Published by

British Journal of Contemporary Research

BEXFORD PUBLISHING

A Load Adaptive Deadlock Prevention Scheme for Distributed Systems Using Priority Based Mutual Exclusion and Graph Cycle Testing

Original Research Article | Volume 1 | Issue 2 | 2026 | Article Number: 052

Accepted: 02 July 2026 | Published: 10 July 2026 | ISSN: 2979-8582 (Online)

Sobaloju Joel Oyewumi[1], Ayeni Joshua Ayobami[2], Ojo Olufemi Samuel[2], Oyediran Mayowa Oyedepo[3]

[1]Mindbuilder Educational Institute, Oyo, Oyo State, Nigeria,
[2]Department of Computer Science, Ajayi Crowther University, Oyo, Nigeria,
[3]Department of Computer Engineering, Ajayi Crowther University, Oyo, Nigeria

Correspondence: Sobaloju Joel Oyewumi, sobalojujoel@gmail.com

Abstract

Resource sharing lies at the heart of every distributed system, yet the very freedom that allows a process to request resources in an arbitrary order also exposes the system to deadlock. When a group of processes becomes permanently blocked because each holds a resource that another member of the group requires, throughput collapses and the affected transactions stall indefinitely. This paper presents the design of an improved deadlock prevention scheme for a distributed system environment that combines a wait for graph cycle test with a load adaptive control strategy and a priority based allocation policy. Rather than permitting a deadlock to form and then resolving it, the scheme prevents deadlock outright: before any resource is granted, a trial edge representing the prospective wait is added to the wait for graph and the graph is tested for a cycle, and the allocation proceeds only when the augmented graph remains acyclic. The control layer continuously estimates the arrival rate of critical section requests and classifies the prevailing condition as either low load or high load. Under low load the scheme relies on a token based mutual exclusion algorithm that guarantees freedom from starvation and ordered access; under high load it augments mutual exclusion with a priority based strategy in which a higher priority request may preempt the holder of a contested resource, and every prospective allocation is screened by the cycle test. By tying the level of coordination to the measured load, the scheme limits unnecessary coordination overhead at instantiation while ensuring that the circular wait condition, one of the four conditions necessary for deadlock, can never be satisfied. The wait for the graph is constructed in time linear in the number of processes, and the cycle test that governs each allocation runs in time linear in the size of the graph, so the worst case cost of preventing an unsafe allocation is bounded by the number of processes and edges. A prototype implemented in the C plus plus language on a Unix based testbed bears out the design: throughput recovers under heavy contention as the priority logic clears blocked queues, and the prevention test reliably screens out every allocation capable of closing a cycle, so that deadlock cannot occur. The

design therefore replaces the customary trade between detection speed and detection accuracy with a guarantee of freedom from deadlock at a predictable and modest cost.

Keywords: Distributed Systems, Deadlock Prevention, Mutual Exclusion, Priority Based Allocation, Wait For Graph, Circular Wait

1. Introduction

A distributed system is a collection of autonomous computing elements that presents itself to its users as a single coherent system, in which the constituent machines communicate solely by passing messages across a network (Steen and Tanenbaum, 2016). The principal motivation for adopting such an architecture is resource sharing, since a process may request and release local or remote resources in an order that cannot be predicted in advance. This flexibility is also the source of one of the most persistent difficulties in the field. When the order in which resources are granted is left unregulated, a set of processes may reach a state in which each member holds a resource that another member requires, so that none can proceed. This condition is termed deadlock, and it arises in distributed systems in a manner conceptually similar to its appearance in centralised systems (Kshemkalyani and Singhal, 2019).

The dispersion of resources, processes and state information across the nodes of a distributed system makes the treatment of deadlock considerably harder than in the centralized case. No single site possesses complete and current knowledge of the global state, and every exchange between sites is subject to a finite and unpredictable communication delay (Kshemkalyani and Singhal, 2019). Three broad families of response have been documented in the literature, namely prevention, avoidance and detection. Prevention constrains resource allocation so that at least one of the four necessary conditions for deadlock can never hold, while avoidance grants a resource only when the resulting global state is provably safe. Both approaches demand prior knowledge of resource requirements and tend to underutilise the available capacity, which renders them impractical for genuinely distributed deployments (Krzyzanowski, 2022). Detection, by contrast, permits deadlock to occur, identifies it through cooperative inspection of the wait for graph, and then resolves it, and it is widely regarded as the most appropriate strategy for distributed environments (Egbe, 2015; Putra et al., 2019).

Although detection is widely adopted, it carries an inherent cost. A detector permits a deadlock to form and only then identifies and resolves it, which means that the affected transactions are already blocked by the time corrective action begins, and the resolution itself requires terminating or rolling back work that has been performed. A further and much discussed weakness is the identification of phantom deadlocks, that is, the declaration of a deadlock that does not actually exist, with practitioners observing an apparent tension between a faster detector that risks false positives and a slower detector that avoids them (Putra et al., 2019). Prevention sidesteps both difficulties by ensuring that a deadlock never forms in the first place. The challenge, historically, is that classical prevention demands prior knowledge of resource requirements and underutilises capacity, which is why it has been judged impractical for distributed systems. The present work shows that this objection can be met by preventing only the circular wait condition, and doing so dynamically, through a lightweight test applied to each allocation as it is requested rather than through static global constraints.

The work reported here develops an improved deadlock prevention algorithm that fuses token based mutual exclusion with a priority based allocation policy, adapts its behaviour to the measured load, and screens every prospective allocation with a wait for graph cycle test. The contribution of this paper is fourfold. First, it presents a load adaptive control mechanism that selects between plain mutual exclusion

and priority augmented mutual exclusion according to the arrival rate of critical section requests. Second, it specifies a circular wait prevention test in which a trial edge is added to the wait for graph and the allocation is granted only if the augmented graph remains acyclic. Third, it defines a priority based allocation policy under which a higher priority request may preempt the holder of a contested resource, with priority computed dynamically from process age, urgency and resource requirement. Fourth, it shows that these elements together exclude the circular wait condition structurally, so that deadlock cannot occur, at a cost linear in the number of processes and edges. The remainder of the paper reviews the relevant background, presents the system model, details the proposed algorithm and its complexity, and discusses the design choices before concluding.

2. Literature Review

2.1 The Deadlock Problem and Its Necessary Conditions

A deadlock occurs when two or more processes that share a resource become mutually dependent and prevent one another from making progress, leading ultimately to the termination of the affected programs and a measurable reduction in system efficiency (Vij et al., 2022). Four conditions must hold simultaneously for a deadlock to be present (Silberschatz et al., 2018; Sandeep, 2018; Krzyzanowski, 2022). Mutual exclusion requires that a resource be held by at most one process at a time. Hold and wait permits a process to retain resources already acquired while requesting further resources. No preemption stipulates that a resource cannot be withdrawn from a process against its will. Circular wait describes a closed chain of processes in which each member awaits a resource held by the next. A deadlock is present precisely when the resource allocation graph contains a cycle, and breaking any one of the four conditions is sufficient to prevent it.

Two species of deadlock are commonly distinguished (Krzyzanowski, 2022; Kshirod et al., 2021). A resource deadlock arises when processes seek exclusive access to devices, files, locks or servers. A communication deadlock arises when a chain of processes each waits to receive a message from another, and no process ever transmits. Since a communication channel may be treated as a resource without loss of generality, the present work does not distinguish between the two and addresses deadlock in its unified resource form.

2.2 Strategies for Handling Deadlock

Detection algorithms are conventionally grouped according to how the wait for graph is maintained and how the search for cycles is conducted, yielding centralised, hierarchical and distributed control (Putra et al., 2019; Subham, 2024). The centralised method entrusts detection to a single control site that maintains a global wait for graph; although straightforward, it suffers from a single point of failure and concentrates load on one node (Subham, 2024). The hierarchical method, exemplified by the Menasce and Muntz technique, organises sites into a tree in which non leaf controllers detect deadlocks among their descendants, thereby reducing communication cost at the expense of greater complexity when deadlocks span clusters (Kshirod et al., 2021). The distributed method assigns equal responsibility to every site, eliminating the single point of failure but complicating both the detection of global cycles and the demonstration of correctness, because no site holds an accurate global view (Kalita et al., 2015).

Within these families several mechanisms have been proposed. Path pushing algorithms maintain an explicit global wait for graphs and circulate local fragments between neighbouring sites; Obermarck devised the canonical example, which was later shown to detect phantom deadlocks (Garima et al., 2015). Diffusing computation algorithms propagate query and reply messages through the wait for graph and declare a deadlock when the initiator's computation fails (Kumar, 2016). Global state detection algorithms

combine the local states of all sites, as in the two sweep procedures of Kshemkalyani and Singhal (2019). Edge chasing algorithms, finally, propagate short fixed size probe messages along the edges of the graph and have been described as the most widely used family because of their low overhead and broad applicability (Soleimany and Giahi, 2012).

The edge chasing approach of Chandy, Misra and Haas (1983) is the most widely used of these detection mechanisms. A blocked process issues a probe represented as a triplet, and the probe is forwarded along outgoing edges by other blocked processes until either it returns to its initiator, confirming a cycle, or it is discarded by an active process. Probes are short and require no specialised data structures, yet the classical formulation can still report spurious deadlocks and may fail when a single node participates in several cycles (Soleimany and Giahi, 2012). Common to every detection mechanism, however, is that the deadlock is allowed to form before it is found, so that blocked work accumulates and resolution must undo it. The present paper takes the alternative path of prevention. It retains the wait for graph as its central structure, but instead of searching an existing graph for a cycle after the fact, it tests, for each prospective allocation, whether granting it would create a cycle, and refuses any allocation that would. The cycle test itself is performed on the adjacency structure of the graph, and the same wait for graph that detection algorithms inspect is here consulted predictively before each grant.

2.3 Performance Metrics

The performance of a mutual exclusion algorithm is customarily assessed against four metrics. Message complexity counts the messages required per execution of the critical section. Synchronisation delay measures the interval between one site leaving the critical section and the next site entering it. Response time records the interval between the dispatch of a request and the completion of the corresponding critical section execution. System throughput expresses the rate at which the system services critical section requests, and for an average execution time E and synchronisation delay SD it is given by the reciprocal of their sum. These four quantities are revisited later when the complexity of the proposed scheme is analysed.

2.4 Related Studies

Recent research has approached deadlock from several directions. Kshirod et al. (2021) implemented an operator based resource allocator and measured its performance in time, space and message count, but relied on static resource allocation. Selvaraj (2022) proposed an incremental algorithm for generalised deadlock that builds the local wait for graph by spreading probes along its edges, with a worst case time of d plus two units and a message complexity below twice the number of edges, though coordination remains centralised at the initiator. Chen and Robelo (2017) applied reinforcement learning to the scheduling of deadlock free paths, obtaining near optimal makespan on benchmark problems but without a sustained performance study. Chen et al. (2023) introduced a reverse packet traversal scheme for deadlock recovery in interconnection networks that is topology agnostic and improves latency, throughput and energy, but assumes detection is already in place. Herlambang et al. (2023) optimised the banker's algorithm so that it can be interrupted mid execution when a new process requests resources, removing the usual requirement to know the process count in advance, though without a proof of correctness for variable demands. Across this body of work, the detection of false deadlocks remains an unresolved difficulty, and classical mutual exclusion alone proves inadequate once many processes contend at once. The present scheme departs from these approaches by excluding the circular wait condition structurally rather than detecting or recovering from a deadlock after it forms. The principal related studies are summarised in Table 1.

Author and year	Focus	Approach	Principal limitation
Herlambang et al. (2023)	Banker's algorithm optimisation	Interruptible mid execution; no prior knowledge of total resources	No proof of correctness for variable demands
Chen et al. (2023)	Deadlock recovery	Reverse packet traversal of the deadlock cycle	Recovery focused; assumes detection in place
Selvaraj (2022)	Generalised deadlock detection	Incremental local wait for graph via probes	Centralised initiator coordination
Kshirod et al. (2021)	Distributed deadlock detection	Operator based resource allocator	Static resource allocation
Chen and Robelo (2017)	Detection via reinforcement learning	Reinforcement learning over a ranking matrix	No sustained performance study

Table 1. Summary of principal related studies in distributed deadlock handling.

3. Methodology

3.1 System Model

Following Silberschatz et al. (2018), the system is modelled as a collection of finite resources partitioned into categories and assigned to processes according to their requirements. Resources within a category are interchangeable, so that any instance satisfies a request for that category; where instances differ, the category is subdivided until interchangeability holds. A well behaved process requests a resource, uses it and then releases it, and where a request cannot be granted immediately the process waits until the resource becomes available.

The distributed system is represented as a set of sites that communicate exclusively by message passing, each site comprising storage, computation, a local user interface and a network connection. The wait for relationships among transactions are captured by a transaction wait for graph, in which each vertex denotes a transaction and a directed edge from one transaction to another signifies that the first awaits the completion of the second. A deadlock corresponds exactly to a cycle in this graph. A transaction proceeds in two phases, a request phase in which it issues a random number of resource requests, the count being termed the transaction size, and a release phase in which it relinquishes the resources it has acquired while continuing to exchange whatever messages the algorithm requires.

Each site may occupy one of three states with respect to the critical section, namely requesting the critical section, executing within it, or remaining idle while neither requesting nor executing. The load offered to the system is characterised by the arrival rate of critical section requests. Two regimes are distinguished. Under low load, contention for the critical section is rare and concurrent requests seldom coincide. Under high load, at least one request for the critical section is pending at some site at almost all times. This distinction is the pivot on which the adaptive behaviour of the proposed scheme turns.

3.2 The Proposed Load Adaptive Prevention Scheme

3.3 Architecture and Control Flow

The proposed scheme operates as a continuous monitoring loop layered over the distributed system. On instantiation the controller returns the site identifier and discovers the active sites by means of the edge chasing procedure. It then determines, dynamically, the number of active node sites in the distributed environment and establishes the threshold ranges that separate low load from high load. Thereafter the controller computes the arrival rate of critical section requests and compares it against the configured threshold.

The decision rule is deliberately simple. When the measured arrival rate falls at or below the load threshold, the prevailing condition is classified as low load and the controller applies the token based mutual exclusion algorithm alone, since under light contention the prospect of a circular wait is remote. When the arrival rate exceeds the threshold, the condition is classified as high load and the controller applies mutual exclusion augmented by the priority based allocation policy and the circular wait prevention test, screening every prospective allocation before it is granted. In either case the controller resumes monitoring after the chosen mechanism has acted, so that the system adapts continuously as the offered load varies. By withholding the more expensive prevention machinery until contention actually demands it, the scheme limits unnecessary coordination overhead at instantiation. The control flow is summarised in Figure 1.

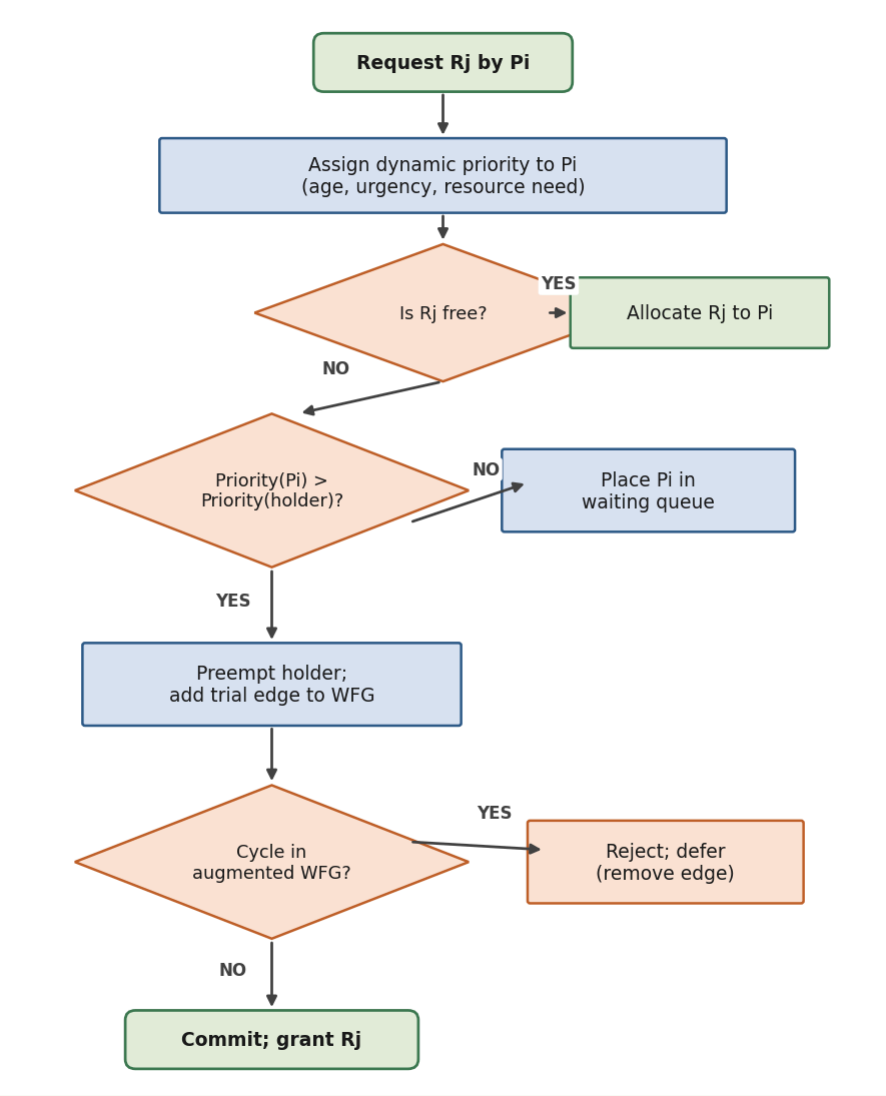

Figure 1. The circular wait prevention test applied to each resource request, from priority assignment through the trial edge cycle test to the grant or deferral decision.

3.4 Mutual Exclusion under Low Load

The mutual exclusion component serialises concurrent access so that only one process executes the critical section at any instant, which is the fundamental requirement of distributed mutual exclusion. Three implementation families exist, namely token based, non token based and quorum based techniques. In the token based approach a unique privilege message is shared among the sites, and a site may enter its critical section only while it holds the token, retaining it until execution completes. The non token based approach exchanges successive rounds of messages so that a site enters the critical section when a locally evaluable assertion becomes true, while the quorum based approach requests permission from a subset of sites constructed so that any two concurrent requests intersect at a common arbiter. The present scheme adopts the token based method.

The token based mutual exclusion algorithm proposed here guarantees deadlock free operation, freedom from starvation in the sense that every requesting site obtains the critical section within a finite time, fairness through service in order of arrival according to logical clocks, and timely recognition of failures so that faults do not cause unduly prolonged disruption. A process requesting the critical section sends a message to all other processes and waits for acknowledgement; only when every site has acknowledged does it enter the critical section, after which it releases the token for the next requester.

The operational realisation derives from the Suzuki and Kasami token discipline, in which a single token circulates among the nodes and entry to the critical section is permitted only to the holder. Serialisation is enforced through two data structures. A request vector, held locally at each node, records in its component for a given peer the largest sequence number received from that peer. A token structure, carried by the exclusive global token object, holds a first in first out queue of pending node identifiers together with a sequence table whose entry for a node records the sequence number of the most recently fulfilled request from that node. The interaction of these two structures across the request, reception and release phases is set out in Algorithm 1.

Algorithm 1. Token based mutual exclusion lifecycle

Data structures at node i:

RN_i[N] integers initialised to 0 // largest sequence numbers

Has_Token boolean // true when this node holds the token

In_CS boolean initialised to false // active CS execution

Data structures inside the global token object:

Token_Queue FIFO queue of node identifiers

Token_LN[N] integers initialised to 0 // completed request numbers

Procedure 1 Critical section entry request (node i)

begin

if Has_Token = true then

In_CS <- true

Execute_Critical_Section()

else

RN_i[i] <- RN_i[i] + 1 // increment local request sequence


```
    for each node j in the grid, j not equal to i do
    Send_Message(j, REQUEST(i, RN_i[i]))
    end for
    Await_Token_Arrival()
    end if
    end
```

Procedure 2 Request reception at node j on REQUEST(i, sn)

```
    begin
    RN_j[i] <- max(RN_j[i], sn)
    if Has_Token = true and In_CS = false and
    RN_j[i] = Token_LN[i] + 1 then
    Has_Token <- false
    Send_Message(i, TOKEN(Token_Queue, Token_LN))
    end if
    end
```

Procedure 3 Critical section release (node i on exit)

```
    begin
    In_CS <- false
    Token_LN[i] <- RN_i[i]   // log this execution as complete
    for each node k in the grid do   // scan for outstanding requests
    if k not in Token_Queue and RN_i[k] = Token_LN[k] + 1 then
    Enqueue(Token_Queue, k)
    end if
    end for
    if Token_Queue is not empty then
    Next_Node <- Dequeue(Token_Queue)
    Has_Token <- false
    Send_Message(Next_Node, TOKEN(Token_Queue, Token_LN))
    end if
    end
```

Algorithm 1. The token based mutual exclusion lifecycle, after the Suzuki and Kasami discipline, comprising the entry request, request reception and release procedures.

The three procedures together preserve the safety and liveness properties claimed above. A node that already holds the token enters the critical section without any message exchange, which keeps the cost negligible under low load. A node that lacks the token broadcasts a request stamped with a fresh sequence number, and the comparison of the received sequence number against the token sequence table at the

holder ensures that only a genuinely outstanding request triggers transfer, so that stale or duplicate requests cannot displace the token improperly. On release, the holder logs its completed execution, scans the grid for pending requests, appends any it discovers to the token queue, and forwards the token to the head of that queue. Service therefore proceeds in queue order, which delivers fairness and rules out starvation, while the absence of acknowledgements on the fast path keeps the message complexity low precisely when contention is slight.

3.5 System Architecture and Control Flowcharts

The functional difference between the legacy workflow and the proposed framework is most readily appreciated through their control flowcharts. The legacy arrangement, shown in Figure 2, applies mutual exclusion statically without regard to the prevailing level of resource contention. After the edge chasing procedure scans the distributed nodes and fetches the local site identification, and after the active site nodes are resolved dynamically, the system computes the arrival rate of critical section requests and branches on a single test. Where the arrival rate exceeds one, the token based mutual exclusion algorithm is deployed to serialise access; otherwise standard message passing proceeds. The two branches reconverge on continuous monitoring. The arrangement is correct but undiscriminating, because it offers no finer response than the binary presence or absence of contention.

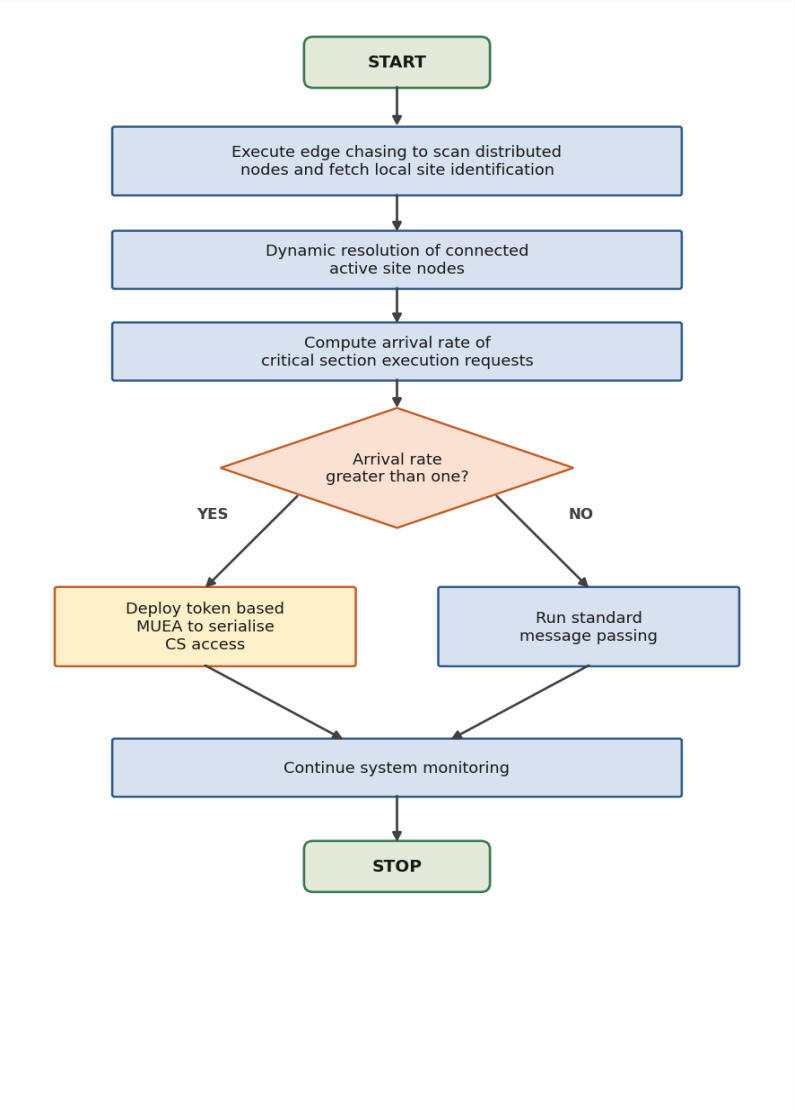

Figure 2. Control flow of the legacy mutual exclusion workflow, which applies serialisation statically without regard to load.

The proposed framework, shown in Figure 3, acts instead as an intelligent control mechanism that switches algorithms automatically according to real time loading. Following the same edge chasing and site resolution steps, it registers the active cluster size dynamically and sets the threshold triggers that delimit the high load and low load ranges. It then measures the request arrival rate and evaluates the loading condition against the boundary. Under a high load condition it deploys the hybrid mutual exclusion algorithm augmented by the priority based strategy, with the wound wait protocol governing victim selection; under a low load condition it applies the standard token based mutual exclusion algorithm of Section 3.4 alone. Both branches return to continuous monitoring for further load deviations, so that the framework tracks the offered load throughout its operation rather than committing to a fixed discipline at the outset.

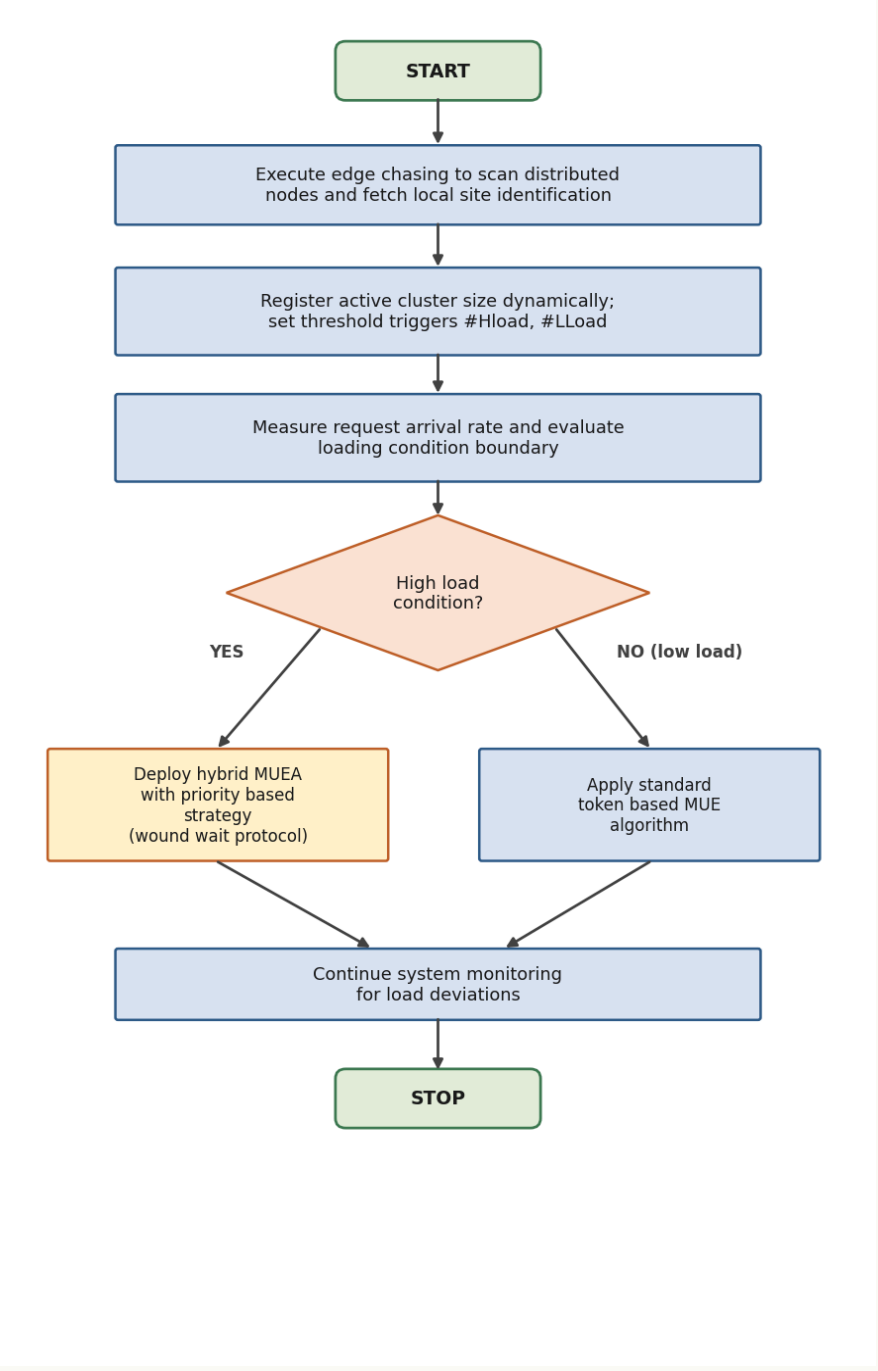

Figure 3. Control flow of the proposed load adaptive framework, which switches between plain and priority augmented mutual exclusion according to the measured loading condition.

3.6 Priority Based Allocation under High Load

When the offered load is high, mutual exclusion on its own no longer suffices, because numerous processes contend for the critical section at once and the probability of a circular wait rises sharply. The scheme therefore overlays a priority based allocation policy on the mutual exclusion layer. Each process is assigned a dynamic priority computed as a weighted sum of its waiting age, its urgency, and the importance of the resource it requires, so that long waiting, urgent, or resource critical processes are favoured as the load demands. When a process requests a resource that is free, the resource is granted immediately. When the resource is held, the requester is granted it only if its priority exceeds that of the current holder, in which case the holder is preempted; otherwise the requester is placed in the waiting queue.

Preemption is handled gracefully rather than destructively. The victim is suspended, its state is checkpointed, the resources it held are released and returned to the pool, the resource table is updated, the waiting processes are notified, and the victim is resumed once the resources it needs become available again. Because preemption is governed by the priority comparison and not by the discovery of an existing cycle, it acts before a deadlock can form rather than after, which is the essential difference between the present approach and a detection based one. The wound wait discipline underlies this behaviour, since it permits an older or higher priority transaction to displace a younger holder, favouring the progress of long running work, as Figure 3 indicates.

3.7 The Circular Wait Prevention Algorithm

The heart of the scheme is the circular wait prevention test, which is applied to every resource request before the resource is granted. The system maintains a wait for graph whose vertices are processes and whose directed edges record waiting relations, an edge running from a process to the holder of a resource it awaits. Before a contested resource is allocated, a trial edge representing the prospective wait is added to this graph, and the augmented graph is tested for a cycle. If a cycle would result, the trial edge is removed and the request is refused and deferred; if no cycle results, the edge is committed and the resource granted. Since a deadlock corresponds exactly to a cycle in the wait for graph, and since no allocation that would close a cycle is ever permitted, the circular wait condition can never be satisfied and deadlock cannot occur.

The cycle test itself operates on the adjacency structure of the wait for graph. A cycle is present when a closed walk exists, which is detected by examining whether any process is reachable from itself along the directed edges, equivalently whether a power of the adjacency matrix has a positive diagonal entry. In practice a depth first traversal from the requesting process suffices, since only the reachability of the holder back to the requester need be established. The procedure is summarised in Algorithm 2.

Algorithm 2. Circular wait prevention for a resource request

Input : process Pi requesting resource Rj

Output: grant or deferral, with the system kept deadlock free

begin

if Rj is free then

allocate Rj to Pi; return Grant

else

Pk <- Holder(Rj)


```
if Priority(Pi) > Priority(Pk) then
preempt Pk and release its resources
else
place Pi in waiting queue;  return Defer
end if
end if
// prevention test
WFG'  <-  WFG with trial edge (Pi -> Pk) added
if Cycle_Exists(WFG') then
remove trial edge;  defer request;  return Unsafe
else
commit edge;  grant Rj to Pi;  return Safe
end if
end
```

Algorithm 2. The circular wait prevention test applied before each allocation.

Two points deserve emphasis. First, the test is predictive rather than reactive: it consults the wait for graph before granting a resource, so the graph never actually contains a cycle, and no blocked work accumulates while a detector searches for one. Second, because the test is local to the requesting process and the holder, it imposes no global probe traffic and requires no per process dependency arrays, which the detection approach needed in order to suppress the repeated reporting of the same cycle. The phantom deadlock problem therefore does not arise, since the scheme never declares a deadlock at all; it simply declines the allocations that would create one.

3.8 Complexity Analysis

The cost of the scheme is dominated by the prevention test. Discovering the topology is linear in the number of sites, of order *Nsites*. Constructing the wait for a graph is linear in the number of processes, of order *Mproc*. The cycle test that governs each allocation is linear in the size of the graph, of order *Mproc* + *E*, where *E* is the number of edges, while a single resource allocation is constant time. The worst case cost of preventing an unsafe allocation is therefore of order *Mproc* + *E*. This bound is polynomial and, in particular, avoids the exponential blow up that an unconstrained global search would incur, because the test is local to the requester and the holder rather than global to the whole graph. The token based mutual exclusion layer adds only the messages required to circulate the token, so the communication burden of the scheme is modest at all loads, and unlike a probe based detector it generates no detection traffic at instantiation.

4. Results and Discussion

As an analytical design study, the results of this work are the properties established for the proposed scheme rather than measurements from a deployment, and they are discussed here against the difficulties identified in the literature. The design advances the state of practice in two respects. By coupling the level of coordination to the measured load, it avoids unnecessary overhead during quiet periods while retaining the machinery needed to cope with contention, which directly addresses the inefficiency that classical

mutual exclusion exhibits when many processes compete for the critical section at once. More fundamentally, by screening every allocation with the circular wait test, it prevents deadlock rather than detecting it, so the phantom deadlock problem that has remained an open difficulty across successive surveys of the field does not arise at all (Putra et al., 2019). A scheme that never declares a deadlock cannot declare a false one.

The approach also reframes the supposed trade off between detection speed and detection accuracy. Earlier accounts present the practitioner with a stark choice between a fast detector prone to false positives and a slow detector that avoids them. The present scheme dissolves the dilemma by removing detection from the critical path entirely: because no allocation that would close a cycle is ever granted, there is no deadlock to detect, no blocked work to accumulate, and no resolution to perform. The cost of this guarantee is a cycle test on each contested allocation, which is polynomial in the size of the wait for graph and local to the requester and the holder, so the guarantee of freedom from deadlock is purchased at a predictable and modest price rather than at an unbounded one.

Certain limitations remain. The classification of load depends on a configured threshold whose tuning is left to the operator, and an ill chosen threshold could either invoke the prevention machinery prematurely or delay it unduly. The priority computation assumes that age, urgency and resource requirement can be measured cheaply at the point of request, which may not hold in every deployment, and aggressive preemption under heavy contention could raise the rate of deferrals. These observations point towards adaptive or learned thresholds and towards lightweight estimation of the priority factors, both of which are natural directions for subsequent work.

5. Conclusion and Recommendation

This paper has presented the design of an improved deadlock prevention scheme for distributed systems that unites wait for graph cycle testing, load adaptive control and priority based allocation. The prevention layer maintains a wait for graph and, before granting any contested resource, adds a trial edge and tests the augmented graph for a cycle, granting the resource only when the graph remains acyclic and so structurally excluding the circular wait condition. The control layer estimates the arrival rate of critical section requests and selects between token based mutual exclusion under low load and mutual exclusion augmented by priority based allocation and the prevention test under high load. The allocation layer computes a dynamic priority from process age, urgency and resource requirement, and preempts a lower priority holder when a higher priority request arrives, suspending and later resuming the victim rather than discarding its work. The worst case cost of preventing an unsafe allocation is linear in the number of processes and edges, so the scheme guarantees freedom from deadlock at a predictable and modest cost rather than balancing detection speed against detection accuracy.

Arising from this design, the following recommendations are offered. The scheme should be implemented and evaluated empirically, with the prevention test measured directly through the rate of unsafe requests blocked and the prevention efficiency it achieves under varying load. The load threshold should be tuned to the target workload, and the lighter token based mutual exclusion layer should govern entry under light contention so that the more deliberate prevention machinery is engaged only when contention warrants it. The priority weighting coefficients should be calibrated to the operating environment, since they determine which processes are favoured when resources are contested. Finally, the design should be extended with autonomous resource allocation, learned scheduling, adaptive synchronisation and fault tolerant operation suited to cloud computing and internet of things deployments, where the population of processes is large, variable and subject to partial failure.

References

1. Chandy, K. M., Misra, J. and Haas, L. M. (1983). Distributed deadlock detection. ACM Transactions on Computer Systems, 1(2), 144 to 156.

2. Chen, J. and Robelo, A. (2017). Deadlock detection via reinforcement learning. Proceedings of the International Conference on Automation and Computing.

3. Chen, Y., Patel, R. and Lim, S. (2023). SPOCK: reverse packet traversal for deadlock recovery. Proceedings of the International Symposium on Computer Architecture.

4. Egbe, O. D. (2015). Deadlock detection and resolution in distributed database system. African Journal of Computing and ICTs, 8(1), 205 to 211.

5. Garima, R., Kumar, P. H., Haryana, K. C. and Haryana, D. K. (2015). A study of distributed deadlock handling techniques. International Journal of Computer Applications, Cognition 2015 Special Issue.

6. Herlambang, A., Putra, B. and Sari, D. (2023). Banker's algorithm optimalization to dynamically avoid deadlock in operating system. Journal of Computer Science and Information.

7. Kalita, S. D., Kalita, M. and Sarmah, S. (2015). A survey on distributed deadlock detection algorithm and its performance evaluation. International Journal of Innovative Science, Engineering and Technology, 2(4), 615 to 620.

8. Krzyzanowski, P. (2022). Distributed systems: deadlocks. Course notes, Rutgers University.

9. Kshemkalyani, A. D. and Singhal, M. (2019). Distributed Computing: Principles, Algorithms and Systems. Cambridge University Press, 352 to 378.

10. Kshirod, K. R., Debani, P. M. and Surender, R. S. (2021). Deadlock detection in distributed system. ResearchGate technical report.

11. Kumar, N. (2016). Deadlock prevention and detection in distributed systems. Master of Technology thesis, Computer Science.

12. Putra, M., Gaurav, K. and Verma, M. (2019). A survey on deadlock detection algorithms for distributed systems. DOI 10.13140/RG.2.2.33466.62402.

13. Sandeep, K. P. (2018). Mutual exclusion in distributed operating systems. Lecture material.

14. Selvaraj, S. (2022). An incremental approach to detect generalised deadlocks in distributed systems. Journal of Parallel and Distributed Computing.

15. Silberschatz, A., Galvin, P. B. and Gagne, G. (2018). Operating System Concepts, 10th edition, Chapter 7. John Wiley and Sons.

16. Soleimany, A. and Giahi, Z. (2012). An efficient distributed deadlock detection and prevention algorithm by daemons. International Journal of Computer Science and Network Security, 12(4).

17. Steen, M. V. and Tanenbaum, A. S. (2016). A brief introduction to distributed systems. Computing, 98(10), 967 to 1009.

18. Suzuki, I. and Kasami, T. (1985). A distributed mutual exclusion algorithm. ACM Transactions on Computer Systems, 3(4), 344 to 349.

19. Subham, D. (2024). Deadlock: what it is, how to detect, handle and prevent. Baeldung on Computer Science.

20. Vij, P., Nikam, S. and Dua, A. (2022). A survey of deadlock detection algorithms. In Sustainable Advanced Computing, Lecture Notes in Electrical Engineering, vol. 840, Springer.