Distributed Algorithm on AHCv2: Bracha-Toueg Deadlock Detection Algorithm

Release V1.0.0

İmre Kosdik

May 11, 2024

CONTENTS

CHAPTER

ONE

BRACHA-TOUEG DEADLOCK DETECTION ALGORITHM

1.1 Abstract

In distributed systems, deadlocks occur due to resource sharing - the concept determines how existing resources are shared and accessed across the system. A deadlock is a condition that the processes request access to resources held by other processes in the system. Resolving the deadlocks is crucial because the processes involved are blocked and waiting indefinitely to acquire resources from the others. Deadlock handling is a challenging problem in distributed systems because no site knows the system state, and the communication involves finite and unpredictable delays. *Bracha-Toueg Deadlock Detection Algorithm* offers a simple and efficient way of detecting deadlocks in distributed systems. Implementing *Bracha-Toueg Deadlock Detection Algorithm*, along with analyzing the message and time complexity by running the algorithm on different network topologies and various numbers of nodes, provides valuable insights into the algorithm's wide-range applicability due to its efficiency and scalability.

1.2 Introduction

Distributed systems are desirable because they allow resource sharing, a critical concept determining how existing resources are shared and accessed across the network. However, resource sharing comes at a price - namely, deadlocks. A deadlock is a condition that the processes request access to resources held by other processes in the system. Deadlocks can cause significant delays and affect a distributed system's performance, making it crucial to resolve them as soon as possible.

Deadlock detection algorithms are crucial due to their role in concurrency and control mechanisms in distributed systems. Since they help identify and resolve the deadlocks, they prevent system failures and waste of resources. Moreover, they improve systems' reliability by contributing to deadlock resolution. The absence of deadlock detection algorithms causes execution halts since deadlocks state that the processes cannot continue their work. Also, it wastes resources because resources held by the processes are neither used nor released. Therefore, this can affect a system's scalability, response time, and throughput. Overall, deadlocks are inherent risks in concurrent and distributed computing environments, and effective detection mechanisms are essential for mitigating their impact and ensuring the smooth operation of distributed applications.

Because deadlock detection requires storing the system state as a graph, the lack of shared memory becomes a bottleneck for constructing and maintaining the graphs. Another issue is to ensure that each process needs to have an accurate knowledge of the system state. However, in distributed systems, communication delays and failures are almost inevitable. Since distributed systems are large and complex, designing an efficient and scalable deadlock detection algorithm is also challenging.

In this paper, we aim to thoroughly explain the implementation details of *Bracha-Toueg Deadlock Detection Algorithm*, discuss how the algorithm overcomes the challenges arising from the nature of distributed systems, and present the results of experiments we conducted on the algorithm related to its time and message complexity. The experiments prove the algorithm is a valuable asset in detecting deadlocks due to its efficiency and simplicity.

We contribute to the field of distributed systems by:

- Implementating *Bracha-Toueg Deadlock Detection Algorithm* on the AHCv2 platform. We explain the implementation details in Section 1.3.
- Conducting experiments on the algorithm over different network topologies and node counts. We discuss the experiment setup and results in Section 1.4

1.3 Bracha-Toueg Deadlock Detection Algorithm

1.3.1 Background and Related Work

A deadlock occurs when a group of processes waits for each other to acquire resources to continue their execution. One of the deadlock models is N-out-of-M Requests, where N is less than or equal to M. In this model, a process makes M requests and can continue execution only if it obtains at least N resources.

Wait-for-graphs model the resource dependencies in distributed systems. [Kshemkalyani2008] In these graphs, nodes represent processes, and there is a directed edge from one process to another if the first process is waiting to acquire a resource that the second process is currently holding. A process can be either active or blocked. An active process has all the resources it needs and is either executing or ready to execute. On the other hand, a blocked process is waiting to acquire the resources it needs.

An active node in a WFG can send an N-out-of-M request. After sending the request, the node becomes blocked until at least N of the requests are granted. Once the node becomes blocked, it cannot send any more requests. Directed edges are included in the graph to indicate the requests, and they go from the node to each node containing the required resources. As nodes grant the resources to the blocked node, the system removes the directed edges correspondingly. Once N requests are approved, the node becomes active again and sends notifications to M-N nodes to dismiss the remaining requests. After that, the system removes the remaining directed edges accordingly. [Bracha1987] [Fokking2013]

Deadlock detection is a fundamental problem in distributed computing, which requires examining the system's WFG for cyclic dependencies. For this purpose, the processes in the system periodically check whether the system contains any deadlock by taking a snapshot of the global state of the system. According to Knapp's deadlock detection algorithm classification [Knapp1987], this approach falls under the global state-based algorithms. These algorithms including the *Bracha-Toueg Deadlock Detection Algorithm* is based on Lai-Yang Snapshot Algorithm [Fokking2013] because the algorithm computes WFG depending on the global snapshot of the system.

1.3.2 Bracha-Toueg Deadlock Detection Algorithm: Bracha-Toueg Deadlock Detection Algorithm

The *Bracha-Toueg Deadlock Detection Algorithm*, proposed by Gabriel Bracha and Sam Toueg [Bracha1987], aims to detect the deadlocks in the system. The algorithm operates on the N-out-of-M deadlock model and is under the assumption that it is possible to capture the consistent global state of the system without halting the system execution. The algorithm starts execution when a node, named initiator, suspects that it may be in a deadlocked state. This can happen after a long wait for a request to be satisfied. The initiator starts a Lai-Yang snapshot *Lai-Yang Snapshot Algorithm* to compute the WFG. To differentiate between snapshots invoked by different initiators, the algorithm associates each snapshot, along with its messages, with the initiator's identity. After a node v constructs its snapshot, it computes two sets of nodes:

- **OUTv**: The set of nodes *u* for which *v*'s request has not been granted or relinquished.
- **INv**: The set of nodes requesting a service from *v*, according to *v*'s point of view. The node *v* received requests from a set of nodes, but *v* has not yet granted or dismissed the requests.

After computing each set of nodes, the algorithm consists of two phases. *Notify* - where processes are notified that the algorithm started execution - and *Grant* in which active processes simulate the granting of requests.

- The process initiating the deadlock detection algorithm sends NOTIFY messages to all processes in *Outv*. (Line 2)
- If the initiator process does not need any resources, it grants its resources to processes needing them by sending *GRANT* messages (Line 19) and makes itself free. (Line 18) It then waits for *ACKNOWLEDGE* messages from these processes indicating that they received the *GRANT* message.(Line 20)
- After performing the *GRANT* operation, it waits *DONE* messages from the processes it sent *NOTIFY* message to. (Line 7)
- If a process receives *NOTIFY* from another for the first time, it sends NOTIFY messages to all processes in its *Outv*. (Line 14). Then, it sends *DONE* message to the process sending the *NOTIFY* message. (Line 16)
- If a process receives *GRANT* message from another, it checks whether it needs additional resources to continue execution. (Line 22). Once it does not need any resources, it grants its resources to waiting processes by executing grant. (Line 25) After that, it sends *ACKNOWLEDGE* message to the process sending the *GRANT* message. (Line 28)
- Once the initiator process receives done from all processes in *Outv*,(Line 7) it checks the value of *free* and decides whether it is deadlocked. (Line 9)

Listing 1: Bracha-Toueg Deadlock Detection Algorithm [Fokking2013].

```
1  Procedure Notify
2  notified <- true
3  send<notify> to all w   OUT
4  if requests = 0 then
5          perform Procedure Grant
6      end if
7  await<done> from all w  OUT
8  if free then:
9      conclude that it is not deadlocked
10 end if
11
12 Upon receipt by v of Notify from a neighbor w:
13 if notified = false then
14         Perform Procedure Notify
15 end if
16 send<done> to w
17
18 Procedure Grant
19 free <- true
20 send<grant> to all w  IN
21 await<ack> from all w  IN
22
23 Upon receipt by v of Grant from a neighbor w:
24 if requests > 0 then
25         requests <- request - 1
26     if requests = 0 then
27             Perform procedure Grant
28         end if
29 end if
30 send<ack> to w
```

1.3.3 Lai-Yang Snapshot Algorithm:

The *Bracha-Toueg Deadlock Detection Algorithm*, utilizes *Lai-Yang Snapshot Algorithm* to compute the WFG graph. Therefore, the process starting the deadlock detection algorithm first executes the Lai-Yang snapshot algorithm. The deadlock detection algorithm uses the global state information captured with *Lai-Yang Snapshot Algorithm* to detect deadlocks. Since this paper focuses on implementing the *Bracha-Toueg Deadlock Detection Algorithm*, we do no explicitly explain the pseudocode given for *Lai-Yang Snapshot Algorithm* below. We give the pseudocode here since we implemented the algorithm as a part of *Bracha-Toueg Deadlock Detection Algorithm*.

Listing 2: Lai-Yang Snapshot Algorithm [Fokking2013]

```
bool recorded
nat counter[c] for all channels c of p
mess-set State[c] for all incoming channels of p

if p wants to initiate a snapshot
perform Procedure TakeSnapshot

if p sends a basic message m into an outgoing channel c<0>
send<m,recorded> into c<0>
if recorded is False then
    counter[c<0>] <- counter[c<0>] + 1
end if

if p receives <m, b> through an incomming channel c<0>
if b = True then
    perform Procedure TakeSnapshot
else
    counter[c<0>] <- counter[c<0>] - 1
    if recorded = True then
        State[c<0>] <- State[c<0>] U {m}
        if |State[c]| + 1 = counter[c<0>] for all incoming channels c of p then
            terminate
        end if
    end if
efficiencynd if


if p receives <presnap, l> through an incoming channel c<0>
counter[c<0>] <- counter[c<0>] + L
if |State[c]| + 1 = counter[c<0>] for all incoming channels c of p then
    terminate
end if

Procedure TakeSnapshot
if recorded = false then
    recorded <- True
    send <presnap, counter<c0>> into each outgoing channel c
    take a local snapshot state of p
end if
```

1.3.4 Example With Deadlock Present in The System

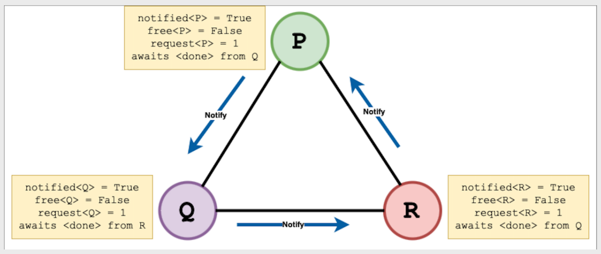

Fig. 1: Step 1

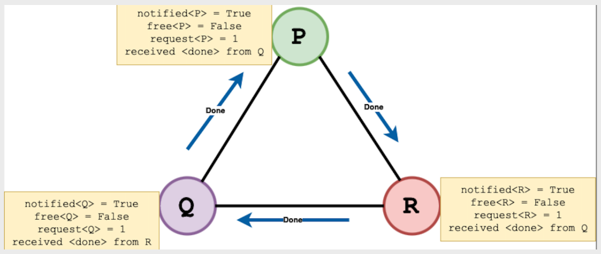

Fig. 2: Step 2

Assume a system with three processes, P, Q and R. The wait-for graph consists of three 1-out-of-1 requests, has been computed in a snapshot. Initially *requests<P>* = *requests<Q>* = *requests<R>* = 1. The walkthrough of the *Bracha-Toueg Deadlock Detection Algorithm* is as follows:

1. The initiator P, sets *notified<A>* to true and sends <**notify**> to Q. P awaits <**done**> from Q. (See Figure 1)
2. Q receives <**notify**> from P and sets *notified<Q>* to true. In order to send <**done**> to P, Q sends <**nofity**> to R and awaits <**done**> from R. (See Figure 1)
3. R receives <**notify**> from Q and sets *notified<R>* to true. In order to send <**done**> to Q, R sends <**nofity**> to P and awaits <**done**> from P. (See Figure 1)
4. Since *notified<P>* is true, P does not send any <**notify**> messages. It directly sends <**done**> to R. (See Figure 2)
5. R sends <**done**> to Q because R is already notified. (See Figure 2)
6. Q sends <**done**> to P because Q is already notified. (See Figure 2)
7. Once P receives <**done**> from all its OUT, consisting of Q, it checks the *free<A>*, and since *free<P>* is false, it concludes that the resources are never granted and it is deadlocked.

1.3.5 Example With Deadlock Not Present in The System

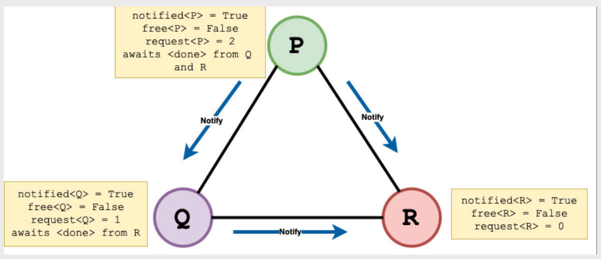

Fig. 3: Step 1

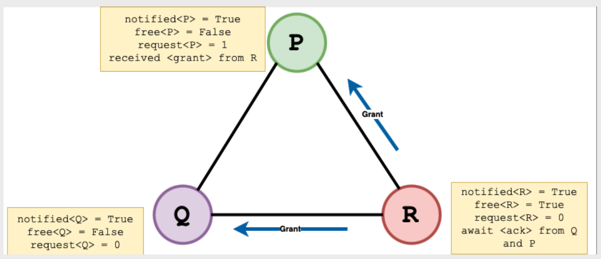

Fig. 4: Step 2

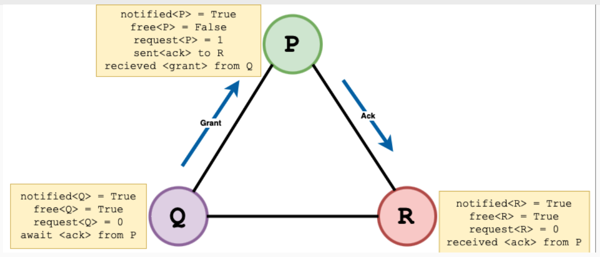

Fig. 5: Step 3

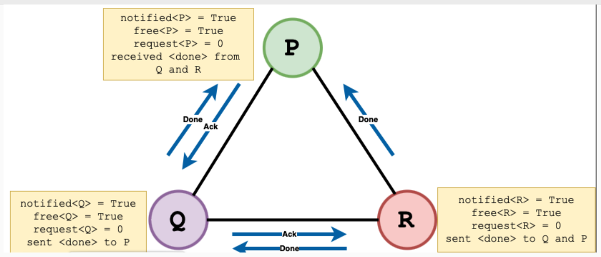

Fig. 6: Step 4

Assume a system with three processes, P, Q and R. The wait-for graph consists of three 1-out-of-1 requests, has been computed in a snapshot. Initially *requests<P>* = 2, *requests<Q>* = 1 and *requests<R>* = 0. The walkthrough of the *Bracha-Toueg Deadlock Detection Algorithm* is as follows:

1. The initiator P, sets *notified<P>* to true and sends <**notify**> to Q and R. A awaits <**done**> from Q and R. (See Figure 3)
2. Q receives <**notify**> from P and sets *notified<Q>* to true. In order to send <**done**> to P, Q sends <**nofity**> to R and awaits <**done**> from R. (See Figure 3)
3. R receives <**notify**> from Q and P and sets *notified<R>* to true. Since requests<R> = 0. It sends <**grant**> to P and R and awaits <**ack**> from them. (See Figure 4)
4. P receives <**grant**> from Q and sets *requests<P>* to 1. P sends <**ack**> to Q. (See Figure 4)
5. Q receives <**grant**> from R and sets *requests<Q>* to 0. It first sends <**ack**> to R, and then sends <**grant**> to P. (See Figure 5)
6. P receives <**grant**> from Q and sets *requests<P>* to 0. It sends <**ack**> to Q. (See Figure 6)
7. R receives <**ack**> from Q and P, it sends <**done**> to P. (See Figure 6)
8. Q receives <**ack**> from P, it sends <**done**> to P. (See Figure 6)
9. P receives <**done**> from Q and R, checks the value of *free<P>* and concludes that it is not deadlocked.

1.3.6 Correctness

Deadlock is detected if and only if the initiator node belongs to a cycle of the WFG. Every process forwards the message to each of its successors in the WFG. Therefore, in a bounded number of steps, the initiator process i receives the message and detects that it is deadlocked. If the initiator does not belong to the cycle, then it will never receive its own message, so deadlock will not be detected. [Ghosh2015]

1.3.7 Complexity

1. **Time Complexity:** The *Bracha-Toueg Deadlock Detection* has time complexity of 4 * d hops, where d is the diameter of a given WFG. [Kshemkalyani1994]
2. **Message Complexity:** In thhe *Bracha-Toueg Deadlock Detection*, at most four messages are sent over each edge, i.e., at most a total of 4e mes- sages are sent. The size of each message is a small constant number of bits. A node v of degree k needs o(k) bits of local storage, and spends o(k) time in local computation. [Bracha1987]

1.4 Implementation, Results and Discussion

1.4.1 Implementation and Methodology

We utilized the Python (version 3.12) scripting language and the Ad-Hoc Computing (adhoccomputing) library while implementing the Bracha-Toueg Detection Algorithm. We also employed the networkx library to generate various network topologies and the matplotlib library to visualize them. Each component in the topology can be the initiator for the deadlock detection algorithm. We implemented a function for which we can simulate processes requesting resources from one another. This function “send_request_to_component” is called for processes before starting the deadlock detection algorithm by sending “DETECTDEADLOCK” event to the initiator process. Once we started the deadlock detection algorithm, we first take the Lai-Yang snapshot of the initiator process. We are only interested in the exchanged “REQUEST” messages for deadlock detection, so we ignore other types of exchanged messages. Once the initiator process completes the Lai-Yang snapshot algorithm, it uses its previously recorded state to understand what processes it is waiting to receive resources and what processes waiting for it to grant resources. This means that, the initiator process computes a WFG graph for itself. Then, it continues with notifying the processes in “OUT” and waits for receiving DONE message from all of them. Once it receives DONE from all processes in OUT, it checks whether it is deadlocked by looking at its local variable “free”. An importing to mention here is that, the grant procedure is embedded inside the notify. This enables a process to need to receive ACKNOWLEDGE messages from all processes in IN, before sending any “DONE” messages.

If there is a cyclic dependency in the WFG, then we should expect that, the variable “free” for the initiator process can never be True. If there is no cyclic dependency in the WFG involving the initiator process, than we see that the variable “free” becomes True because the initiator process is able to grant some resources to other processes.

We implemented the Lai-Yang Snapshot Algorithm and the Bracha-Toueg Deadlock Detection Algorithm by employing the pseudocode descriptions given by in [Fokking2013]. We used the same message types given in the descriptions to achieve the message passing between the components. The make the component who is the initiator of the basic algorithm send itself “DETECTDEADLOCK” message to trigger the algorithm. Depending on whether the initiator process can set the variable “free” to True, the algorithm detects the deadlock. An important note here is that, waiting for DONE and ACKNOWLEDGE messages should not prevent receiving and sending GRANT and NOTIFY messages. Therefore, we increased the number of threads executing in a component to 3 to reflect the asynchronous wait operations.

1.4.2 Results

We designed two distinct scenarios to evaluate the message complexity of the Bracha-Toueg Deadlock Detection Algorithm. For the first scenario, we considered ring topologies with node counts as deadlock occurrence is guaranteed once each node passes a request to one of its neighbors in the same direction. In this scheme, the network topology guarantees that every node in the topology is part of the deadlock as well. Once each component sends a request to its neighbor, we start executing the algorithm through the initiator of the topology. After that, we measure the time elapsed until the initiator component detects the deadlock in the distributed system. Table 1 presents the elapsed time along with the number of control message components exchanged while running the algorithm. Note that we do not include the control messages of the Lai-Yang snapshot algorithm as it is not part of the deadlock detection algorithm.

Table 1: Message Complexity Analysis of Deadlock Detection Algorithm on a Ring Topology

Node Count	Number of Exchanged Control Messages	Time Elapsed Until Detection
5	12	0.5193710327148438
10	22	0.10675311088562012
20	42	0.11047983169555664
50	72	4.122158050537109
100	123	7.982053995132446

The plots below shows the relationship between the node count in the network with the time elapsed until detection and number of exchanged control messages.

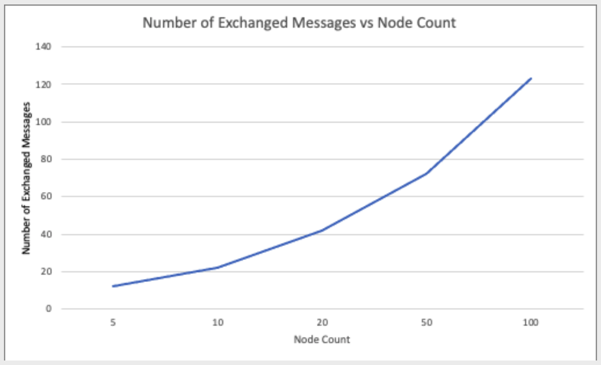

Fig. 7: The Relationship Between Node Count and Number Of Exchanged Messages

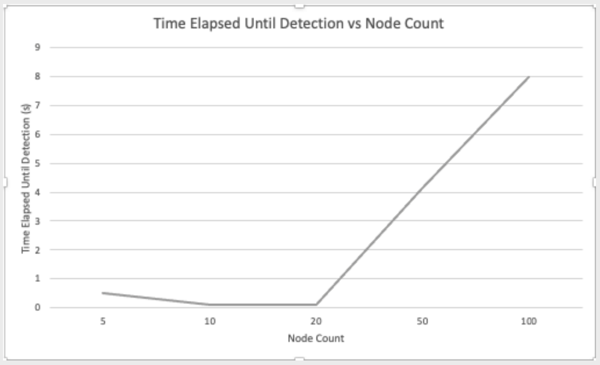

Fig. 8: The Relationship Between the Node Count and the Elapsed Time

In the second scenario, we considered a complete topology of 10 nodes. In such a topology, there are different cycles with different participating nodes. Before executing the algorithm, we found random cycles with 9, 8, 7, 5, and 3 nodes and made each component send a request to its neighbors in the same direction. After that, we start executing the algorithm through a participating node of the cycles. Table 2 presents the time elapsed until an initiator detects the deadlock of the same cycle in the system.

Table 2: Message Complexity Analysis of Deadlock Detection Algorithm on a Complete Topology

Cycles in the Topology	Initiator Node	Time Elapsed Until Detection
[0, 2, 3, 4, 5, 6, 7, 8, 9]	7	0.1082148551940918
[0, 1, 2, 3, 6, 7, 8, 9]	8	0.10948896408081055
[2, 3, 4, 5, 6, 7, 8]	2	0.7286171913146973
[0, 2, 4, 8, 9]	4	0.5165529251098633
[0, 3, 6]	6	0.31595897674560547

1.4.3 Discussion

We conducted two separate experiments to analyze the message and time complexity of an algorithm. To distinguish between the experiments, we did not modify the underlying computation. Instead, we made each process send a request to one of its neighbors in the same direction. By not changing the underlying computation, we could observe the changes in elapsed time over different topologies.

During the experiments, we had to add delays between events because we observed that, in the absence of the delays, we could not see the exchange of all the messages we sent. Additionally, we simulated some processes waiting to acquire resources as requests running in our topologies to conduct experiments.

Since we were only interested in generating a Wait-For-Graph (WFG), we ignored other exchanged messages in the distributed system while taking the Lai-Yang Snapshot. As a result, the algorithm became heavily dependent on the custom event.

Despite the challenges, we observed that as the number of nodes participating in a cycle increased, the time it took to detect the cycle in the topology went up proportionally.

1.5 Conclusion

The Bracha-Toueg Deadlock Detection Algorithm is a reliable and adaptable solution for identifying and resolving deadlocks in distributed systems across various network topologies. We have explored the algorithm extensively, emphasizing its significance, implementation intricacies, and message complexity evaluation. Our analysis of the algorithm's message complexity has demonstrated its efficiency and practicality. By using the ad-hoc computing library, we have conducted experiments that show a message complexity of O(4E), which is in line with theoretical expectations. This high efficiency makes the algorithm suitable for real-world distributed systems. If the algorithm is included in the ad-hoc computing library, anyone can use it in their research. We discussed how we implemented the request mechanism in Section 1.4.3. In future work, we could approach the issue differently and change the implementation to reflect requests as not specific events but messages that belong to the algorithm having a deadlock in the system. We also mentioned that the Lai-Yang Snapshot implementation only considers the specific request messages and discards others as they do not relate to deadlock detection. Another approach would be considering all kinds of messages, but not storing the messages' contents would be more efficient and generic because what is in a message is irrelevant.

CHAPTER

TWO

ASSESSMENT RUBRIC

Your work and documentation will be assessed based on the following list of criteria.

2.1 Organization and Style

[15 points] The documentation states title, author names, affiliations and date. The format follows this style?

1. Structure and Organization: Does the organization of the paper enhance understanding of the material? Is the flow logical with appropriate transitions between sections?
2. Technical Exposition: Is the technical material presented clearly and logically? Is the material presented at the appropriate level of detail?
3. Clarity: Is the writing clear, unambiguous and direct? Is there excessive use of jargon, acronyms or undefined terms?
4. Style: Does the writing adhere to conventional rules of grammar and style? Are the references sufficient and appropriate?
5. Length: Is the length of the paper appropriate to the technical content?
6. Illustrations: Do the figures and tables enhance understanding of the text? Are they well explained? Are they of appropriate number, format and size?
7. Documentation style: Did you follow the expected documentation style (rst or latex)?

2.2 Abstract

[10 points] Does the abstract summarize the documentation?

1. Motivation/problem statement: Why do we care about the problem? What practical, scientific or theoretical gap is your research filling?
2. Methods/procedure/approach: What did you actually do to get your results?
3. Results/findings/product: As a result of completing the above procedure, what did you learn/invent/create? What are the main learning points?
4. Conclusion/implications: What are the larger implications of your findings, especially for the problem/gap identified?

2.3 Introduction and the Problem

[15 points] The problem section must be specific. The title of the section must indicate your problem. Do not use generic titles.

1. Is the problem clearly stated?
2. Is the problem practically important?
3. What is the purpose of the study?
4. What is the hypothesis?
5. Are the key terms defined?

2.4 Background and Related Work

[15 points] Does the documentation present the background and related work in separate sections.

1. Are the cited sources pertinent to the study?
2. Is the review too broad or too narrow?
3. Are the references/citation recent or appropriate?
4. Is there any evidence of bias?

2.5 Implementation and Methodology

[15 points] Does the documentation present the design of the study.

1. What research methodology was used?
2. Was it a replica study or an original study?
3. What measurement tools were used?
4. How were the procedures structured and the implementation done?
5. Were extensive exprimentations conducted providing not only means but also confidence intervals?
6. What are the assessed parameters and were they adequate?
7. How was sampling and measurement performed?

2.6 Analysis and Discussion

[15 points] Does the documentation present the analysis?

1. Did you collected enough and adequate data for analysis?
2. How was data analyzed?
3. Was data qualitative or quantitative?
4. Did you provide main learning points based on analysis and results?
5. Did findings support the hypothesis and purpose?

6. Did you provide discussion as to the main learning points?
7. Were weaknesses and problems discussed?

2.7 Conclusion and Future Work

[15 points] Does the documentation state the conclusion and future work clearly?

1. Are the conclusions of the study related to the original purpose?
2. Were the implications discussed?
3. Whom will the results and conclusions effect?
4. What recommendations were made at the conclusion?
5. Did you provide future work and suggestions?

CHAPTER

THREE

CODE DOCUMENTATION

BrachaToueg.BrachaToueg

3.1 BrachaToueg.BrachaToueg

Classes

class BrachaToueg.BrachaToueg.**BrachaTouegComponentModel**(*componentname*, *componentinstancenumber*, *context=None*, *configurationparameters=None*, *num_worker_threads=3*, *topology=None*, *child_conn=None*, *node_queues=None*, *channel_queues=None*)

on_receiving_detect_deadlock(*eventobj*)

This method triggers the Bracha-Toueg Deadlock Detection Algorithm by first taking a local snapshot of the component starting the algorithm.

on_receiving_take_snapshot(*eventobj*)

This method triggers the Lai-Yang Snapshot Algorithm

send_request_to_component(*component*)

This method sends a request to the given component in the arguments, simulating that this process requires some resource/communication from the given component REQUEST is part of the basic algorithm and independent of the detection algorithm.

on_receiving_request_from_component(*eventobj*)

This method receives the request by comparing the process sending the request's local snapshot and the process receiving the request. If process sending this request has already taken the snapshot, this process also takes it. If this process took its snapshot then, it proceeds to check the condition for termination

on_receiving_presnapshot(*eventobj*)

The component receiving the presnapshot control message starts to take its own local snapshot. After that, if the termination condition is satisfied, the component terminates the snapshot algorithm.

`on_message_from_bottom`(*eventobj*)

This function processes the message from top events by calling related functions according to the message's header.

`take_snapshot`()

This method takes a local snapshot of the component after sending presnapshot messages to all its outgoing channels by storing *self.incoming_channels*, *self.outgoing_channels*, *self.state* and *self.counter*.

`terminate_snapshot`()

This method notifies the component by calling the deadlock detection algorithm to proceed with since the WFG is computed.

`check_termination_condition`()

This function compares the number of incoming and outgoing messages of a given component with its incoming message set length and reaches a conclusion about the algorithm termination

`generate_message`(*messagetype*, *messagefrom*, *messageto*, *payload*)

This function generates a GenericMessage object given the type of the message, the payload of the message and the message source and destination

`notify`()

Process sends notify to all its outgoing messages and if it does not wait for any resources, it starts granting the resources. After receiving DONE messages from its outgoing channels, it checks the value of free and decides whether it is deadlocked.

`grant`()

Process that is able to grant resource to its incoming channels becomes a free process and waits for ACKNOWLEDGE message from its incoming channels before proceeding.

`on_receiving_notify`(*eventobj*)

Process receiving the notify message notifies its outgoing components if it is not notified earlier. After that it sends DONE message to the process sending the notify message.

`on_receiving_grant`(*eventobj*)

Process receiving the grant message grants resources to its incoming channels if it does not wait for any resources. After that, it sends ACKNOWLEDGE message to the process granting the resource to it.

`on_receiving_acknowledge`(*eventobj*)

`on_receiving_done`(*eventobj*)

`class BrachaToueg.BrachaToueg.BrachaTouegEventTypes`(*value*, *names=<not given>*, *values, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

`DETECTDEADLOCK = 'DETECTDEADLOCK'`

`class BrachaToueg.BrachaToueg.BrachaTouegMessageTypes`(*value*, *names=<not given>*, *values, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

`REQUEST = 'REQUEST'`

`NOTIFY = 'NOFIFY'`

`DONE = 'DONE'`

`GRANT = 'GRANT'`

`ACKNOWLEDGE = 'ACKNOWLEDGE'`

class `BrachaToueg.BrachaToueg.`**`LaiYangEventTypes`**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

`TAKESNAPSHOT = 'TAKESNAPSHOT'`

class `BrachaToueg.BrachaToueg.`**`LaiYangMessageTypes`**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

`PRESNAPSHOT = 'PRESNAPSHOT'`

class `BrachaToueg.BrachaToueg.`**`SnapshotState`**(*snapshot_recorded*, *counter*, *state*, *outgoing_channels*, *incoming_channels*)

Attention: For RST details, please refer to reStructuredText Documentation.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Fokking2013] Wan Fokkink, Distributed Algorithms An Intuitive Approach, The MIT Press Cambridge, Massachusetts London, England, 2013

[Bracha1987] G. Bracha and S. Toeug, “Distributed Deadlock detection”. Distributed Comput., vol. 2, pp. 127-138, 1987

[Kshemkalyani2008] Ajay D. Kshemkalyani, Mukesh Singhal, Distributed Computing: Principles, Algorithms and Systems, Cambridge Univeristy Press, New York, USA, 2008

[Kshemkalyani1994] A. D. Kshemkalyani and M. Singhal, “Efficient detection and resolution of generalized distributed deadlocks,” in IEEE Transactions on Software Engineering, vol. 20, no. 1, pp. 43-54, Jan. 1994,

[Knapp1987] E. Knapp, “Deadlock Detection in Distributed Databases”, ACM Computing Surveys, Volume 19, Issue 4, pp 303-328, 1987

[Ghosh2015] S. Ghosh, “Distributed Systems: An Algorithmic Approach, 2nd Edition”, CRC Press pp. 184, 2015

PYTHON MODULE INDEX

INDEX