Deriving Distributed Algorithms from a General Predicate Detector

J. Roger Mitchell

Vijay K. Garg

TR TR-PDS-1995-001 January 1995

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Deriving Distributed Algorithms from a General Predicate Detector

J. Roger Mitchell* Vijay K. Garg†

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

August 8, 1995

Abstract

Many problems or applications in distributed systems require the detection of conditions across the entire system. As an illustration, monitoring the status of an application often requires detection of termination, and to use virtual time requires the periodic calculation of the global virtual time. Often, to perform this detection an algorithm is devised which meets this need. A better approach might be to have a standard algorithm which meets many needs and can be applied to many detection problems. The Generalized Conjunctive Predicate (GCP) detector offers a method to derive detection algorithms for many problems based on optimizing the base algorithm.

1 Introduction

In distributed systems it is often desirable to detect a condition that depends on the global state, that is the states of the processes that make up the system. Since the condition depends on the global state, none of the processes can independently determine if the condition is true. Because of this fact, detecting global conditions has been a problem in distributed systems that arises in designing, testing, and debugging these systems. There have been many approaches to detecting global conditions, or predicates, as in [1, 2, 5, 9, 11, 13, 15, 17, 18, 22].

Some of the approaches cited above can detect conditions within channels as well as at the processes of the system. However, these approaches either cannot perform detection during the computation or cannot detect a very broad range of predicates. The GCP algorithm, introduced in [6, 7], can be used to detect the conjunction of local and channel predicates during the distributed computation. For efficient detection, the class of channel predicates encompassed by this algorithm is called *monotonic* and includes many desirable predicates. The approach used by the GCP method is to collect states for which the local process predicates are true, and then attempt to find a group of these states which are part of a consistent cut. Channel predicates are checked following this. This is a different approach to detection from that used in snapshot algorithms [1, 14]. A snapshot

*supported in part by MCD and Virginia & Ernest Cockrell fellowships
†supported in part by the NSF Grant CCR-9110605, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

algorithm attempts to find consistent cuts and then determine if the cuts satisfy the predicate. Although snapshot algorithms run during the computation, because of their approach, they are suitable only for stable predicates. The GCP algorithm, however, can detect stable or unstable predicates. See [20] for a survey of stable and unstable predicate detection. The method used in GCP will not only detect unstable predicates, but will also find the earliest occurrence of such a predicate, that is, the first consistent cut which satisfies the predicate. Snapshot algorithms cannot, in general, find the first occurrence of a predicate. How close the snapshot cut is to the first occurrence of a predicate is a function of the frequency with which the algorithm is run.

We will show that the GCP algorithm can be thought of as a generic algorithm from which other algorithms can be derived. This is done by optimizing GCP according to the properties of detection problems. We will define these properties and the optimizations for the GCP algorithm that stem from them. Then we will demonstrate how to derive algorithms using these optimizations by applying this approach to three specific detection problems, namely: termination detection, detection of channel buffer overflow, and global virtual time progress.

2 Notation

In this paper, we use notation of states, events, predicates, etc. to illustrate the concepts of GCP. A lower case letter in a relationship is used to represent a process or channel state and an upper case letter represents a global state, that is, one local state from every process and the corresponding channel states between these processes. For example, s, t, u are local process states, while G, H are global states or cuts. We use $LP(x), CP(y)$, and $GP(Z)$ to indicate that a local predicate is true at state x, a channel predicate is true for channel state y, and a global predicate is true for global state Z. For channel predicates, the channel state is a sequence of messages, belonging to any set Σ^* (a sequence of messages). Therefore, $CP : \Sigma^* \rightarrow \{true, false\}$, that is, CP acts as a function which maps a sequence of messages, the channel state, to *true* or *false*.

The ordering of local events are represented using $\prec$, $\prec_{im}$, and $\rightarrow$. If $s \prec t$ then s occurred before t on the same process. If $s \prec_{im} t$ then s occurred immediately before t on the same process. The action $s \rightarrow t$ holds if and only if one of the following holds: 1) $s \prec t$, 2) the action following s is the send of a message and the action preceding t is the reception of that message, 3) there exists a state u such that $s \rightarrow u$ and $u \rightarrow t$. Note that when $\prec$ is used as a relation for cuts, as in $G \prec H$, this means that each local state in G is equal to or $\prec$ the state on the same process in H, and that at least one local state in $G \prec$ a state in H.

Often, m_i is used to represent message i. The send and receive events are represented by $m_i.send$ and $m_i.receive$.

2.1 Overview of token-GCP algorithm

The GCP algorithm was originally presented in [6] which contains a centralized version of the algorithm. The GCP algorithm in this paper is the token based algorithm found in [7]. Token-GCP uses a token to determine when all local and channel predicates are true concurrently. The token contains a possible global cut in which the desired predicates could be true. This global cut contains only local states for which the local predicates are true. If the global cut is also consistent then all local predicates are true concurrently. If not, the token is forwarded to any process which violates the consistency. Once a consistent cut is found, the channel predicates are checked. If true, then the GCP is detected.

The channel predicates detected are within the class called *monotonic*, and further divided into *send* and *receive monotonic* predicates. A *send-monotonic* channel predicate is one which, if false, cannot be made true by sending more messages on a channel. The predicate "the channel is empty" is an example of a send-monotonic predicate, because sending more messages cannot make this

predicate true. A *receive-monotonic* channel predicate cannot be made true by receiving more messages from the channel.

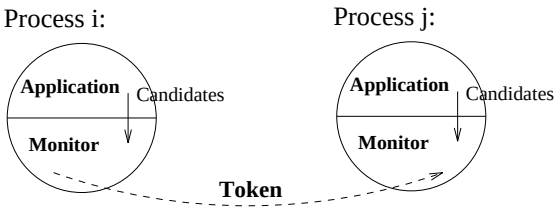

Figure 1: The GCP detector.

Additions to Application Process (A_i):

```
Initialize_clocks_and_vars(i);                                  GCP1
Before send of message m to A_j do
    Add_send_message_in_order(i, m);                            GCP2
Upon receive of message m from A_j do
    Add_receive_message_in_order(i, m);                         GCP3
Upon local_predicate_true() do {                                GCP4
    send (vector clock, send list, receive list) to
        M_i as candidate                                        GCP5
    clear_send_rcv_vars();                                      GCP6
    }
```

Monitor Process (M_i):

```
Upon receive of token do {
    do {
        receive candidate from A_i
        Update_send_rcv_of_token(i);                            GCP7
        } until candidate.vclock[i] > token.vclock[i]           GCP8
    If ∃j : j ≠ i: token.vclock[j] < candidate.vclock[j]
        then send_token(j); /* Send to M_j. */                  GCP9
    Else if channel_predicates_true(i)                          GCP10
        then GCP = true.                                        GCP11
    }
```

Figure 2: Application and monitor code at process i

The implementation of GCP is divided into application and monitor portions at each process. This is shown in figure 1. The application process checks for local predicates and the monitor receives the token and checks for a consistent cut and then channel predicates. Figure 2 gives a high level overview showing the application and monitor code. Note that the monitor code is only activated when the token has been received (there is only one token in GCP). Statement GCP1 initializes the algorithm and GCP2-3 add a sent or received message to message lists for the application. GCP4 detects when the local predicate is true since the last send or receive. When this occurs the application process sends the message list, called a candidate, to the monitor, M_i (GCP5). The message lists are then cleared(GCP6). The monitor waits to receive the GCP token and when it does, begins receiving candidates from the application process. GCP7 collects information from the

application about the channels until a candidate is received with a clock later than that of the token (GCP8). GCP9 checks the token to see if the new cut is inconsistent because of information from another process. This is done by checking for a token vector clock that is less than the received candidate clock. If so, then the token is forwarded to the monitor of that process. Otherwise, GCP10 evaluates channel predicates associated with process i. If all channel predicates are true, the predicate has been detected and GCP is set to true by GCP11.

2.2 Properties of GCP

Many of GCP's interesting uses are characterized by detection of channel predicates. What follows is a list of properties inherent to different distributed programming problems for which the GCP algorithm is suited.

- Channel predicates can be *message content independent*. Those predicates which are independent of message content include those that are based on number or order of messages in a channel.

  The property message content independent is equivalent to:

  $$CP(s) \ \wedge \ (\ |s| = |t| \ ) \ \Rightarrow CP(t) \quad \text{(R1)}$$

  where CP is the channel predicate and s and t are channel states. So, if (R1) holds, then if the predicate is true at s and the number of messages at t equals that at s, then the predicate is true at t. Termination, deadlock, and buffer overflow are problems with channel predicates that are content independent.

- Channel predicates can be *message ordering independent*. This property is equivalent to:

  $$CP(s) \ \wedge \ (s \sim t) \ \Rightarrow CP(t) \qquad \text{(R2)}$$

  where $\sim$ is the relation "is a permutation of." If (R2) holds, then if a predicate is true at a state s, and the messages at s are a permutation of messages at another state, t, then the predicate is true in t as well. Empty channels, and greater than k messages in a channel are predicates which are ordering independent. It should be noted that (R2) represents a weakening of (R1) and therefore, (R1) $\Rightarrow$ (R2). An example of a predicate which satisfies (R2) but not (R1) is "marker in channel".

- Global predicates are either *stable* or *unstable*. These predicates are made up of local and channel predicates. The global predicate, $GP()$, can be written in terms of cuts G and H. Stating that $GP()$ is stable is equivalent to establishing:

  $$GP(H) \ \wedge \ (H \prec G) \ \Rightarrow GP(G) \quad \text{(R3)}$$

- Predicates can be *cut consistency independent*. These predicates can become true in a cut with messages received but not sent. If $G[i]$ represents the state of process i at the cut G, then a consistent cut can be defined as:

  $\text{consistent_cut(G)} \equiv \forall i, j : G[i] \not\rightarrow G[j]$

  Using G and H for cuts, and then consistency independence is equivalent to:

  $$\exists H : CP(H) \ \wedge LP(H) \iff \exists G : CP(G) \wedge \ LP(G) \wedge \text{consistent_cut(G)} \qquad \text{(R4)}$$

  (R4) states that a channel and local predicate are true for some cut H, if and only if there exists a consistent cut, G, for which both predicates are true. Termination detection and calculation of global virtual time are consistency independent problems.

- Channel predicates can be *receive monotonic* or *send monotonic*. For example, if a channel predicate is receive monotonic and is true at state s, and the messages in the channel at state s are a prefix of the set of messages at t, then the predicate is also true in t. This can be stated as:

  $$C(s) \ \wedge \ s \text{ prefix of } t \Rightarrow C(t) \qquad \text{(R5)}$$

  An example of receive monotonicity is the channel predicate "$> k$ messages in channel." Send monotonicity can be written as:

  $$C(s) \ \wedge \ t \text{ suffix of } s \Rightarrow C(t) \qquad \text{(R6)}$$

  Following the example for receive monotonicity, "$< k$ messages in channel" is a send monotonic channel predicate. As mentioned earlier, the GCP algorithm can only detect monotonic channel predicates.

2.3 Classifying detection problems using GCP properties

By knowing which properties a detection problem has, their presence can be exploited for optimizing the GCP algorithm. For example, knowing that a problem is message content independent and that this property implies ordering independence, the GCP algorithm can be optimized so that instead of saving messages sent and received, only the count of messages sent and received is maintained. The table in figure 3 gives the list of properties and what optimizations can be performed given their presence.

Property	**Optimization**	**Change to GCP**
Message content independent	Save only number of messages	GCP2 = **Send:** Increment_send_count() GCP3 = **Receive:** Increment_receive_count()
Message ordering independent	Send and receive lists not strictly ordered	GCP2 = Concatenate_send_messages GCP3 = Concatenate_receive_messages
Global predicate stable	Do not store previous cuts	GCP7 = receive last candidate
Cut consistency independent	Consistent cuts not required	GCP8 = } until no more **candidates** Delete GCP9 GCP10 = **If** channel_predicates_true()
Strictly prefix closed	Check only for receive monotonicity	check for F_s removed from channel_predicates_true() monitor process does not evaluate to F_s
Strictly suffix closed	Check only for send monotonicity	check for F_r removed from channel_predicates_true() monitor process does not evaluate to F_r

Figure 3: Optimizations for properties of predicates

The standard GCP algorithm saves messages in any channel to lists by concatenating. For detection problems which are message ordering independent, any method of appending to the messages lists can be used. Generally, a method will be chosen which is most efficient for the condition to be detected.

A look at the table shows that for prefix and suffix closure, the change to the algorithm is to remove values to which the function channel_predicates_true() can evaluate to. This function determines whether channel predicates are true. Any that are not are set to F_s or F_r indicating that the predicate is false and is currently send-monotonic or receive-monotonic respectively.[1] A

[1] It should be noted that some predicates are *dynamically monotonic*, that is, these predicates can be send or receive monotonic based upon their evaluation. For example, the predicate "the channel contains six messages" is send-monotonic if there are greater than six messages in the channel and receive-monotonic if there are less than six messages.

channel predicate which is strictly prefix or suffix closed can only be send or receive monotonic, but not both, hence the optimization in the table.

In this paper, the three problems of termination detection, global virtual time progress, and channel buffer overflow are implemented. For the first, the inherent properties are message content, and therefore ordering, independence, and cut consistency independence of channel predicates. The second problem is not message content independent, but *is* message ordering independent, as well as cut consistency independent. Detection of channel buffer overflow is message content independent, and has prefix closure for the channel predicates.

Other detection problems for which the GCP algorithm is suitable include system wide deadlock, loss of token, too many marker colors, violation of FIFO, and channel underutilization.

Knowing the properties for a given problem will allow us to optimize the GCP algorithm in each case.

3 GCP for Termination Detection

Termination detection is a standard problem in distributed computing. The problem is to detect when there are no more tasks that need to be performed for the computation in question. A bit more precisely, all processes are in a state in which they will not execute any more commands, including sending or receiving messages. The state that the system reaches when this is true is called termination. Since an idle process is one which cannot cause itself to become non-idle:

> A process, if idle, remains idle until it receives a message. (T1)

Termination detection, then, usually involves checking that the following holds:

> All processes are idle and there are no messages in the channels. (T2)

It is not difficult to see that if (T1) holds and (T2) becomes true then the system is terminated and will remain terminated until some message is sent from outside the system. Because of this, termination is a stable property and satisfies (R3). Note, however, that the local predicate, "process is idle" and the channel predicate, "channel is empty," are both unstable. The channel predicate is also content and order independent, and will be shown later to be cut consistency independent as well. Obviously, detection of empty channels only requires knowledge of the number of messages in the channel, without concern for their order. So, (R1) is satisfied which implies (R2) as well.

To detect (T2) requires detecting that all processes are idle and in addition that all channels contain zero messages, that is, they are empty. Detecting that a process is idle can be done by one process, but detecting that a channel is empty requires that the processes at both ends of the channel get involved. Since the local predicates are "process is idle" and are easy to detect, we will concentrate on methods for detecting "channels are empty".

Let s_{ij} and r_{ij} be the number of sends from i to j and the number of receives from i to j respectively. Since a channel is empty when the number of sends equals receives on that channel,

$$\bigwedge_{i,j:i\neq j} s_{ij} = r_{ij} \iff \text{all channels empty} \tag{1}$$

So, to use GCP for termination detection requires that the local and channel predicates be defined as:

> ith process's local predicate $\equiv$ process i idle
>
> Channel i, j predicate $\equiv s_{ij} = r_{ij}$

Note that consistency of the states where s_{ij} and r_{ij} are measured is not a concern. Because s_{ij} and r_{ij} are strictly increasing, equation 1 is true independent of whether or not these two variables

are measured when processes i and j are concurrent. In fact, when s_{ij} does equal r_{ij}, the two will be concurrent. The GCP algorithm, though, does perform predicate detection using consistent cuts. First we will look at applying this to GCP using consistent cuts and then optimize GCP so that consistent cuts are not used.

3.1 Termination detection with consistent cuts

If we have a consistent cut then we know the following:

$$\forall i, j : s_{ij} \geq r_{ij} \tag{2}$$

This is because in a consistent cut a message cannot be received that has not been sent. Because of this, equation 1 is used to obtain:

$$(\sum_{ij:i\neq j} s_{ij} = \sum_{ij:i\neq j} r_{ij}) \wedge (\forall i, j : s_{ij} \geq r_{ij}) \Rightarrow \text{all channels empty} \tag{3}$$

This equation is true because if for all $i, j : s_{ij} \geq r_{ij}$, and the sum of all sends equals the sum of all receives, then there cannot be an $s_{ij} > r_{ij}$. Rather, if the sums equal for a consistent cut, then for all $i, j : s_{ij} = r_{ij}$.

So, the second term of equation 3 can be replaced by the requirement that the cut be consistent:

$$(\sum_{ij:i\neq j} s_{ij} = \sum_{ij:i\neq j} r_{ij}) \wedge \text{consistent cut} \Rightarrow \text{all channels empty} \tag{4}$$

Therefore, the GCP algorithm can detect empty channels by keeping track of the number of sends and receives as the token is passed throughout the system. So, the channel predicates are message content and ordering independent. Using the optimization for this property given in figure 3 and the channel predicate as defined in equation 4, a new algorithm for this problem can be derived. Figure 4 gives the new algorithm for the monitor and application at each process. Here, the GCP application code at each process only keeps track of the *number* of sends and receives and gives this information to the monitor. The monitor, then, keeps a running total of all sends and receives of the system. If a consistent cut is found and $S = R$ in the token, then termination has been detected. This is only somewhat more efficient than checking that the number of sends and receives are equal on each channel. Next, an optimization will be studied in which consistent cuts are not used.

3.2 Termination detection without consistent cuts

Since we know equation 1 holds for inconsistent cuts, the detection of an equal number of sends and receives for a channel can be done without regard for consistency. The GCP optimization would be removal of the check for consistency. There is, however, another improvement that can be made with this approach and it involves an approach first presented by Mattern in his *vector counting* algorithm [16]. Although shown by Mattern, for completeness a proof is given for the following theorem to validate the use of inconsistent cuts. Let $G[i]$ represent the state of process i at the cut, and:

$G[i].r$ = number of messages received by process i at the cut.
$G[j].s[i]$ = number of messages sent from process j to process i.
$G[j].R[j]$ = history of messages received at i from j at the cut.
$G[i].S[j]$ = history of messages sent from j to i at the cut.
m_i = message i.
$m_i.receive$ = the receive event associated with message i.

Additions to Application Process (A_i):

```
Initialize_clocks_and_vars(i);
Before send of message m to A_j do
    Increment_send_count(i);              /* send_count++ */
Upon receive of message m from A_j do
    Increment_receive_count(i);           /* receive_count++ */
Upon local_predicate_true() do {          /* Is application process idle? */
    send (vector clock, send_count, receive_count) to
        M_i as candidate
    clear_send_rcv_vars();                /* clear receive_count and send_count. */
    }
```

Monitor Process (M_i):

```
Upon receive of token do {
    do {
        receive candidate from A_i
        Update_send_rcv_of_token();
        } until candidate.vclock[i] > token.vclock[i]
    If ∃j : j ≠ i: token.vclock[j] < candidate.vclock[j]
        then send token to M_j.
    Else if channel_predicates_true(i)
        then GCP = true.
    }
```

Figure 4: GCP at process i optimized for termination detection with consistent cuts

Theorem 3.1 *Given (T1) and* $\forall i : G[i] = idle$*:*

$$\forall i, j : G[i].r = \sum_j G[j].s[i] \iff \forall i, j : G[i].R[j] = G[j].S[i].$$

That is, for all i, the number of messages recorded as received by i at the cut equals the sum of sends to i by all other processes, if and only if for all channels, the set of messages sent is equivalent to the set of messages received (or equivalently, the channels are empty).

Proof: ($\Leftarrow$): this is obviously true.
($\Rightarrow$) will be proved by contradiction by assuming $G[i].r = \sum_j G[j].s[i]$ and $G[i].R[j] \neq G[j].S[i]$ for some i, j. Let N = number of messages received but not sent that are in a chain. See 5 for illustration. This will be called a chain of inconsistent messages. For $N > 1$ the following is true:

$$\bigwedge_{j=1}^{N} (m_j.receive \prec G[m_j.receive.p]) \wedge \bigwedge_{j=1}^{N-1} (m_j.receive \rightarrow m_{j+1}.receive)$$

where $m_j.receive.p$ is the process which received message j.

Case 1: Let $N = 0$, then $\exists m_0 : (m_0 \in G[j].S[i] - G[i].R[j]) \vee (m_0 \in G[j].R[i] - G[i].S[j])$
(because $G[j].S[i] \neq G[i].R[j]$).
$\Rightarrow$: $\exists m_a, k : m_a \in G[i].R[k] - G[k].S[i]$, because $\forall k : G[i].r = \sum_k G[k].S[i]$ and m_0 exists. That is, a receive of a messsage, m_a, must have been recorded, but not the corresponding send.

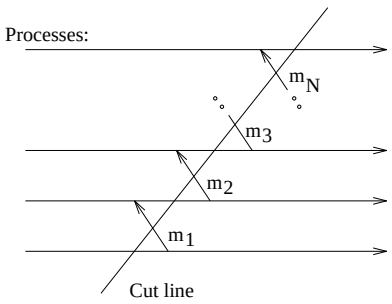

Figure 5: A chain of inconsistent messages

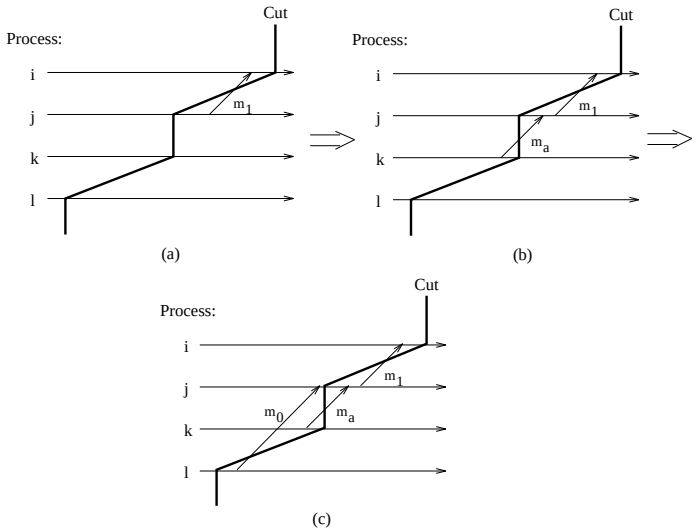

Figure 6: An inconsistent cut in detecting termination.

$\Rightarrow$: $N > 0$ and, therefore, that Case 2 exists.

Case 2: Let $N = n : n > 0$, then $\exists m_1 : m_1 \in G[i].R[j] - G[j].S[i]$ where m_1 is the tail of a chain. This message is shown in figure 6a.
$\Rightarrow$: $\exists m_a, k : m_a \in G[k].S[j] - G[j].R[k]$, because process j must receive a message after the cut to become active and send m_1, according to rule (T1). That is, there must exist a message that was sent before and received after the cut. The receive of this message must causally precede the send of m_1, because of (T1). Figure 6b shows a possible m_a.
$\Rightarrow$: $\exists m_0, l : l \neq i : m_0 \in G[j].R[l] - G[l].S[j]$, because $\forall l : G[j].r = \sum_l G[l].s[j]$ and because if sent from process i a message loop would be created with each message "happening-before" all others, an impossibility. Figure 6a-c illustrates the implications:
m_1 exists $\Rightarrow$ m_a exists $\Rightarrow$ m_0 exists $\Rightarrow$ $m_0.receive \rightarrow m_a.receive$.
$\Rightarrow$: $m_0.receive \rightarrow m_1.send$, therefore m_1 is not the tail of the chain of inconsistent messages and $N = n + 1$.
$\Rightarrow$: $N > n$, a contradiction.
$\Rightarrow$: $\forall i, j : G[i].R[j] = G[j].S[i]$
$\Rightarrow$: consistent cut $\square$

So, theorem 2.1 says that if $\forall i : G[i].r = \sum_j G[j].s[i]$ is detected for any cut, then the channels are empty, $\forall i, j : G[i].R[j] = G[j].S[i]$, and the cut is consistent. Therefore, the requirement for cut consistency independence, (R4), holds for termination detection, and in this case, $G = H$. In addition, because it has been shown that if $\forall i : G[i].r = \sum_j G[j].s[i]$ is true, then all channels are empty and we know termination is stable.

Now, termination detection can be implemented with the properties: message content, message ordering, and cut consistency independence. Using the optimization table in figure 3 to implement the changes associated with these properties gives the algorithm in figure 7. Note that the channel predicate on any channel from process i to process j is $R_j = \sum_k S_{kj}$, as described above. The variables r and s_j are maintained by A_i, while R_j and S_{ij} are contained in the token. See the appendix for the subroutines unique to this implementation.

4 GCP for detection of channel buffer overflow

To monitor channel buffers the problem is to detect when too many messages have been sent without being received. The reason would be to detect a sender that has violated the limit on the maximum number of messages to send before an acknowledge from the receiver. The case given here would be concerned with one channel in particular, or all channels individually. To use GCP fully, a detection of all or a subset of channels with too many messages could be used to flag saturation of the network.

Note that this predicate is message content and ordering independent. It is also prefix closed because the channel predicate "# messages $> k$" is receive monotonic. There is not a local predicate for this detection, therefore the local predicates used here would be set to TRUE. To use GCP requires that the local and channel predicates be defined as:

ith process's local predicate $\equiv$ TRUE

Channel i, j predicate $\equiv s_{ij} - r_{ij} > k$

where s_{ij} and r_{ij} are send and receive counts. Unlike termination detection, the channel predicate does require a consistent cut. For example, suppose $k = 3$. Figure 8 shows an inconsistent cut which would incorrectly detect that greater than three messages had been sent between acknowledgements. From this example it is obvious that (R4) is not satisfied.

Because detection of channel buffer overflow is message content and ordering independent, as well as prefix closed, the optimizations for these can be implemented in the GCP algorithm. Because the

Additions to Application Process (A_i):

```
Initialize_clocks_and_vars(i);
Before send of message m to A_j do
    Increment_send_count(i, j);          /* send_count_j ++ */
Upon receive of message m from A_j do
    Increment_receive_count(i, j);       /* receive_count++ */
Upon local_predicate_true() do {         /* Is application process idle? */
    send (vector clock, all send_count_j, receive_count) to
        M_i as candidate
    clear_send_rcv_vars();               /* clear receive_count and all send_count_j . */
    }
```

Monitor Process (M_i):

```
Upon receive of token do {
    do {
        receive candidate from A_i
        Update_send_rcv_of_token(i);
        } until no more candidates;
    If channel_predicates_true(i) /* If NOT true, token will be sent to another monitor. */
        then GCP = true.
    }
```

Figure 7: GCP at process i optimized for termination detection without consistent cuts.

local predicate is TRUE, this check is eliminated in the GCP application code so that a **candidate** is sent to the monitor after every send and receive. Implementing these changes gives the algorithm in figure 9. The subroutines unique to this algorithm are given in the appendix.

5 Detection of global virtual time $\geq k$

For global virtual time, GVT, processes maintain their own local virtual time, LVT, which indicates how far each has progressed in their program. At any time, however, a process might receive a message from another process which has not progressed as far as itself. In this case, the message will contain a timestamp which is earlier than the local virtual time. When this happens the local process receiving this message must "rollback" to the timestamp of the message. This requires reexecuting events, including message receives and sends, possibly even nullifying a send that has already taken

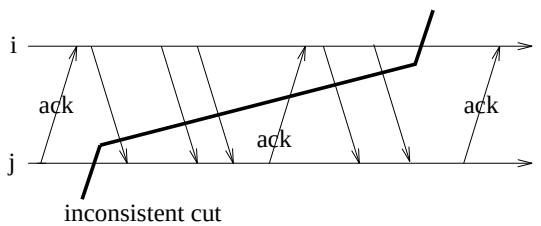

Figure 8: An example of an inconsistent cut which would incorrectly detect greater than 3 messages in a channel.

Additions to Application Process (A_i):

```
Initialize_clocks_and_vars(i);
Before send of message m to A_j do
    Increment_send_count(i, j);          /* send_count_j++ */
Upon receive of message m from A_j do
    Increment_receive_count(i, j);       /* receive_count_i++ */
Upon local_predicate_true() do {         /* Local predicate always true; check for firstflag. */
    send (vector clock, all send_count_j, all receive_count_j) to
        M_i as candidate
    clear_send_rcv_vars();               /* clear all receive_count_j and send_count_j. */
    }
```

Monitor Process (M_i):

```
Upon receive of token do {
    do {
        receive candidate from A_i
        Update_send_rcv_of_token(i);
        } until candidate.vclock[i] > token.vclock[i]
    If ∃j : j ≠ i: token.vclock[j] < candidate.vclock[j]
        then send token to M_j.
    Else if channel_predicates_true(i)
        then GCP = true.
    }
```

Figure 9: GCP at process i for channel capacity usage.

place. Because of this, the process must maintain a history of its execution back to a time at which all processes must have reached, which is the current known estimate of GVT.

Keeping the history of processes as small as possible is important for efficiency. Therefore, it is important to regularly update the GVT known to all processes so that the history saved can be reduced and memory reclaimed. Tomlinson and Garg [23] give a method for updating GVT in which an initiator sets a target virtual time, TVT. Using a TVT allows for regular progress of GVT and spreads the calculation out over time. This approach detects when GVT reaches TVT and then informs all processes. It is apparent that GVT $>$ TVT is a stable property (because GVT never decreases) and that GVT $>$ TVT satisfies property (R3).

GCP can be used to implement a predicate which detects GVT $\geq$ TVT. GVT for any cut of the execution is the minimum of all LVTs and the timestamps of all messages in the channels. Let $msg_j.CVT$ be the channel virtual time of message j, or rather the timestamp of the message. Then, to detect GVT $\geq$ TVT requires:

$$(\forall i : LVT_i \geq TVT) \wedge (\forall j : msg_j.CVT \geq TVT) \tag{5}$$

where msg_j is a message in a channel. So, the predicates can be defined as:

ith process's local predicate $\equiv LVT_i \geq TVT$

Channel i, j predicate $\equiv min\{msg_j.CVT \mid msg_j \in S_{ij} - R_{ij}\} \geq TVT$

Because the timestamp of messages need to be checked, the channel predicate is not content independent, but is message ordering independent. The predicate can also be shown to be cut

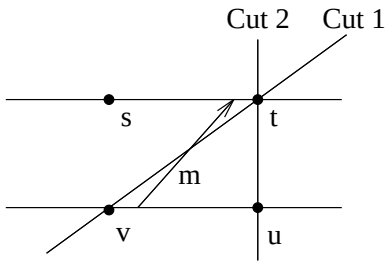

Figure 10: Consistent and inconsistent cuts for two processes

consistency independent. Let GVT_i equal the GVT of an inconsistent cut and GVT_c equal the GVT of a consistent cut.

Theorem 5.1 *If an inconsistent cut precedes a consistent cut in which the only difference is the exclusion of send events then* $GVT_i \leq GVT_c$.

Proof: For a consistent cut and $GVT \geq TVT$ the following holds:

$$\forall i, j : i, j \; processes : G[i].LVT \geq TVT \wedge \forall m : m \in (G[i].send - G[j].receive) : m.CVT \geq TVT$$

An inconsistent cut precedes a consistent cut: Referring to figure 10, cut 2 is consistent and is preceded by cut 1 which is not. Since the inconsistent cut only excludes send events which are in the consistent cut, an inconsistent cut as defined can differ only as shown by the figure.
Now:

$GVT_c = min(\{X\})$ where $\{X\}$ is the set of all LVTs and CVTs in the cut.
From the figure: $v.LVT < u.LVT$ and $m.CVT < u.LVT$.

{Let '+' and '-' be set addition and subtraction, and using the figure.}
$\Rightarrow GVT_i = min(\{X\} + v.LVT + m.CVT - u.LVT)$
{Since u is part of the consistent cut, either $u.LVT = GVT_c$ or $u.LVT \neq GVT_c$:
If $u.LVT = GVT_c$ then GVT_i is the minimum of $v.LVT$ and $m.CVT$.
If $u.LVT \neq GVT_c$, then GVT_i is the minimum of {X}, $v.LVT$, and $m.CVT$}.
$\Rightarrow GVT_i = min(v.LVT, m.CVT) < u.LVT \;\; \vee \;\; GVT_i = min(\{X\} + v.LVT + m.CVT)$.
{$u.LVT = GVT_c$ on lhs}
$\Rightarrow GVT_i = min(v.LVT, m.CVT) < GVT_c \;\; \vee \;\; GVT_i = min(\{X\} + v.LVT + m.CVT)$.
{Simplifying on LHS; on RHS, removing elements from the minimum set can only increase the minimum.}
$\Rightarrow GVT_i < GVT_c \vee GVT_i \leq min(\{X\})$.
{Simplify.}
$\Rightarrow GVT_i \leq GVT_c$. □

Theorem 5.2 *If two cuts, G and H, differing by only one state, have the relationship* $G \prec H$ *then* $GVT_G \leq GVT_H$.

Proof: The reasoning is the same as that of the previous theorem. □

Theorem 5.3 *Theorems 4.1 and 4.2 imply that (R4a) and (R4b) hold, and therefore that GVT calculation is cut consistency independent.*

Additions to Application Process (A_i):

```
Initialize_clocks_and_vars(i);
Before send of message m to A_j do
    Add_send_message_in_order(i, m);
Upon receive of message m from A_j do
    Add_receive_message_in_order(i, m);
Upon local_predicate_true() do {
    send (vector clock, send list, receive list) to
        M_i as candidate
    clear_send_rcv_vars();
    }
```

Monitor Process (M_i):

```
Upon receive of token do {
    do {
        receive candidate from A_i
        Update_send_rcv_of_token(i);
        } until no more candidates;
    If channel_predicates_true(i)
        then GCP = true.
    }
```

Figure 11: GCP at process i for detecting GVT > TVT

Proof: For (R4a) and (R4b):
For (R4a):
Using theorem 4.1, if an inconsistent cut has been found for which equation 5 holds, then by including the send events of all inconsistent messages a consistent cut is created for which 5 holds and is preceded by the inconsistent cut. This is the scenario of theorem 4.1, and with theorem 4.2: $GVT_c \geq GVT_i \geq TVT$. So, local and channel predicates are true in both cuts. H is the inconsistent cut and G the consistent cut in (R4a) and (R4a) is true in this case. The other case is where H in (R4a) is a consistent cut, then (R4a) still holds. Therefore, (R4a) holds in general for calculating GVT.
For (R4b):
From theorem 4.2, a cut which follows a consistent cut in which $GVT > TVT$ is true, has $GVT > TVT$ true as well. Therefore, all cuts which follow the consistent cut will have this property true and it will eventually be detected. □

Now, using the three properties identified with this problem (message content and ordering, and cut consistency independence), and implementing the optimizations as indicated in the table of figure 3 gives the algorithm shown in figure 11. The subroutines unique to this algorithm are given in the appendix.

6 Comparison of optimized GCP to snapshot algorithms

To compare GCP to snapshot algorithms requires that we restrict our discussion to stable conjunctive predicates. This is because snapshot algorithms cannot detect unstable predicates and GCP can only detect conjunctive predicates.

As discussed in [7] the message complexity for the standard GCP algorithm is $O(mn)$ where m

is the number of messages sent by the distributed application and n is the number of processes. Often, as has been demonstrated in this paper, the size of these control messages is greatly reduced because the channel predicate is message content independent. While GCP only needs to be run once, snapshot algorithms generally require multiple runs to detect any predicate. Therefore, the snapshot algorithm of [1] has message complexity of $O(rn^2)$, where r is the number of runs to detect the predicate. The snapshot algorithm of [14] requires only $O(rn)$ messages, although the messages contain entire message history for channels.

For space complexity, the GCP is required to store the S, R, IR sets of messages, therefore the space requirements are at most $O(mn)$. The snapshot algorithms both require $O(m)$ messages, but for the second algorithm m represents a larger amount of messages. Again, for many problems the GCP channel predicates are optimized such that the space complexity is reduced to $O(n)$. This optimization is also mentioned by Lai and Yang in [14] for termination detection.

It should be noted that the GCP algorithm has zero latency between the first cut for which the predicate is true and the cut which detects the predicate. That is, the GCP algorithm always detects the occurence of the predicate for the first cut in which it was true. This is not true for snapshot algorithms, for which the latency is a function of the frequency with which the algorithms are run. Obviously, the more often a snapshot is taken, the smaller the latency will be.

7 Conclusions

We have given properties of distributed detection problems and how to use the presence of these properties to easily optimize the GCP algorithm for distributed systems. Therefore, GCP can be thought of as a base algorithm from which many standard problems can be derived. These optimizations can be performed for unstable as well as stable predicates, which allows for a broader range of problems than in [1, 14]. In addition, the optimized GCP algorithms will detect the first occurrence of any predicate unlike [1, 14]. The message complexity is comparable since the for GCP it is $O(mn)$ and for the two snapshots algorithms it is $O(rn^2)$ or $O(rn)$, where r is the number of runs required for the snapshot.

References

[1] K. Chandy, and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No 1, pp 63-75, February, 1985.

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

[3] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. VanGasteren, "Derivation of a Termination Detection Algorithm for Distributed Computation," *Inf. Proc. Letters,*, Vol. 16, June 1983, pp. 217-219.

[4] C. J. Fidge, "Partial Orders for Parallel Debugging," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* also *SIGPLAN Notices*, Vol. 24. No. 1. January, 1989. pp. 183-194.

[5] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations," *Proceedings of the 23rd Int. Conference on Parallel Processing*, Pennsylvania State University, August 1994.

[6] V. K. Garg, C. Chase, R. Kilgore, J. R. Mitchell,"Detecting Conjunctive Channel Predicates in a Distributed Programming Environment," *Proceedings of the Hawaii International Conference on System Sciences*, Vol II, January 1995, pp. 232-241.

[7] V. K. Garg, C. Chase, "Distributed Algorithms for Detecting Conjunctive Predicates," Technical Report ECE-PDS-94-03, ECE Dept, University of Texas at Austin, 1994.

[8] V. K. Garg, and A. Tomlinson, "Using Induction to Prove Properties of Distributed Programs," *Proceedings of the Symposium on Parallel and Distributed Processing*, Dallas, Texas, December, 1993, pp.478-485.

[9] V. K. Garg, and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, March 1994, pp. 299-307.

[10] T. H. Lai, T. H. Yang, "On Distributed Snapshots," *Global States and Time in Distributed Systems*, pp. 23-26, IEEE Computer Society Press, Los Alamitos, CA, 1994.

[11] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems," *Proc. of the 21st Intl. Conf. on System Sciences*, Vol. 2, Jan 1988, pp 166 – 175. 1990, pp. 134 – 141.

[12] M. Hurfin, N. Plouzeau and M. Raynal, "Detecting Atomic Sequences of Predicates in Distributed Computations," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging,* San Diego, California, May, 1993.

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

[14] T. H. Lai, T. H. Yang, "On Distributed Snapshots," *Global States and Time in Distributed Systems*, pp. 23-26, IEEE Computer Society Press, Los Alamitos, CA 1994.

[15] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B. V., 1989, pp. 215–226.

[16] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, 2:161-175, 1987.

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

[18] B. P. Miller and J. Choi, "Breakpoints and Halting in Distributed Programs," *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988, pp. 316–323.

[19] J. Misra, "Detecting Termination of Distributed Computation Using Markers," *Proc. of the 2nd annual ACM Symposium on Principles of DC*, Aug, 1983, pp. 290-294.

[20] R. Schwartz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," SFB124-15/92, Department of Computer Science, University of Kaiserslautern, Germany, December 1992.

[21] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," *Proceedings of the 6th International Conference on Distributed Computing Systems,*, 1986, pp. 382-388.

[22] A.I. Tomlinson and V. K. Garg, "Detecting Relational Global Predicates in Distributed Systems," *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, May 1993, pp. 21–31.

[23] A. I. Tomlinson, V. K. Garg, "An algorithm for minimally latent global virtual time," *Proceedings on the 7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, May 1993.

```
inline Initialize_clocks_and_vars(i) {
    /* This initial code declares variables, ... */
    var sendlist, recvlist: vector of messages;
    vlocks: vector[1..n] of integer;

    ∀j : j ≠ i: vlock[j] = 0;
    vclock[i] = 1;
    firstflag = true;
    sendlist = recvlist = 0;
    }

Add_send_message_in_order(i, m) {
    /* Update the vector clock, set flag, add message to sendlist. */
    vclock[i]++;
    firstflag = true;
    sendlist = sendlist ⊕m; /* concatenate message m */
    }

Add_receive_message_in_order(i, m) {
    /* Update the vector clock, set flag, add message to recvlist. */
    ∀j:vclock[j] = max(vclock[j], m_vclock[j]);
    vclock[i] ++;
    firstflag = true;
    recvlist = recvlist ⊕m; /* concatenate message m */
    }

Update_send_rcv_of_token(i) {
    /* Update S and R of token from candidates and set channel pred's to false as needed. */
    for(all pending update messages) do {
        /* Update S and R from IR of pending update messages. */
        receive update message;
        j = source of update message;
        R[j] = R[j] ⊕ (message.IR - S[j]);
        S[j] = S[j] - message.IR;
        }
    for(all j: candidate.sendlist[j] ≠ 0) do {
        /* Update S and R, the send and receive lists for M_i. Each M_i maintains its own
        S and R and so evaluates channel predicates for sends directed from process i.
        S[j] = S[j] ⊕ candidate.sendlist[j] - R[j];
        R[j] = R[j] - candidate.sendlist[j];
        if (cp[j] = T ∨ cp[j] = F_r) cp[j] = X;
        CP[i] = X;
        }
    for(all j: candidate.recvlist[j] ≠ 0) do {
        /* For the recvlist, the messages received at process i are held
        until sent as updates indicating receipt to originating processes
        IR[j] = IR[j] ∪ candidate.recvlist[j];
        CP[j] = X;
        }
    }
```

Figure 12: Base GCP subroutines

```
local_predicate_true() {
      if (local_pred = true ∧ firstflag) then return true;
      }

clear_send_rcv_vars() {
      sendlist = recvlist = 0;
      }

channel_predicates_true(i) {
      /* Evaluate channel predicates on the cut. CP[] part of token. */
      while (M_i has token) {
            if (∃j: cp[j] = F_s) then {
                  CP[i] = X;
                  send_token(j);
                  }
            else if (∃j: cp[j] = F_r) then {
                  CP[i] = X;
                  send_token(i);
                  }
            else if (∀j: cp[j] = T) then {
                  CP[i] = true;
                  if (∃j: cp[j] = X) send_token(j);
                  else return true;
            else {
                  let j: CP[j] = X;
                  cp[j] = evaluation of channel predicate for S[j];
                  }
            }
      }

send_token(k) {
      / * When sending token always inform sending processes of messages
      received from last time M_i had token until now. */
      for (all j: IR[j] ≠ 0) {
            updates[j] ++; /* This vector is sent with the token. */
            send IR[j] to A_j;
            IR[j]:=0;
            }
      send token to M_k
      }
```

Figure 13: Base GCP subroutines (cont'd)

```
Increment_send_count(i) {
    /* Update the vector clock, set flag, increment send_count.*/
    vclock[i]++;
    firstflag = true;
    send_count++;
    }

Increment_receive_count(i) {
    /* Update the vector clock, set flag, increment receive_count.*/
    ∀k:vclock[k] = max(vclock[k], m_vclock[k]);
    vclock[i] ++;
    firstflag = true;
    receive_count++;
    }

local_predicate_true() {
    if ((application process idle) ∧ firstflag) then return true;
    }

clear_send_rcv_vars() {
    send_count = receive_count = 0;
    }

Update_send_rcv_count_of_token() {
    /* Update token counts, S and R, based on candidates.
    NOTE: S and R are part of token. */
    for(all j: candidate.send_count[j] ≠ 0) do {
        S = S + send_count;
        }
    for(all j: candidate.receive_count[j] ≠ 0) do {
        R = R + receive_count;
        }
    }

channel_predicates_true(i) {
    /* Evaluate channel predicates on the cut */
    while (M_i has token) {
        if (R ≠ S) then
            send token to M_j (for some j); /*M_j chosen randomly*/
        else return true;
        }
    }
```

Figure 14: GCP subroutines for termination detection with consistent cuts.

```
Increment_send_count(i, j) {
    /* Update the vector clock, set flag, increment send_count.*/
    vclock[i]++;
    firstflag = true;
    send_count_j++; /* P_j is message destination */
    }

Increment_receive_count(i, j) {
    /* Update the vector clock, set flag, increment receive_count.*/
    ∀k:vclock[k] = max(vclock[k], m_vclock[k]);
    vclock[i] ++;
    firstflag = true;
    receive_count++;
    }

local_predicate_true() {
    if ((application process idle) ∧ firstflag) then return true;
    }

clear_send_rcv_vars() {
    for (all j) send_count_j = 0;
    receive_count = 0;
    }

Update_send_rcv_count_of_token(i) {
    /* Update token counts, S_ij and R_i, based on candidates.
    NOTE: S_ij and R_i are part of token. */
    for(all j: candidate.send_count[j] ≠ 0) do {
        S_ij = S_ij + send_count_j;
        }
    for(all j: candidate.increcv[j] ≠ 0) do {
        R_i = R_i + receive_count;
        }
    }

channel_predicates_true(i) {
    /* Evaluate channel predicates on the cut */
    while (M_i has token) {
        if (∃j: R[j] ≠ ∑_k S_kj) then
            send token to M_j;
        else return true;
        }
    }
```

Figure 15: GCP subroutines for termination detection without consistent cuts.

```
Increment_send_count(i, j) {
    /* Update the vector clock, set flag, increment send_count.*/
    vclock[i]++;
    firstflag = true;
    send_count_j++; /* P_j is message destination */
    }

Increment_receive_count(i, j) {
    /* Update the vector clock, set flag, increment receive_count.*/
    ∀k:vclock[k] = max(vclock[k], m_vclock[k]);
    vclock[i] ++;
    firstflag = true;
    receive_count_j++;
    }

local_predicate_true() {
    if (firstflag) then return true;
    }

clear_send_rcv_vars() {
    for (all j) send_count_j = receive_count_j = 0;
    }

Update_send_rcv_count_of_token(i) {
    /* Update token counts, S_ij and R_ji, based on candidates.
    NOTE: S_ij and R_ji are part of token. */
    for(all j: candidate.send_count[j] ≠ 0) do {
        S_ij = S_ij + send_count_j;
        }
    for(all j: candidate.increcv[j] ≠ 0) do {
        R_ji = R_ji + receive_count_j;
        }
    }
channel_predicates_true(i) {
    /* Evaluate channel predicates on the cut */
    while (M_i has token) {
        if (∃j, n : send_count_jn ≤ receive_count_jn − CAP) then
        /* CAP is the maximum capacity of the channel. */
            send token to M_j;
        else return true;
        }
    }
```

Figure 16: GCP subroutines for channel capacity usage

```
local_predicate_true() {
    if (LVT > TVT ∧ firstflag) then return true;
    }

channel_predicates_true(i) {
    /* Evaluate channel predicates on the cut. CP[] part of token */
    while (M_i has token) {
        if (∃j: min{S[j].vclock : j ∈ S} < TVT) then {
            CP[i] = X;
            send_token(j);
            }
        else if (∀j: min{S[j].vclock : j ∈ S} ≥ TVT) then {
            CP[i] = true;
            if (∃j: CP[j] = X) send_token(j);
            else return true;
            }
        }
    }
```

Figure 17: GCP subroutines for GVT > TVT