Global State Deadlock Detection Algorithms

Distributed Systems Final Project

By Ojaswi Binnani(20161006), Muhammed Yaseen Harris(20161047), Mahathi Vempati(20161003)

The idea of global state deadlock detection is to somehow capture a snapshot of the Wait-for-graph and examine it for deadlocks. This snapshot can be distributed and each node can independenly help in the examining of deadlocks. The examination and snapshot taking need not be done simultaneously.

We discuss three algorithms here: Kshemkalyani-Singhal, Bracha Toueg, and D.A.S -

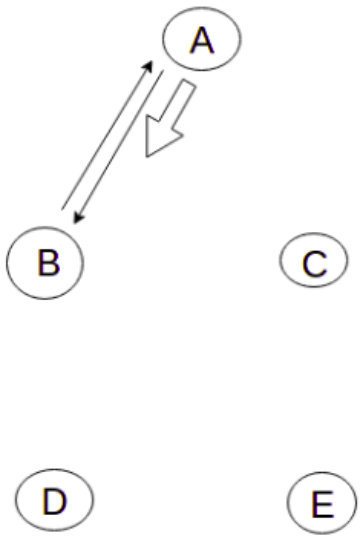
Brief Comparison

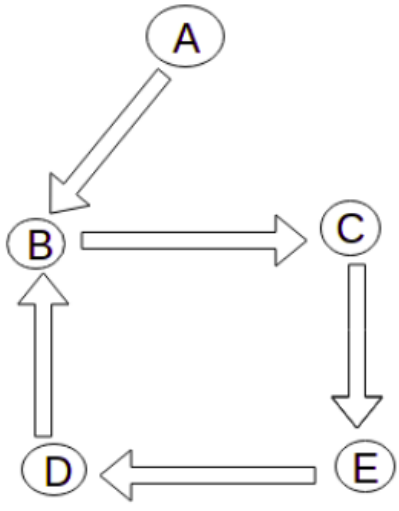
Criteria	**Kshemkalyani-Singhal**	**D.A.S.**	**Bracha-Toueg**
Deadlock detection - i.e. all possible deadlocks are detected	True	True	True
Phantom Deadlocks- i.e. deadlocks that don’t exist are detected	False	False	False
Works in Non-FIFO	No	No	No
Works in FIFO	Yes	Yes	Yes
Works in Causal Channel	Yes	Yes	Yes
Message Complexity	4e - 2n + l, where e is the number of edges, n is the number of nodes, l is the number of leaf nodes.	O(2n) - n forward and n backward messages sent between initiator and all other processes in worst-case.	O(4N^2) - 4 different types of messages that a process can send, NOTIFY and GRANT messages can be sent to more than one process at one time.

Criteria	Kshemkalyani-Singhal	D.A.S.	Bracha-Toueg
Message Length, for each type of message in the algorithm.	FLOOD - {2 bit identifier, Sender name, weight}, SHORT - {2 bit identifier, weight}, ECHO - {2 bit identifier, weight}. The space dedicated to weight depends on the number of messages expected to be sent, which is shown above. This is so that the weight-throwing algorithm can continuously divide without issues.	O(Maximum number of outgoing edges of a node)	2 bits (4 different types of messages can be represented in two bits). However, the way we coded it, the message contains the process id and the message number.
Time Complexity	O(2d), where d is the diameter of the graph	O(2d) where d is the diameter of the graph	O(4T^2) where T is the maximum time taken to send a message
Snapshot mechanism for global state is based on	Lamport	As proposed in the algorithm	Lamport
System assumptions for algorithm to work	Yes, we assume that once a node is free, it first replies to pending requests and only then sends more requests to other nodes. Taking the diagram given below itself, assume that the REQUEST reached before the FLOOD, and an ECHO was sent back as well, but after getting the REQUEST, instead of sending a REPLY, the algorithm decides to send out other REQUESTS and gets into a deadlock.	None	The algorithm assumes that each process knows which processes are dependent on it and which it is dependent on. It also assumes that the process that receives a message knows the sender of the message.Here also, the deadlock detection algorithm gives the wrong output.
Language used for coding the algorithms	Erlang	Erlang	C++, MPI

Criteria	Kshemkalyani-Singhal	D.A.S.	Bracha-Toueg
How does implementation vary from the real algorithm?	In the Erlang implementation, we assume that the underlying processes (REQUEST, REPLY) are much slower than the deadlock detection - to the extent that the Wait For Graph remains constant (no REQUESTS/ REPLIES during deadlock detection)- We allow the user to input a Wait For Graph, This graph is distributed among the processes such that each of them know who they are dependent on , The processes then run the algorithm to determine if they are deadlocked.	The original algorithm is run on a P out of Q model and assumes a dynamic process underneath. Because of this, to ensure that a consistent snapshot is taken, we need to send the time back with every process. Then, edges that were placed after the snapshot are thrown away (removal of unmatched edges). Also, we need to check for ties. In our implementation we run it on a static AND model, so we look for cycles, and we don't remove unmatched edges.	We did not find code online for this algorithm, so we cannot compare its implementation differences. However, each functionality of the algorithm is coded in our implementation. A difference from the pseudocode is that our messages contain the process id sending the message. Also, we added another message type which the initiator sends after the algorithm is terminated. This adds another n-1 messages to the algorithm. More variables were added as well.

Why NON-FIFO channels don't work?





In the left diagram, A first sends a REQUEST (thick arrow), which is part of some underlying computation. The REQUEST does not reach B yet. Now, to check if it is in a deadlock, it sends a FLOOD (thin arrow). As you can see above, since B is not waiting on anyone, it sends an ECHO back to A to confirm to it that it is NOT in a deadlock. (FLOOD and ECHO are algorithm specific here, but this will hold for any similar algorithm).

Now, by the time A's REQUEST reaches B, it could get itself into a deadlock, and hence A would be in a deadlock, even though it assumed it wasn't by the deadlock detection algorithm.

Hence, this does not work for non-FIFO channels. They work for both FIFO channels and causal channels. Since the algorithms do not send any extra messages to compare the time of two messages, there is no extra advantage of being causal.

Kshemkalyani-Singhal

Basic Idea

Assume there is a situation with processes having outstanding requests for resources that is depicted in the form of a Wait-For Graph. Each edge in the graph is an outstanding request. Each process in the whole distributed system has knowledge of only their local Wait-For Graph, that is, the requests to them, and the requests they have sent out.

The MAIN IDEA here is, a Wait-For Graph is dynamic. When processes finish their work, they send replies to those who have requested them for resources. When a reply is sent, we consider that edge in the Wait-for graph as removed. This process of edges slowly being removed is called the reduction of the Wait-For graph. Finally, if there comes a situation where all outgoing edges of the initiator are removed, the initiator is not in a deadlock.

Now, assume at some point in time, a node has a few outgoing requests that seem to be taking time, and it wonders if it is in a deadlock. It initiates the K-S algorithm to SIMULATE this reduction of Wait-For graph quickly, so that it can check if it is in a deadlock, or it will get replies to its requests at some point in the future.

Note that when this algorithm is running, the underlying computation is running as well, so actual replies may be sent across that remove edges from the graph while the deadlock detection is running. Hence, the algorithm should take care of this as well.

Theory

The underlying computation is spoken about in terms of REQUESTS and REPLIES. A request is an edge on the graph. A REPLY deletes the edge.

In the deadlock detection algorithm, we use three kinds of messages: FLOOD, ECHO and SHORT. FLOOD - this is used to nudge every node to inform it that some initiator node has started a deadlock detection algorithm. ECHO - This simulates the REPLY's. If at some point in the future, the underlying computation would have sent a REPLY, K-S detects this, and an ECHO is sent now. SHORT - This is to send back extra weight as part of termination detection.

If the initiator receives as many ECHOes as it would have needed REPLY's, it assumes it is not in a deadlock. If the algorithm terminates without enough ECHO's being received, then it is in a deadlock. Note that the Wait-For Graph is constantly changing as real replies keep deleting edges. Then which set of nodes should a process send ECHOes to when it determines it is going to be free at some point?

To do this, the first FLOOD that arrives at a node takes a snapshot of all the incoming requests. ECHOs are then sent back to those requests.

Proof of Correctness

The proof of correctness briefly involves three steps:

1. We can show that the execution of the algorithm terminates
2. The entire Wait For Graph reachable from the initiator is recorded in a consistent distributed snapshot in the outward sweep.
3. In the inward sweep, ECHO messages correctly reduce the recorded snapshot of the WFG.

Deng-Attie-Sun Algorithm

Basic Idea

- The Deng-Attie-Sun algorithm detects generalized deadlocks in a distributed system .
- Detecting generalized deadlocks in distributed systems is a difficult problem, because it requires detection of a complex topology in the global WFG.
- A process initiates the algorithm when it blocks on a resource request. Instead of recording a distributed snapshot, the algorithm incrementally constructs an “image” of the WFG, which is stored locally at the initiator process.
- The algorithm is composed of a sequence of stages. In the first stage, the initiator process i sends an inquiry (called FORWARD) message to each process j which it is waiting for, and then each j reports its state information to i via a BACKWARD message; in the second stage, it sends a FORWARD message to each process k which some j which was involved in the first stage is waiting for, and then each k reports its state information to i via a BACKWARD message.
- This process continues stage-by-stage in a similar manner. At each stage, the new processes are those processes that the processes of the previous stage are waiting for.
- At the end of each stage, the WFG is locally (at process i) updated (based on the new information from the received BACKWARDS), reduced, and checked for the existence of a deadlock.

Theory

- The initialization of the proposed algorithm is composed of three steps:
    - WFG_i is created at the initiator i, containing only one vertex i, where i.t = t-blocki, i.out = outi, i.in = in_i , and i.p = p_i ;
    - New_i = out
    - Pool_i := 0.
- New_i is the set of new vertices which will be inserted into W FG_i at the next stage
- Each stage of the algorithm can be divided into the following five steps:
    - For every j E Newi, if it is in Pooli, then remove it from Pool_i and insert it into WFG_i ; otherwise, send a FORWARD message to process j .
    - When i receives a BACKWARD message from j , a new vertex j is inserted into WFG_i . (The edges associated with j are inserted automatically.) After i receives all BACKWARD messages sent by the processes in Newi, go to the next step.
    - (a) Remove all unmatched edges from WFGi, (b) Remove all reducible edges from WFGi, (c) Remove all invalid vertices from WFG_i and New_i.
- New_i = Union of all j.out_i - WFGi.N, where WFGi.N is the set of vertices in WFGi.
- One of the following actions is taken:
    - (a) If there is a tie in WFG_i , a deadlock is detected and the algorithm terminates.

    - (b) If i.p = 0, the algorithm terminates, with no deadlock detected.
    - (c) If New_i = 0 and there is no tie in WFGi, the algorithm terminates, with no deadlock detected.
    - (d) Otherwise, go to the next stage.
- When a process j receives a FORWARD message from the initiator i, it responds simply by sending a BACKWARD message (t-blockj, inj, outj , pj) to i.

Proof of Correctness

- Termination: The algorithm terminates as mentioned in step v. of each stage of the algorithm. All the various outcomes are hence taken care of.
- Every deadlock in the system will be detected: That is, there will be a tie in the WFG constructed by some instance of the algorithm for every deadlock set in the system.
- No false deadlock is detected: That is, every tie in the WFG constructed by the algorithm corresponds to a deadlock set in the system.

Bracha-Toueg

Basic Idea

Bracha-Toueg is a global state deadlock detection algorithm that detects deadlocks in the AND model. It detects cycles in the graph and concludes whether the initiator is deadlocked or not. Since the algorithm runs a Lamport's algorithm to find each process's local wait-for-graph, we can say that this algorithm works on a FIFO channel, and hence would work in a causal channel and would fail in a NON-FIFO channel.

The algorithm starts off by using Lamport's Algorithm so that each process finds its local wait-for-graph. Each process knows the messages it has received, hence knows the processes waiting on this process (in array), and the messages it has sent, hence knows the processes it is waiting on (out array).

When a GRANT message is sent, we assume that they have given the resource to the processes that need it, and the algorithm concludes that the process does not belong in a cycle or is not waiting on a deadlocked process.

If the initiator becomes free to release its resource, we can conclude that it too does not belong in a cycle and is not waiting for a resource from a deadlocked process, and hence the deadlock detection algorithm outputs that the initiator is not deadlocked.

Theory

This algorithm has four different types of messages in use: Notify, Grant, Done, Ack. It also has two functions Notify and Grant. Each process keeps a track of all the edges dependent on it (in), and all the edges it is dependent on (out). The number of processes in out is kept in requests. Additionally, each process has two variables notified and free. Originally, each variable is set to false. To begin the algorithm, the initiator goes into the Notify function.

Notify_u ->

- Set notified as true
- For all w belonging in the out array, send notify message
- If requests == 0 -> Enter Grant function

- For all w belonging in the out array, wait for done message

Grant_u ->

- Set free as true
- For all w belonging in the in array, send grant message
- For all w belonging in the in array, wait for ack message

On receiving notify message

- If notified = false -> Go into Notify function
- Send done message to source

On receiving grant message

- If (requests >0)
    - Requests = requests - 1
    - If requests == 0 -> Go into Grant function
- Send ack message

When the initiator receives all the done messages, it can conclude that the algorithm has completed. If its variable free is set to true, it is in a non-deadlocked state, and if the variable is false, it is deadlocked.

Proof of correctness

- Algorithm Terminates The algorithm terminates when the initiator receives all the done messages because every other process would have done whichever functions it should have performed BEFORE sending the done message back.

- Algorithm detects if the initiator is deadlocked If the initiator is in a cycle, none of the processes in the cycle go into grant, and hence none of the processes become free. Since, the initiator is not free as well, and after the algorithm terminates, the free variable of the initiator will remain false and hence deadlock will be declared.If the initiator is waiting on a deadlocked process (e.g. the process is in a cycle, or waiting on a process in a cycle, etc.), that process does not enter the grant state and hence does not become free. Hence the initiator does not become free, and deadlock will be declared.

- Phantom Deadlocks are not detected The initiator only becomes free when it has received all the grant messages it needed, implying the resources it needed are available to it. Hence, when the initiator becomes free, no-deadlock is declared. It cannot be declared as a deadlock since the variable is changed.

Implementation Details

The implementation allows the user to enter the wait-for-graph as a 2-d graph. The graph is distributed amongst the processes so each process knows its in group and out group.

While inside a function, the process must still be able to reply to messages it receives. We handled this by having each process check if a message is coming. Depending on the message it receives, it reacts differently. For example, if it receives a done message, it checks how many more done messages it needs to receive, and if it is zero, it sends its done message to its parent.

The algorithm assumes that a process receiving a message knows the sender process. We implement this making the message contain the process id sending the message as well.

We also added an extra message that the initiator sends when the algorithm is over to make the other processes stop waiting for or checking for messages.

Resources

1. Bracha-Toureg: link
2. Kshemkalyani Singhal: The Kshemkalyani textbook
3. D.A.S : link

(The accompanying video shows example dry runs on various graphs as well as code runs).