Implementation of Clocks and Sensors

Term Paper

EE 382N Distributed Systems

Dr. Garg

November 30, 2000

Submitted by:

Yousuf Ahmed
Chandresh Jain
Onur Mutlu

Global Predicate Detection in Distributed Systems

The detection of global predicates is one of the fundamental problems in distributed computing. The difficulty of the problem arises due to the fact that there is no shared memory and no shared clock in a distributed system. Hence, no process has access to the global state. Therefore, the truth value of a global predicate cannot be determined by a single process. Determining the truth value of a global predicate requires the participation of those processes which can change the value of the global predicate.

The importance of global predicate detection can be seen in several domains such as designing, testing, and debugging of distributed programs. For example, predicate detection is crucial for implementing breakpoints in a distributed debugger. If we want to stop the debugger when local predicates on different processes become true, it is necessary to detect that all the local predicates on different processes became true concurrently. This is not an easy task given that no process has access to the global state of the distributed system. In fact Chase and Garg have shown that global predicate detection problem is NP-complete [2].

In this paper, our objective is to present several algorithms used to detect global predicates and also give the implementation details of one such algorithm. In Section 1, we present a brief classification of predicates. Section 2 is a literature review of "predicate detection problem". It includes several possible algorithms and approaches to detect global predicates. Section 3 presents the implementation details of our project in which we implemented a token-based distributed algorithm to detect generalized conjunctive predicates.

1. Classification of Predicates

Predicates can be classified with respect to locality, stability, and strength. We will define each dimension and point out its importance.

1.1 Local, Channel, and Global Predicates

A local predicate is defined as a boolean-valued formula on a local state [5]. A process can obviously detect the truth of its local predicates. A channel predicate is any boolean function of the state of the channel [7]. The state of the channel is determined by the send events and receive events on the channel. A global predicate is a boolean valued formula that is formed by the conjunction of channel predicates and local predicates on different processes. We will call this a generalized conjunctive predicate (GCP) from now on in our paper.

1.1.1 Linear vs. Non-Linear Channel Predicates

A channel predicate is linear (monotonic) if given any channel state in which the predicate is false, either receiving more messages on the channel without sending any message or sending more messages on the channel without receiving any message is guaranteed to leave the predicate false. Linearity is usually a necessary condition for the efficient detection of channel predicates.

1.2 Stable vs. Unstable Predicates

A stable predicate remains true once it becomes true. An unstable predicate does not have such a property. Stable predicates can be detected by taking global snapshots of the system periodically as described by Chandy and Lamport [1]. However, such an approach may miss some snapshots in which an unstable predicate became true and therefore cannot be used for detection of unstable predicates.

1.3 Strong vs. Weak Predicates

A run of a distributed program generates a partial order of events, and there are many total orders consistent with this partial order [8]. A predicate is weak if there exists a total order of a distributed computation for which the predicate becomes true. This definition corresponds to Cooper and Marzullo's definition of *possibly* ϕ [3]. A predicate is strong if it becomes true for all possible orders. A strong predicate corresponds to Cooper and Marzullo's definition of *definitely* ϕ [3].

2. Review of Literature on Predicate Detection

Several different algorithms have been proposed to detect different classes of predicates. Although we will mention many of these algorithms in this section, we will only focus on those that are designed to detect weak unstable GCPs.

2.1 Detecting Stable Predicates

As we have already mentioned, stable predicates can be detected by taking periodic consistent global snapshots of the distributed system and checking whether or not the predicate was true in that snapshot. If the predicate was true at the end of a snapshot, then the algorithm detects the predicate as true. If it was false at the end of a snapshot then it should have been false at the beginning of the algorithm. The problem of detecting unstable predicates is more complicated since we cannot determine whether the predicate became true by taking periodic snapshots.

2.2 Lattice-Based Predicate Detection

One of the first algorithms to detect unstable predicates was presented by Cooper and Marzullo [3]. This algorithm constructs the lattice of consistent global states that correspond with an observed execution. A weak unstable predicate will be detected true if it is true for any global state in the lattice. A strong unstable predicate will be detected if

it becomes true in some global state for each path from the initial global state to the final global state. By examining the lattice of consistent global states, we can detect whether a predicate becomes true or not. Thus, Cooper and Marzullo's algorithm is able to detect both strong and weak unstable predicates. However, the cost of detection might be exponential due to the explosion of the number of global states in the lattice.

2.3 Centralized WCP Algorithm

More efficient algorithms to detect strong and weak unstable predicates were proposed by Garg and Waldecker [8, 9]. We will focus on their work on weak unstable predicates. In [8], they present a centralized algorithm to detect the weak conjunctive predicate (WCP) of the form ($l_1 \wedge l_2 \wedge ... \wedge l_n$) where l_i is a local predicate on process *i.* In this algorithm one process is designated as a checker process. All the other processes keep track of the truth values of their local predicates. Whenever its local predicate becomes true for the first time between two external events, each process sends its vector clock to the checker process. The checker process receives these vector clocks [4,11] from each process in a separate queue in FIFO order and tries to find a consistent cut by comparing the vector clocks that are at the head of the queues. If such a consistent cut is found then the predicate would be detected. Otherwise, the checker process can advance on the queue whose vector clock is smaller than any other. As shown in [8], this algorithm requires $O(n^2m)$ comparisons among vectors where n is the number of processes that are involved in the WCP and m is the maximum number of messages sent or received by any process. This algorithm is optimal, because any algorithm that is based on comparing vector clocks to determine the truth of a predicate requires at least $\Omega(n^2m)$ comparisons [8].

However, the algorithm proposed in [8] has several drawbacks. First, it requires a single checker process to store queues from every other process. This may require an unreasonable amount of space in the checker process. Second, there is no chance of detecting the predicate if the checker process fails. Third, it does not handle the channel predicates, which are different from local predicates as we mentioned in Section 1.1. Hence, it is not suitable for an application that requires the detection of channel predicates.

2.4 Token-based WCP Algorithm

To alleviate some of the drawbacks of this algorithm, Garg and Chase [6] propose two distributed algorithms for detecting weak conjunctive predicates. The first algorithm is a token-based algorithm that requires a monitor process M_i associated with each application process P_i. Each application process checks for its local predicate and sends a message (local snapshot) to its monitor process whenever its local predicate becomes true for the first time since the last receive or send event. The monitor process maintains a queue of the local snapshots of its application process. The monitor process is activated whenever it receives the token, which carries two vectors. One vector *G* defines the current candidate cut that is being examined for consistency. If *G[i]* has the value *k* then state *k* from process P_i is part of the candidate cut. Another vector, *color*, is used to indicate the color of the candidate states from each process. If the color of a state is red, then that state and all its predecessors have been eliminated and cannot satisfy the WCP. Therefore, the monitor process should receive a snapshot that happened after the state that is colored red. If the color of a state is green then there is no state in cut G that causally precedes that state.

The token is sent to monitor process M_i only when *color[i]* = red. Upon receiving

the token, M_i receives a new candidate state from its application process and checks whether it is consistent with the current cut. M_i repeats this process until it receives a candidate that is consistent with states from all other processes that are in cut G. Then the monitor process updates the vector clock of the token and examines the token to see whether any other states in G violate the concurrency requirement. If there is such a state on process *j*, M_i makes *color[j]* = red and sends the token to P_j. If there is no state that violates the consistency of cut G then WCP is detected. This algorithm has the same complexity in terms of number of comparisons as the centralized WCP algorithm discussed previously, however it is decentralized in the sense that it does not require a single checker process. Garg proposes a way to make this algorithm more parallel in [5]. However, this algorithm is still not suitable for detecting channel predicates. We now turn our attention to algorithms that are designed to detect Generalized Conjunctive Predicates, which include channel predicates.

2.5 Centralized GCP Algorithm

The detection of the following predicate (termination) requires the detection of several channel predicates: "All processes are passive and all channels are empty". Hence, channel predicates need to be detected in order to detect several important GCPs in distributed programs. We will survey a centralized and token-based algorithm for detecting GCPs. These algorithms are designed for the efficient detection of *linear* channel predicates.

The centralized GCP algorithm was proposed by Garg, Chase, Kilgore, and Mitchell [7]. This algorithm is quite similar to the centralized WCP algorithm. It makes use of a checker process to detect the GCP. All application processes are responsible for

detecting their own local predicates and keeping track of the state of their outgoing and incoming channels. The application process sends a local snapshot message to the checker process whenever it detects that its local predicate became true for the first time since the last receive or send event on the process. In this message, it includes its current vector clock just as in the centralized WCP algorithm along with some information about the current state of its channels. Specifically it sends the list of messages it received from (incremental receive history) and list of messages it sent (incremental send history) to other application processes since the last time it sent a local snapshot to the checker process, which maintains this information in separate queues dedicated for each process.

The task of the checker process is to find a consistent cut that satisfies all the channel predicates. Similar to the centralized WCP algorithm the checker process advances the cut on the state that has the smallest vector clock or on the state which does not satisfy any one of the channel predicates. Whenever the checker process finds a cut in which all states are concurrent and all channel predicates are satisfied, the GCP is detected. Obviously, this algorithm shares the same drawback with the centralized WCP algorithm in that it might impose unreasonable space and time requirements on the checker process and the whole detection process depends on one central process. Hence, we turn our attention to a token-based GCP algorithm proposed by Mitchell and Garg [12].

2.6 Token-based GCP Algorithm

This algorithm is an extension of the token-based WCP algorithm that handles channel predicates. The distributed system is divided into two domains by this algorithm: Application Domain and Detection Domain. Application Domain consists of the

application processes which communicate with each other using the normal program messages. The Detection Domain contains the monitor processes, which are paired with a specific application process. A monitor process can send messages to other monitor processes using a single token but not the application processes. It can receive messages from other monitor processes and its application process. To make the algorithm less costly the application process and monitor process can be placed on the same uniprocessor so that the communication cost between the monitor and the corresponding application process is minimized.

An application process keeps track of its local predicates and records the activity on its incoming and outgoing channels. When all of its local predicates become true, the application process sends to its monitor process its vector clock and lists of messages it has sent/received on its channels since the last time it sent a message to its monitor. These messages are queued in the monitor process.

The monitor process stays passive until it receives the token from another monitor process in the system. When it receives the token, it starts receiving candidate states from the queue (application process) until it gets a vector clock which is later than the vector clock that is maintained in the token. The vector clock maintained in the token specifies a possible cut in which desired predicates could be true. After receiving such a candidate state from the queue, the monitor process updates its own component of the vector clock maintained in the token and checks whether there are any states in the current cut that violates consistency. If it finds out that the candidate state on process *j* is inconsistent with then it sends the token to the monitor process *j*. Otherwise if the current cut is consistent, then the monitor process checks whether all the channel predicates are

satisfied. It is able to do so, because it has received information on all the channels of its application process. If all the channel predicates are true, the GCP is detected. On a high level view the application processes are responsible for detecting their local predicates and the monitor processes are responsible for detecting the channel predicates and finding a consistent cut that satisfies all the predicates. We will elaborate more on the details of the implementation of this algorithm in Section 3.

3. Implementation of Vector Clocks and Sensors

The implementation of vector clocks and sensors is an extension of the Webscape system designed initially. A Webscape system can be viewed as a collection of tables on the internet, where each table is a two dimensional array of cells. Each cell has two fields associated with it – expression and value. The value is the number displayed for the cell. The expression is a formula associated with the cell, which is used to evaluate its value. This may result in communication with other tables, for instance, if the cell's value is dependent on the value of some remote cells in other tables. A cell may also have a target cell. Whenever the value of the cell changes, the new value is sent to the target cell, which in turn updates its value. Target cells can be remote cells as well. The communication between the tables is established using cell servers as discussed in the problem statement.

3.1 Vector Clocks

Vector clocks are associated with every Webscape table. It shows the vector clock for that process at all times. It is assumed that each process knows the total number of tables (processes) and identity (process_id) of each table. This is read from the input file during the initialization of the Webscape tables. The format of the input file is discussed

later. The vector clock of a process is included in all out-going messages and is updated using Mattern's vector clock update rules [11]. Whenever a message is sent, the local component of the clock is incremented and upon receiving a message, a component-wise maximum of the vector is taken and then the local component is incremented.

3.2 Sensors

A sensor is a cell type in the Webscape table. Two types of sensors are implemented – local and global sensors. A local sensor is simply a boolean condition on local data cells. For example, (R2C3 > 4) && (R0C0 == 7) is a local sensor which becomes true only when the value of the cell in R2C3 exceeds 4 AND value of the cell R0C0 equals 7. The sensors can only acquire a value of 0 or 1 depending on whether the predicate defined by that sensor is true or false. A suffix LS is displayed in the table to distinguish between a normal cell and a local sensor.

Two types of global sensors, channel sensor and conjunctive sensors are also implemented. A channel sensor detects a channel property. It acquires a value of 1, if the channel predicate associated with it becomes true. For example the following channel predicate R1C0 → tick:8888 > 2 will be true if at anytime the number of messages sent from the cell R1C0 in this table to the table at tick exceeds 2. A suffix CS is displayed in the table to differentiate a channel sensor from other cells.

A conjunctive sensor detects conjunction of local and channel sensors. It is set to 1, when all the conjuncts (local sensor and channel sensor) become true, for example the following predicate (R1C0 → sorata:8888 >= 10) && (R0C0 > 20) is a conjunction of the channel sensor (R1C0 → sorata:8888 >= 10) and the local sensor (R0C0 > 20). A

suffix CNJS is displayed in the table to differentiate a conjunctive sensor from the other cells.

Webscape application can have one cell to detect the global predicate, which is a conjunction of local and channel predicates on multiple tables for example (tick:8888/R1C1 > 5) && (sorata:8888/R3C2 < 100) && (omni:8888/R1C1 → tick:8888 == 5) . The value of the cell is set to 1, when the global predicate is detected. A suffix GS is displayed in the table to differentiate a global sensor from the other cells.

All the sensors detect if the property they are monitoring ever became true. Once they turn true they stay true. An interface is provided to the user to manually reset them to false (0).

3.2.1 Channel Predicates

We have implemented an algorithm to detect linear, stable as well as unstable channel predicates. Also, our algorithm considers multiple channels between processes, i.e., there is a channel between each cell of the Webscape application to all other Webscape applications. Channel predicates are of the form R1C1 → tick:8888 > 5

The application process keeps track of the minimum (Tmin) and maximum (Tmax) number of messages in transit in the channel from each cell to every other process.

- Tmin [no_of_processes] [no_of_rows] [no_of_cols]
- Tmax [no_of_processes] [no_of_rows] [no_of_cols]

An AckHandler process keeps track of the number of messages sent (incsend history) by each cell to other processes and the number of messages received (increcv history) by a particular cell of other processes.

- SEND [no_of_processes] [no_of_rows] [no_of_cols]
- RECV [no_of_processes] [no_of_rows] [no_of_cols]

Whenever a cell *(a, b)* of process A_i sends a message to process A_j, it updates its AckHandler process H_i. H_i increments the send count of the messages from cell *(a, b)* to A_j in SEND. Then it sends a request message to AckHandler H_j to get the number of messages received by A_j from cell *(a, b)* of process *Ai*. H_j returns the receive count from its receive array RECV. H_i calculates the number of messages in transit at this state and returns the current value to A_i. A_i updates its Tmin and Tmax based on the current number of messages in transit.

Upon receive of a message from cell *(a, b)* of process A_j, the process updates its AckHandler, which simply increments the receive count of the messages from cell *(a, b) of* A_j in RECV. Whenever a channel sensor cell is evaluated, it gets the Tmin and Tmax values for that. Whenever the local predicates are true in a process, information about the channels is sent to the monitor process.

```
Application Process (Ai):

Initialize_clocks_and_vars(i);
Before send of message m to A_j do
      Increment_send_count(j, myrow, mycol);
Upon receive of message m from A_j do
      Increment_receive_count(j,row,col);
Upon local_predicate_true() do {
      send(v) to M_i as candidate
      clear_send_rcv_vars();
}


Monitor Process (Mi):

Upon receive of token do {
   do {
      receive candidate from A_i
      update send_rcv_of_token(i);
      } until candidate.v[i] > token.v[i]
   update_token(candidate.v[i]);
   if ∃j : j ≠ i: token.v[j] < candidate.v[j]
      then send_token(j); /* Send token to M_j */
   if (still has_token) {
      if chan_predicates_true() {
            then GCP = true;
            send_broadcast(pred_detected);
      }
   }
```

Figure 1 : GCP detection algorithm

3.2.3 Token Based Distributed Algorithm for Detecting Generalized Conjunctive Predicates (GCP)

A token based distributed algorithm for detecting GCP is implemented (Figure 1). It is based on the algorithm suggested by Mitchell and Garg [12]. The implementation of the algorithm is divided into application and monitor portions at each process. The application process checks for local predicates and keeps track of the activity in the channels associated with this process. The algorithm uses a token to detect GCP. The token contains a possible global cut in which the desired predicates could be true. The monitor receives the token and checks for a consistent cut. If the global cut is consistent then all the local predicates are true concurrently. If not, the token is forwarded to any process which violates the consistency. Also, other processes that violate the consistent cut are marked so that they will receive the token at some point in the future. Before forwarding the token, vector clock information inside the token is updated according to the current candidate. If the global cut is consistent, then the monitor process checks for the channel predicates associated with the application process. If channel predicates are also satisfied, then GCP will be detected. Otherwise, the whole detection process starts again by labeling the violating processes as inconsistent and sending them the token.

The monitor code is activated only if it has the token, otherwise it just buffers the candidates received from the application process. Figure 2 shows the message flow between the application process and monitor process. Candidates are nothing but the vector clocks in which the local predicate became true first time after an external event and channel states associated with them. The application and the monitor process are placed on the same uniprocessor to reduce communication complexity.

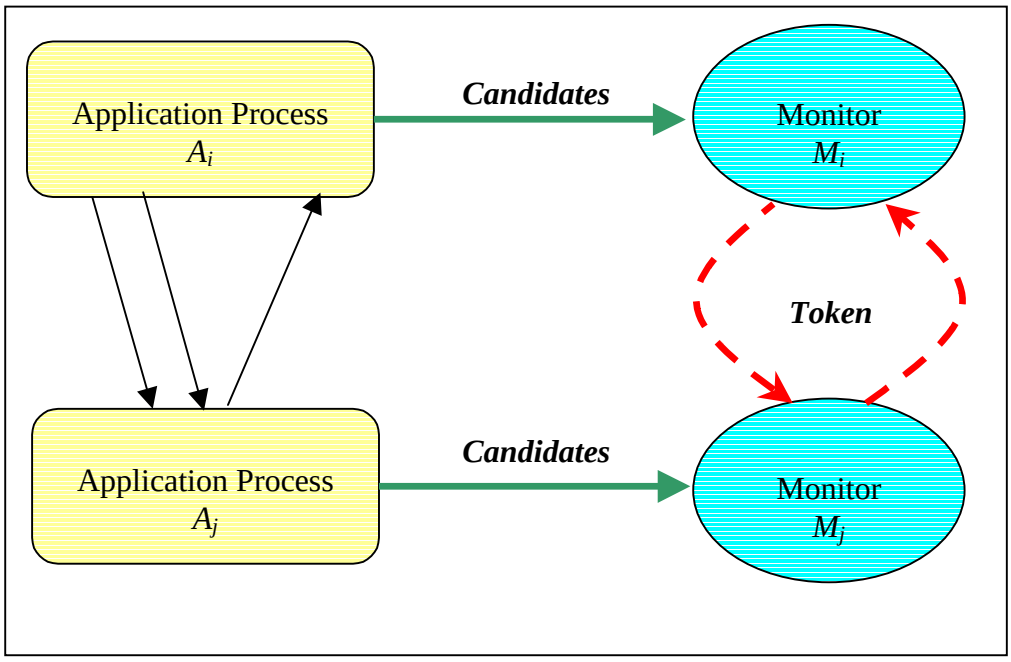

Figure 2 : The application and monitor process of GCP

3.3 Implementation Details

3.3.1 Initialization of the Webscape Table

The first step at the start of the process is the initialization of the Webscape table. The initial configuration of the table is read from an input file. The input file has the following information:

- Total number of processes
- Process ID of the running process
- Identity of the other tables (machine name:port, process id)
- Total number of rows and columns in this table
- Initial cell entries (expression, value, target)
- Local conjuncts (process' part of the global predicate)

User interface for updating the cell entries and saving the current state into a file is also provided. It is assumed that the table has a fixed size.

Every process has a unique process id. A hashtable of process id's and addresses is constructed and it is used for communication with other tables. A cell-server thread is launched during the initialization phase. The cell-server serves the requests for reading and updating cells. This server starts listening on the port number associated with the table. TCP/IP sockets are used for communication. Initialization for the AckHandler and the monitor processes is then done. The token, vector clock and other data structures are also initialized. Only one monitor process has the token in the beginning. The vector clock of the token is initialized so that the token eventually visits all the monitor processes at least once before the global predicate is detected.

3.3.2 Cell Updates

The run time system picks the cell in a round robin fashion, evaluates the expression associated with the cell, updates the new value in the table and if a target cell exists, it sends its value to the remote cell. Two interfaces *getRemoteVal* and *sendRemoteVal* are provided to achieve this functionality. *getRemoteVal* gets the value of the remote cell and *sendRemoteVal* sends the current value to the remote cell. It then checks if the local predicate and the channel predicates associated with it are true. If true, then it sends a **candidate** to the monitor process. The GCP algorithm is run in the monitor process as described in Section 3.2.3. One of the monitor process eventually detects the global predicate (if it exists) and sends a broadcast to all the other processes so that they stop checking for their local predicates.

4. Conclusion

In this project we have implemented a token-based distributed algorithm that is able to detect weak unstable Generalized Conjunctive Predicates, which are formed by the conjunction of local and channel predicates on multiple independent processes. Our implementation is based on an algorithm proposed by Mitchell and Garg in [12]. This algorithm would be very useful in debugging and testing distributed programs as discussed in Section 1.

One shortcoming of our implementation is the fact that it is not fault-tolerant. If one of the monitor processes or application processes become faulty, our implementation will not work correctly. Perhaps a next step to improve our project is to make it more fault-tolerant. Garg and Mitchell describe an algorithm that will detect a conjunction of local and send-monotonic channel predicates in faulty, asynchronous distributed environments in [10] using infinitely-often accurate detectors. Our implementation might be modified to be suited for their algorithm to be made more fault-tolerant.

Another problem with our implementation is that it relies on only one token. It is not distributed enough to provide higher levels of fault-tolerance. A more distributed algorithm is proposed in [7]. Perhaps, by combining the approaches of [7] and [10], a more fault-tolerant and parallel predicate detection scheme can be implemented.

References

[1] K. M. 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] C. M. Chase, and V. K. Garg, "Efficient Detection of Restricted Classes of Global

Predicates", *Proceedings of the 9th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science*, Vol. 927, pp. 303-317, 1995.

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

[4] C. Fidge, "Partial Orders for Parallel Debugging", *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, pp. 130-140, May 1988.

[5] V. K. Garg, "Observation of global properties in distributed systems", *IEEE International Conference on Software and Knowledge Engineering*, Lake Tahoe, Nevada, pp. 418-425, June 1996.

[6] V. K. Garg, C. Chase, ``Distributed Algorithms for Detecting Conjunctive Predicates", *Proc. IEEE International Conference on Distributed Computing Systems*, Vancouver, Canada, pp. 423-430, June 1995.

[7] V. K. Garg, C. Chase, J. R. Mitchell, R. Kilgore, ``Detecting Conjunctive Channel Predicates in a Distributed Programming Environment", *28th Hawaii International Conference on System Sciences*, pp. 232-241, January 1995.

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

[9] V. K. Garg, B. Waldecker, ``Detection of Strong Unstable Predicates in Distributed Programs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 12, pp. 1323 - 1333, December 1996.

[10] V. K. Garg, J. R. Mitchell, ``Distributed Predicate Detection in a Faulty Environment”, *Proceedings of IEEE International Conference on Distributed Computing Systems*, Amsterdam, Netherlands, 1998

[11] 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, pp 215-226, 1989.

[12] J. R. Mitchell, V. K. Garg, ``Deriving Distributed Algorithms from a General Predicate Detector", *Proceedings of the Nineteenth Intl. Computer Software and Applications Conference*, Dallas, Texas, pp. 268 -- 273, August 1995.