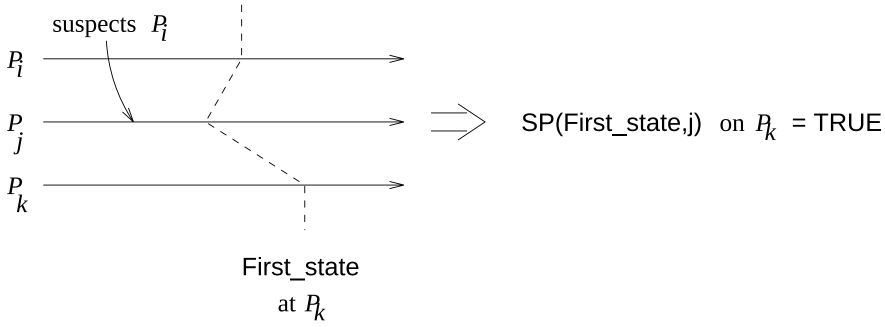

Figure 11: The guarantee that any suspicion on G will be known by any process considering G.

The next consideration is the removal of suspicions. Our algorithm allows for the removal of all suspicion types. For process suspicions, when a removal occurs, the algorithm could handle the situation in one of two ways: 1) require that the state in which removal occurred be stored and continue using suspicion until First_state reaches this removal state, 2) stop using suspicion immediately upon knowing of this removal. We implemented the second approach because the removal of suspicion means the process is now assumed to be unfailed. Therefore, we should utilize this knowledge as soon as possible, since the goal is to find predicates across unfailed processes. The same applies to channel and message suspicions. It should be noted, however, that because we do not make assumptions about suspicion removal, we have code in our algorithm which ensures correctness when a process is live but receives a suspicion of itself. When this happens the process does not update First_state, instead it simply resends the global state. These lines in our algorithm are not needed if suspicions are automatically removed when received by the suspected process.

Other considerations are of channel and message suspicions. Suppose there are process suspicions that are active on First_state, that is, processes were suspected before First_state. If First_state satisfies the conjunction of local predicates with these suspicions, then when checking channel predicates all channels incident to suspected processes are also excluded from consideration. We do this because the states of channels incident on any process we assume to be failed are undefined. Next, the consideration of message suspicions concerns how the suspicion affects the state of the channel. There are two approaches: 1) if a message is suspected assume it is not present in the channel, 2) if a message is suspected assume that the suspicion will always trigger a resend and therefore the message can be considered to be in the channel. We use the first approach because we do not make any assumption about message recovery. In addition, we assume any retransmitted message will be tracked for failures as well.

As stated above, the suspicion information input to our algorithm should contain certain information. So we define suspicion tags as follows:

- Process suspicion – (i, s, j) – P_i is suspected after state s on P_j.
- Channel suspicion – (i, j, s, t) – The channel from P_i to P_j is suspected after state s on P_i and t on P_j.
- Message suspicion – (i, j, l) – message l sent from P_i to P_j is suspected.

We define $SP(G,i)$, $SC(G,(i,j))$, and $SM(G,m)$ in terms of the suspicion sets and the suspicion tags.

Definition 6.3
$SP(G,i) \iff \exists$ *a tag* (i,l,s,j) *in* SSP: $s \preceq G[j]$.
$SC(G,(i,j)) \iff \exists$ *a tag* (i,j,s,t) *in* SSC: $s \preceq G[i]$ *and* $t \prec G[j]$.
$SM(G,m) \iff \exists$ *a tag* (i,j,l) *in* SSM *such that the source of* m *is* P_i, *the destination is* P_j *and the message number is* l.

These definitions are used to determine whether processes, channels, or messages should be excluded from consideration. We use functions named $\mathsf{SP}(G,i)$ and $\mathsf{SC}(G,i)$ which implement $SP(G,i)$ and $SC(G,i)$ of definition 6.3.

Since we use suspicions of failures, and we know that perfect failure detectors as defined by [3] are not implementable in an asynchronous environment, the operation of the failure detectors directly affect our algorithm. If we are using strong-complete failure detectors, that is, eventual detection of failures, but possible suspicion when a failure has not occurred, the only problem is the case in which $SP(G,i)$ and P_i not failed on G. Then, as we have indicated earlier, even though $LP_k(G[k]) = \text{FALSE}$ and P_i has not failed at G it is possible that $\bigwedge_i [LP_i(G) \vee SP(G,i)$. That is, basing detection on strong-complete failure detectors means that our predicate detection algorithm is also strong-complete: it will always eventually detect a predicate that is true, but may inadvertently detect a predicate that appeared to be true because of a false suspicion. It should be apparent that the accuracy of our algorithm is directly affected by the accuracy of suspicions. Any monitoring application needs to be aware of this fact. The predicates we have given as examples can be used in spite of this fact.

Since strong-complete failure detectors are easy to implement (and we believe the only type of failure detectors that are realizable in our environment), we assume the use of these. This means our algorithm can be characterized as strong-complete. Based on this, our presentation and proofs of correctness will show that if a suspicion is held long enough (any correct suspicion would) that our algorithm will correctly detect a predicate which is true.

6.1 The algorithm for faulty environments

In this section we give our algorithm for detecting predicates in the form of definition 6.1:

$$\exists G : \text{process\_conditions}(G) \wedge \text{channel\_conditions}(G) \wedge \text{suspicion\_conditions}(G)$$

Using our definitions of suspicion (6.3), we implement the local predicate detection for the above equation as:

$$[LP_i(G) \vee \mathsf{SP}(G,i)] = [LP_i(G) \vee SP(G,i)]$$

For channel predicates, we implement detection as:

$$[CP_{ij}(G) \vee \mathsf{SC}(G,(i,j))] = [CP_{ij}(G) \vee SC(G,(i,j))]$$

For the suspicion function we use:

$$\mathsf{f}(G) = f(SSP_G, SSC_G, SSM_G)$$

Following the same approach as with the earlier algorithm for error-free environments, the algorithm here first attempts to find a consistent global state, G, for which $\forall i : LP_i(G) \vee \mathsf{SP}(G,i)$. Once a G is found, the algorithm checks to see if $\forall i,j : i \neq j : CP_{ij}(G) \vee \mathsf{SC}(G,(i,j))$ and that $\mathsf{f}(G)$.

Following we give the entire algorithm for predicate detection in a faulty environment. Descriptions of the lines which implement key features of the algorithm follow.

- 7: Advance First_state to VC, the last consistent global state known by P_k if State_list = NULL and this process is not suspected. State_list will only equal NULL at this point if the last receive cleared out State_list. So, this assignment simply keeps First_state up to date if multiple sends have occurred since then. Note that suspected processes do not advance First_state, an operation which could cause a global state to be skipped improperly.

- 14: If this process is suspected, then simply assign First_state from First_state of the received message, otherwise, perform lines 17-34. This is done so that suspected processes will not advance First_state, an operation which could cause a global state to be skipped improperly.

- 21: If for all processes, either the local predicate is true in First_state or the process is suspected and the number of suspicions is less than s_p, proceed.

The function *check_channel_pred()* is given in figure 13 for a faulty environment. For every channel, if the predicate is not true and the channel is not suspected, the function determines whether the sending or receiving process id should be the return value, based on the channel predicate. If all channel predicates are true, 0 is returned. All suspected messages are considered to be lost. Also note that any process which is suspected causes suspicion of its incident channels. Line C1 checks to see if $\mathsf{f}(G)$ is satisfied. If not, the global state is advanced for P_k. We must advance P_k in this case, because it knows of suspicions it generates.

6.2 Correctness

We now present the proof of correctness for our conjunctive predicate algorithm in a faulty environment. For ease in reading, we will use F to represent First_state and V to represent VC in the proofs. We also continue to use F[i] to mean the state uniquely identified by F[i].val and F to specify the global state it represents. The same is true for V[i]. Therefore, we use the relation $\prec$ to relate F and V with global states such as G and H. Lines 1-4, 5-9, and 10-24 are considered atomic for the proofs. Therefore, when a variable is referenced in our proofs, lines 1-4, 5-9, and 10-24 have completed. The first lemma shows that F is always consistent.

Lemma 6.4 *F always represents a consistent global state.*

Proof: We note that the vector clock, V, always defines a consistent global state. This is true because before a message is sent from P_i, V[i] is incremented. When the destination process receives the message, it advances the global state past the send and receive event, always leaving a consistent global state. F is set to V in lines 7 and 18. F is also initialized to a consistent global state ([0,...,0]). All other times that F is modified, it is combined or reassigned. It is only reassigned in line 14 to F from a message. In lines 16 and 19 it is combined with F from a message, or the top entry of

```
When LP_k(VC[k])= TRUE for the first time for VC[k]:
1.      if (FS[k].val = VC[k].val) then
2.              First_state[k].LP ← TRUE;
3.      VC' ← VC; VC'[k].LP ← TRUE;
4.      add VC' to State_list;

When sending msg m:
5.      m.send ← VC[k].val;
6.      add record m to Msg_list;
7.      if (State_list = NULL ∧¬SP(VC,k)) then First_state ← VC;
8.      VC[k].val ← VC[k].val + 1;
9.      send VC, First_state, and Msg_list as part of message;

When receiving msg m:
10.     Receive VC_m, First_state_m, and Msg_list_m from msg;
11.     m.rcvd ← VC[k].val;
12.     VC ← max(VC, VC_m);
13.     VC[k].val ← VC[k].val + 1;

14.     if (SP(First_state,k)) then First_state ← First_state_m;
15.     else do
16.             First_state ← max(First_state, First_state_m);
17.             State_list ← {VC' ∈ State_list | (VC'[k].val > First_state[k].val) ∨
                        (VC'[k].val = First_state[k].val ∧ First_state[k].LP ≠ CHAN_FALSE)};
18.             if (State_list = NULL) then First_state ← VC;
19.             else First_state ← max(First_state, top entry of State_list);

20.     Msg_list ← {m' ∈ Msg_list ∪ Msg_list_m |
                First_state[m'.dest].val < m'.rcvd ∧ m' ∉ suspicion_list};

21.     if (for all j: First_state[j].LP ∨ SP(First_state,j)
          then
22.             Msgs_in_channels ← {m' ∈ Msg_list | m'.send < First_state[m'.source].val};
23.             if ((proc ← check_channel_pred(Msgs_in_channels)) = 0) then issue detected;
24.             else First_state[proc].LP ← CHAN_FALSE;
```

Figure 12: The algorithm for detecting global predicates in a faulty environment at P_k.

```
int check_channel_pred(list Msgs)
      list Cur_channel;
C1       if f(First_state) = FALSE then return k;
C2       for all i, j, i ≠ j do
C3             Cur_channel ← {m ∈ Msgs | m.source =i ∧ m.dest = j ∧ m¬ SM(G, m)};
C4             if channel predicate is FALSE for Cur_channel ∧
               ¬ SP(First_state,i) ∧ ¬ SP(First_state,j)∧
               ¬ SC(First_state,(i, j)) then
C5                   if sending more messages may make channel predicate true then return i;
C6                   if receiving more messages may make channel predicate true then return j;
C7       return 0;
```

Figure 13: The function *check_channel_pred()* for P_k.

State_list. The first is initially a consistent global state; the second is always consistent. It is now only necessary to show that combining two consistent global states to get the latest global state produces a consistent global state, and this is a standard result [15]. $\square$

The next lemma shows that F will be advanced if it does not represent G, a global state which satisfies the predicate. This lemma demonstrates that our algorithm is live.

Lemma 6.5 *As long as there is some P_k that is alive and some G which satisfies the global predicate, then $\forall s$ on P_k:*
$(\neg process\_conditions(s.F) \vee \neg channel\_conditions(s.F) \vee \neg suspicion\_conditions(s.F)) \Rightarrow \exists t : s \prec t : s.F \prec t.F$

Proof:

We show that the lemma statement holds for both disjuncts of the LHS of the implication:

We consider the first disjunct: $\neg$process_conditions(s.F)

$\Rightarrow \quad \exists j : \neg LP_j(s.\mathrm{F[j]}) \wedge \neg SP(s.\mathrm{F},j)$

There are two cases:

i) k=j

$\Rightarrow$ $\neg s$.F[k].LP $\wedge\neg$SP(s.F,k)

{s.F[k].LP could only be set TRUE in lines 2 or 3. Also, by process suspicion definition 6.3.}

$\Rightarrow$ s.F $\neq$ s.State_list$_{top}$

{s.State_list$_{top}$[k].LP is always TRUE (by lines 3 and 4), therefore, they cannot be equal.}

$\Rightarrow$ s.F[k].val $>$ s.State_list$_{top}$[k].val

{By definition of max(), s.F[k].val must be greater or else s.F[k].LP would be TRUE.}

$\Rightarrow$ s.State_list = NULL

{By line 17, all entries for which s.F[k].val $>$ s.State_list$_{entry}$[k].val are removed. If this is not true for the top entry, then s.State_list must be empty.}

$\Rightarrow$ $\exists u : s \prec u : LP_k(u)$

{If s.State_list is empty, then up to s no state at P_k could be used to satisfy the global predicate. Because G must exist, there must be a later state at which $LP()$.}

$\Rightarrow$ $\exists t : s \prec t : s.\mathrm{F} \prec t.\mathrm{F}$

{By assumption A1, eventually a message will be received. Let the state after the receive be t. When this occurs, u.V will be at the top of t.State_list and either t.F will be advanced to u.V, or to F_m (the global state received with the message) by lines 14-19.}

ii) k $\neq$ j

$\Rightarrow$ $\exists v : s$.F[j] $\prec v \wedge \forall$ V $\in v$.State_list: V[j].val $> s$.F[j].val

{By assumption A1, there must eventually be a message sent to P_j from P_k after state s. By line 17, all entries from v.State_list that are earlier than s.F[j].val will be removed.}

There are two subcases:

iia) v.State_list = NULL

$\Rightarrow$ v.F[k].val = v.V[k].val

{By line 18, if v.State_list is NULL, then v.F is set to v.V.}

$\Rightarrow$ v.F[k].val $> s$.F[k].val

{Because of previous result and because v.V[k].val must be greater than s.F[k].val since s.F[k] $\prec v$.V[k] (v.V[k] is the state following the send to P_j which must follow s.F[k]).}

iib) v.State_list $\neq$ NULL

$\Rightarrow$ v.F[k].val = max(s.F[k].val, v.State_list$_{top}$[k].val)

{By line 19, since v.State_list is not NULL.}

$\Rightarrow$ v.F[k].val = v.State_list$_{top}$[k].val

{By previous result, v.State_list only contains entries with V[k].val $> s$.F[k].val.}

$\Rightarrow$ v.F[k].val $> s$.F[k].val

The result of iia) and iib)

$\Rightarrow$ $\exists t : s \prec t : s.\mathrm{F} \prec t.\mathrm{F}$

{Eventually a message will be sent from P_k to the process with s (by A1). This message will contain a F_m such that F_m[k].val $\geq v$.F[k].val. And we already know that v.F[k].val $> s$.F[k].val.}

Next consider the second disjunct: $\neg$channel_conditions(s.F)

$\Rightarrow$ $\exists i, j : \neg CP_{ij}(s.\mathrm{F[i]}, s.\mathrm{F[j]}) \wedge \neg SC(s.\mathrm{F}, (i, j))$

For $| SC_{s.\mathrm{F}} | > s_c$:

$\Rightarrow$ channel predicate for (i,j) is FALSE $\wedge$

$\neg$ SC(s.F,(i,j))

{The channel predicate for (i,j) could not be TRUE and it is not suspected.}

$\Rightarrow$ Let $l = i$ or j: s.F[l].LP = CHAN_FALSE
{By definition of function check_channel_pred().}

$\Rightarrow$ $\exists v : s$.F[i] $\prec v \wedge \forall$ V $\in v$.State_list: V[j].val $>$ s.F[j].val
{By same reasoning as in case ii) of $\neg$process_conditions(s.F) and the removal of an equivalent state if v.F[l].LP = CHAN_FALSE.}

$\Rightarrow$ $\exists t : s \prec t : s$.F $\prec t$.F
{The same as for case ii).}

Finally consider the third conjunct: $\neg$suspicion_conditions(s.F)

$\Rightarrow$ f(s.F) = FALSE

$\Rightarrow$ s.F[k].LP = CHAN_FALSE
{By lines C1 and 24 and by definition of function check_channel_pred().}

$\Rightarrow$ $\exists t : s \prec t : s$.F $\prec t$.F
{By assumption A1, eventually P_k must receive another message, when it does, by lines 17-19, s.F will be moved forward at least for P_k.}

□

The next lemma demonstrates that all global states preceding F do not satisfy the global predicate.

Lemma 6.6 *For all i, s on P_i:*
$\forall$ H: H $\prec$ s.F: $\neg$(process_conditions(H) $\wedge$ channel_conditions(H) $\wedge$ suspicion_conditions(H))

Proof: Let the lemma statement be referred to as invariant I. To prove I we first give an invariant of State_list, and call it invariant IC:

- All states, t, which satisfy $LP_i(t)$ are placed in State_list at P_i.
  {Lines 3-4 do this when $LP()$ becomes TRUE for the first time in any state.}

We now prove I.
For s = initial state: s.F = 0,...,0 $\Rightarrow \nexists$ H: H $\prec$ s.F. Therefore, I is satisfied.

For $s \neq$ initial state: There are two cases:

i. P_i knows it is suspected:
There is one statement which increases s.F[j].val for some entry(ies) j for this case.

- Line 14: s.F $\leftarrow$ F$_m$ — I holds for F$_m$, therefore I holds for s.F.

ii. P_i is not known to be suspected:
There are five statements which increase s.F[j].val for some entry(ies) j for this case. We show that I continues to hold for these statements if it held initially.

- Line 16: s.F' $\leftarrow$ max(s.F, F$_m$) — Since I holds for s.F and we are only considering conjunctive predicates, all states preceding s.F can never be used to satisfy the predicate. The same is true for F$_m$. Therefore, this is also true for s.F' and so, I holds for s.F'.

- Line 17: All V' earlier than s.F are removed from s.State_list. All V' equal to s.F, where s.F[i].LP = CHAN_FALSE, are removed from s.State_list — All V' where V'[i] $\prec$ s.F[i], cannot possibly satisfy the global predicate, since I holds for s.F. Also, all V' where V'[i].val = s.F[i].val and s.F[i].LP = CHAN_FALSE, have been shown to not satisfy the predicate. Now, we note that for any t at P_i, t.V always defines the earliest consistent global state which includes t. Also, the earliest state that satisfies $LP()$ at P_i is the top entry of s.State_list (recall by IC, *all* states for which $LP()$ is TRUE are put in State_list). Therefore, I holds for the top entry of s.State_list.
- Lines 18 and 7: s.F $\leftarrow$ s.V if s.State_list = NULL — s.State_list = NULL implies there are no states before s which can be used to satisfy the global predicate (by IC and line 17). Therefore, I holds for s.V (s.V is the earliest consistent global state which includes s), and s.F.
- Line 19: s.F' $\leftarrow$ max(s.F, s.State_list) — I holds for s.F and s.State_list, therefore I holds for s.F'.

$\square$

We next demonstrate that messages are included in Msg_in_channels iff they cross the global state defined by F.

Lemma 6.7 *For all s,*
$\forall m : m \in s.\textsf{Msgs\_in\_channels} \iff$ *m sent but not received before s.F.*

Proof: By lines 20 and 22:

$\forall m : m \in s.\textsf{Msgs\_in\_channels}$

$\Rightarrow$ m.send $\prec$ s.F[m.source] $\wedge$ s.F[m.dest] $\prec$ m.rcvd
{First conjunct from line 22, second from line 20.}

$\Rightarrow$ m crosses global state s.F
{Definition of a message crossing a global state.}

$\forall m : m$ crosses global state s.F

$\Rightarrow$ m sent before s.F and not received before s.F
{Definition of consistent global state.}

$\Rightarrow$ m.send $<$ s.F[i].val $\wedge$ m.rcvd $\geq$ s.F[i].val
{By lines 5-6,9, and 11.}

$\Rightarrow$ $m \in$ Msg_in_channels
{By lines 20 and 22.}

$\square$

The next theorem demonstrates that our algorithm detects correctly.

Theorem 6.8 *The algorithm issues* detected *iff:*

$$\exists G : \mathit{process\_conditions(G)} \wedge \mathit{channel\_conditions(G)} \wedge \mathit{suspicion\_conditions(G)}$$

Proof: $\Rightarrow$: We first demonstrate that if *detected* is issued that $\exists G$: local_conditions(G). That is,

$$\exists G : \bigwedge_i [LP_i(G) \vee SP(G,i)]$$

The algorithm must satisfy the 'if' condition at line 21 to issue *detected*. Consider a process P_k that satisfies the following statement at state s:

$$\forall j : (\text{F[j].LP} \vee \mathsf{SP}(\text{F,j}))$$

We will show that there is a consistent global state, G, such that if s.F[j].LP then $LP(G[j])$.

1) For any j such that s.F[j].LP

There are two cases:

i) j=k:

$\Rightarrow$ s.F[k].val = s.State_list$_{top}$[k].val)

{If j=k, then s.State_list $\neq$ NULL. Otherwise s.F[j].LP would be FALSE, because line 18 sets F to V which is FALSE for all .LP. Since s.State_list $\neq$ NULL, the result must be true since line 17 removed all entries earlier than s.F[k].val.}

$\Rightarrow$ Let V be State_list$_{top}$ then (LP_k(V[k]) $\wedge$ V[k].val = F[k].val))

ii) j $\neq$ k:

$\Rightarrow$ $\exists m$:(F$_m$[j].LP $\wedge$ F[j].val = F$_m$[j].val))

{For any j $\neq$ k, since P_k cannot set F[j].LP directly, this must have been set via a received message.}

$\Rightarrow$ $\exists$ V at P_j:(LP_j(V[j]) $\wedge$ F$_m$[j].val = V[j].val))

{Another process must have executed lines 1-4 such that there exists a VC in which VC[j].val = F$_m$[j].val.}

From i) and ii):

$\Rightarrow$ s.F[j].val = V[j].val $\wedge LP_j$(V[j])

{When j=k this is implied directly, when j$\neq$k we have already said that F[j].val = F$_m$[j].val = V[j].val, where V[j] is on P_j.}

$\Rightarrow$ LP(F[j])

{By lemma 6.4, F is consistent, so for all j such that F[j].LP, the local predicate was true on F. The global state G is F.}

2) SP(s.F,j)

$\Rightarrow$ $\exists t$: t.V[j].val $\leq$ F[j].val $\wedge$ SP(t.V,j)

{P_j must have been suspected before or at the time of F[j].val.} 1) and 2) lead

$\Rightarrow$ $j \in SP_{\text{F}}$

{G is F.}

to the result since F is the same for both.

We next demonstrate that if *detected* is issued that

$$\exists G : |SC_G| \leq s_c \wedge \bigwedge_{i,j,i\neq j} [CP_{ij}(G) \vee SC(G,(i,j))]$$

The algorithm must satisfy the 'if' condition at line 23 to issue *detected*. This condition is:
check_channel_pred($\mathsf{Msgs\_in\_channels}$)= 0

$\Rightarrow$ $\forall i, j$,m,$\exists k, l$: [channel predicate TRUE for
$\{$m | m $\in$ $\mathsf{Msgs\_in\_channels}$ $\wedge$ m.source = i $\wedge$ m.dest=$j\}$ $\vee$
SC(F,(i,j))
$\{$By definition of function in figure 13.$\}$

$\Rightarrow$ $\forall i, j$: $[CP_{ij}$(F[i],F[j]) $\vee\mathsf{SC}$(F, (i, j))
$\{$Because $\mathsf{Msgs\_in\_channels}$ only includes messages which cross F (by Lemma 6.6), and because channel (i, j) was suspected before F.$\}$

$\Rightarrow$ $\bigwedge_{i,j}[CP_{ij}(\text{F}[i], \text{F}[j]) \vee \mathsf{SC}(\text{F}, (i, j))$
$\{$Rewriting.$\}$

Finally we show that $\exists G : f(SSP_G, SSC_G, SSM_G)$. Again, the algorithm must satisfy the 'if' condition at line 23 to issue *detected*. This condition is:
check_channel_pred($\mathsf{Msgs\_in\_channels}$)= 0

$\Rightarrow$ $\mathsf{f}(G)$

$\Rightarrow$ $f(SSP_G, SSC_G, SSM_G)$

We note that the F of local predicate detection is the same as that for check_channel_pred(), and therefore, the result follows.

$\Leftarrow$:

Now assume:

$$\exists G : \text{local\_conditions}(G) \wedge \text{channel\_conditions}(G) \wedge \text{suspicion\_conditions}(G)$$

We show that the algorithm will issue *detected*.

$\exists G : \text{local\_conditions}(G) \wedge \text{channel\_conditions}(G) \wedge \text{suspicion\_conditions}(G)$

$\Rightarrow$ $\exists s, k : s$ on P_k: s.F = G
$\{$By lemmas 6.5 and 6.6, F will continue advancing if the global predicate is not satisfied,
and all consistent global states preceding F do not satisfy this predicate.$\}$

We show that the four disjuncts from process_conditions(G) and channel_conditions(G) imply conditions in the algorithm.

i) $\exists j : LP_j$(G)

$\Rightarrow$ $\exists t$ on P_j: t.State_list$_{top}$[j] = G[j]
$\{$Again, by lemmas 6.5 and 6.6, F at all j cannot advance past G and must eventually reach G. This implies that there must be a t such that t.State_list$_{top}$[j] = G[j] if LP_j(G).$\}$

$\Rightarrow$ t.F[j].LP
$\{$F is G at t, t.State_list$_{top}$[j] = G[j] = t.F[j] and t.State_list$_{top}$[j].LP must be true (by lines 3-4). Therefore, by line 19, t.F[j].LP.$\}$

ii) $\exists j : j \in SP_G$

$\Rightarrow$ $\exists t : t \rightarrow G[i] : \mathsf{SP}(t$.V,i)
$\{$By definition 6.3.$\}$

$\Rightarrow$ $\mathsf{SP}(s$.F,j)
$\{$Because $s = G[i]$ and s.F = G.$\}$

iii) $\exists i, j : CP_{ij}(G)$

$\Rightarrow$ 'if' at line C4 returns FALSE

{By lemma 6.7, Msgs_in_channels contains all messages which cross the global state defined by s.F, and s.F is G. Line C3 creates Cur_channel with messages in channel (i, j) which are not suspected. By our definition of $CP_{ij}()$, suspected messages are considered to have never been sent.}

iv) $SC(G, (i, j))$

$\Rightarrow$ SC(s.F,(i, j))

{By definition 6.3 of $SC(G, (i, j))$, and because s.F = G.}

From i,ii,iii, and iv):

v) (for all j: $LP_j(G) \vee j \in SP_G$)

$\Rightarrow$ for all j: First_state[j].LP $\vee$ SP(First_state,j))

vi) (for all $i, j, i \neq j : CP_{ij}(G) \vee SC(G, (i, j))$) and $f(SSP_G, SSC_G, SSM_G)$

$\Rightarrow$ for all $i, j, i \neq j : CP_{ij}(G) \vee SC(G, (i, j))$) and f($G$)

$\Rightarrow$ check_channel_pred(Msgs_in_channels) = 0

The results of v) and vi)

$\Rightarrow$ *detected* issued

{By lines 21 and 23.}

$\square$

6.3 Complexity and Latency

Recall that N and m are the number of processes in the system and the number of messages received at one process during the monitoring period respectively. VC and First_state are both $O(N)$. Assume that First_state lags VC in the worst case l_1 states on each process and that there are l_2 messages in the worst case from each process sent before but received after First_state. Let $m' = l_1 + l_2$. The size of Msg_list is $O(m'N)$. Therefore, the size of information added to every application message is $O(m'N)$. In most cases, $m' \ll m$, since processes exchange information and First_state advances. We will show how to reduce the total overhead to $O(N)$ for certain channel predicates in the next section. Also note that if channel predicates are not being detected, the algorithm does not need to send Msg_list and this reduces to $O(N)$. In the worst case, each process has m states in which its local predicate is true before a global state is reached in which *detected* is issued. Then, every process performs $O(mN + N) = O(mN)$ comparisons and $O((m+m)N) = O(mN)$ updates before detection.

Our algorithm has latency in detection due to the use of application messages rather than control messages. This is directly related to the frequency and spread of application messages sent. For example, if a process remains isolated in the application for some time by not sending any messages, it can delay detection of the predicate. This will also occur if sets of processes are isolated from other sets. We give an approach next for reducing this latency.

To reduce latency, the algorithm can pass information during a suspicion. Here, any process that begins suspecting, broadcasts its predicate information (First_state, and Msg_list) to all other processes that are monitoring. Any process receiving this message also then immediately broadcasts. To avoid multiple broadcasts, it may be desirable to have processes not broadcast if they suspect a process for which a suspicion tag has already been received. This approach has the following benefits: 1) it allows for the evaluation of the predicate at the current global state whenever an entity is suspected of failure, which is useful for the types of predicates this algorithm can detect, 2) since failures and suspicions can be assumed to be rare, the broadcasts will occur very infrequently.

6.4 Reducing overhead

It should be noted that although we have described the algorithm across the N processes in the system, it can also be used across a subset of these N processes. In this case, all vectors such as First_state and VC would be smaller. Any process not part of the subset, however, would have to relay the predicate information through it to processes which were. Conjunctive local predicate detection across subsets of N is given by [11] in their paper on detecting conjunctions of process predicates in an error free environment. Next, we present optimizations for certain channel predicates which decrease the information added to messages to $O(N)$ or $O(1)$.

The first optimizations for channel predicates are for those which are based on an aggregate numbers of messages. For example, "no token in channels" or "empty channels". Consider the predicate "empty channels." We can decrease the overhead for channel predicates to $O(N)$. It is enough to know that all messages sent into a channel have been received. Therefore, Msg_list is now two arrays, each of dimension N. The first array is the number of messages sent from each process before First_state and the second is the number of messages received from each process before First_state. Each process also maintains a list of messages it has received in the same format as the original Msg_list. It uses this to set its entry in the receive message array. Note that since First_state[i].LP is only set to TRUE by P_i, when P_i does so it can accurately update its entries in the send and receive message lists. P_i can also check all channel predicates if for all processes in First_state, the local predicate is true or is suspected. The function check_channel_preds() would be modified to perform a check on all channels together, rather than singly.

For detection of no token in channel, we can use Msg_list as it is, and instead of adding all messages to the list, only add token messages to the list when sent. Since for many algorithms one token is employed, Msg_list becomes $O(1)$.

6.5 Conclusions

The algorithm we have presented detects conjunctions of process and channel predicates in spite of failures. One feature of our approach is that control messages are not required. Previous work has not considered detection of such general predicates in a faulty environment or has required many control messages.

References

[1] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the Isis Toolkit*, various chapters. 1993.

[2] T. Chandra, V. Hadzilacos, S. Tuoeg, "The Weakest Failure Detector for Solving Consensus," *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Systems*, Aug. 1992, pp. 147-158.

[3] T. Chandra, S. Toueg, "Unreliable Failure Detectors for Asynchronous Systems," *Proceedings of the 10th ACM Symposium on Principles of Distributed Systems*, Aug. 1991, pp. 325-340.

[4] K. Chandy, and L. Lamport, "Distributed Snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, no. 1, February 1985, pp. 63-75.

[5] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, pp. 163-173.

[6] M. Fischer, N. Lynch, M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no.2, 1985, pp.374-382.

[7] V. Garg, C. Chase, J. Mitchell, R. Kilgore, "Detecting conjunctive Channel Predicates in a Distributed Programming Environment," 28th HICSS, January 1995, pp. 232-241.

[8] V. Garg, C. Chase, "Distributed Algorithms for Detecting conjunctive Predicates," Technical report TR-PDS-1994-003, University of Texas at Austin, June, 1994.

[9] V. Garg, C. Chase, "Distributed Algorithms for Detecting conjunctive Predicates," Proceedings of the Int'l Conference on Distributed Computing Systems, June, 1995, pp. 423-430.

[10] V. K. Garg, and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, # 3, Mar 1994, pp. 299-307.

[11] M. Hurfin, M. Mizuno, M. Raynal, M. Singhal, "Efficient Distributed Detection of Conjunctions of Local Predicates," IRISA technical report, PI-967, 1995.

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558-565.

[13] P. Li, B. McMillin, "Fault-Tolerant Distributed Deadlock Detection/Resolution," *Proceedings of the 17th International COMPSAC*, November, 1993, pp. 224-230.

[14] Y. Manabe, and M. Imase, "Global Conditions in Debugging Distributed Programs," *Journal of Parallel and Distributed Computing*, Vol. 15, 1992, pp. 62-69.

[15] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, 1989, pp. 215-226.

[16] J. R. Mitchell, V.K. Garg, "Distributed predicate detection in a faulty environment," ECE dept, University of Texas, TR-PDS-1997-001.

[17] A. Ricciardi, K. Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments," *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Systems*, Aug. 1992, pp. 147-158.

[18] Y.-C. Tseng. "Detecting termination by weight-throwing in a faulty distributed system," *Journal of Parallel and Distributed Computing*, 25:7–15, 1995.

[19] S. Venkatesan, "Reliable protocols for distributed termination detection," IEEE Transactions on Reliability, Vol. 38, No.1, April 1989, pp 103–110.