6.1 Preamble

In OODS, the reusable data sources i.e. databases are mapped onto the objects. Hence, objects are viewed as resources here. To preserve the consistency of the objects, concurrency control is applied on objects and as a consequence live locks and deadlocks might occur.

The structure of the object oriented system is defined using static structural diagram namely class diagram or class lattice. From the class diagram, objects participating in the system, their attributes, methods and relationships with other objects in the system can be inferred. The transactions call the methods of these objects to satisfy their request. Hence, the dependency among the transactions can be inferred from the class diagram. Deadlocks can be handled by prevention, avoidance and detection and resolution. Deadlock prevention is easier to implement than deadlock detection in distributed systems. Deadlock prevention using resource ordering is generic than other types of Deadlock Prevention Algorithm (DPA) as it does not impose any constraints on the nature of resources.

Hence, our objectives are namely: - proposing resource-ordering policy for objects and proposing a deadlock prevention algorithm using the proposed resource ordering policy for resources i.e. objects in OODS. As mentioned earlier, resource ordering technique is not new. The novelty in the proposed algorithm lies in defining the resource ordering policy by exploiting the dependency among objects participating in the domain. The dependency of the objects with other objects can be inferred by their relationship with other objects. The possible relationships between objects can be inheritance, association and aggregation. Section 2.2 shows the dependency existing among objects in inheritance, association and aggregation.

The proposed algorithm also alleviates the problem of starvation in poverty and starvation in wealth by framing a policy for access ordering of transactions rather than using only timestamp ordering. Hence, by combining resource ordering with transaction access ordering, the proposed scheme prevents deadlock as well as starvation.
Deadlock detection and resolution algorithms are also available for distributed systems. Probe based distributed deadlock detection and resolution algorithm is one of the popular DDDR algorithms widely used because of its simple mechanism. The initiator, usually one of the transactions involved in the circular wait, send probe messages along the dependency edges of the Global Wait-For-Graph. When the probe comes back to the initiator, it indicates the presence of deadlock. Then resolution phase is initiated to choose a victim transaction and abort it.

The Probe based DDDR algorithm has two limitations namely: It cannot work in faulty environment and it requires a separate resolution phase to identify a victim and abort it. If the probe does not come back to the initiator, it could be because of live lock or faulty environment. If it is due to live lock, then the initiator can wait for finite time before sending the request message. If it is due to faulty environment, the probe might have been lost and initiator will not be aware of it. It will assume that the delay is due to live lock and wait. Providing fault tolerance is not a function of DDDR. But the initiator must always know the status of the probe in all cases.

The victim transaction can be identified while sending the probe instead of spending time on resolution algorithm to resolve the victim. If the victim can be dynamically selected at every site and included as part of the probe, then the probe will contain the final victim when it reaches the initiator. Then a message can be sent to kill the victim.

In section 6.2, a deadlock prevention algorithm based on resource ordering is proposed using the object semantics. In section 6.3, a modified fault informant probe based algorithm is proposed using colored probes. In section 6.4, the existing deadlock resolution algorithms are explored to identify the system parameters favored by the transaction attributes. A weight based victim selection algorithm is proposed to select the victim dynamically based on the choice of the desirable system parameters like throughput, resource utilization, fairness etc.

6.2 Deadlock prevention Algorithm for OODS

OODS follows AND request model as transactions need all the resources before execution. The resource ordering policy decides how the resources are ordered. The transactions are expected to request for the resources from smaller IDs to larger IDs. Then it is ideal to assign smaller resource IDs to independent resources and higher resource IDs to resources dependent on these independent resources. The proposed
The resource ordering technique is based on the dependency among the objects participating in the system. In OODS, the dependency among objects is based on their relationship with other objects as seen in section 2.2. Then dependency can be categorized based on

1. Relationship between object and class.
2. Objects related by inheritance.
3. Objects related by aggregation or composition.
4. Objects related by association.

The objective for resource ordering is to promote concurrency and it is based on granularity and degree of dependency. When the granularity size varies from coarse to fine, the concurrency increases. So lower resource IDs are given for fine granules and higher resource IDs are given for coarse granules. As mentioned earlier, the dependency is based on the relationships. As the dependency becomes more as given in section 2.2, number of objects needed is more. This reduces the concurrency. Based on these observations and constraints mentioned in section 2.2, partial ordering is done on each of the case mentioned above, and then total ordering of all the objects in a system is done.

In section 6.2.1 the resource ordering policy is framed and its formal model is given using predicate calculus in section 6.2.2. In section 6.2.3, system model is described. In sections 6.2.4 and 6.2.5, DPA modules and the algorithm are explained. In section 6.2.6, the access ordering rules proposed to alleviate the problem of starvation are listed. In sections 6.2.7 and 6.2.8, informal and formal proofs for the proposed DPA are given. Section 6.2.9 gives the summary.

6.2.1 Resource ordering technique

*Partial ordering on a class and its objects:*

The transactions can request for resources in two granularities namely single object in a class or all the objects in a class. If all the objects are requested, the lock is applied on the class rather than on the object to minimize the number of locks. Simultaneous request to both the cases is not allowed to maintain consistency. In this situation, two possibilities exist:

**Case 1:** Objects are assigned lower resource IDs than their class.
**Case 2:** Classes are assigned lower resource IDs than its objects.
In case 2, transactions executing class methods and requesting all the objects will have higher priority than transactions executing instance (object) method and requesting single object.

Example 1:

Let A be a class. Let a1, a2 and a3 be its objects.
Let T1 requests A. i.e., T1 requests \{a1, a2, a3\}.
(i.e. all objects in the class. Hence, the class itself is locked as per point (1) in 2.2)
Let T2 requests \{a1\}.
Let T3 requests \{a3\}.

Then concurrent execution of these requests proceeds as follows, if case 1 is considered. Both T2 and T3 are executed first and T1 is executed afterwards.

**Case 1**

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
<th>Transaction T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for a1,a3</td>
<td>Get a1 Execute a1</td>
<td>Get a3 Execute a3</td>
</tr>
<tr>
<td></td>
<td>Release a1</td>
<td></td>
</tr>
<tr>
<td>Get a1,a2,a3 Execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Release a1,a2,a3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Case 2**

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
<th>Transaction T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get a1,a2,a3 Execute a1</td>
<td>Wait for a1</td>
<td>Wait for a3</td>
</tr>
<tr>
<td>Execute a1,a2,a3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Release a1,a2,a3</td>
<td>Get a3 Execute a3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Release a1</td>
<td></td>
</tr>
</tbody>
</table>

If case 2 is considered for the same scenario, then execution will be as follows. Both T2 and T3 are blocked, while T1 is executed first. Here case 1 improves concurrency and hence improves the throughput of the system. Hence, case 1 is better.
than case 2. It can be justified as follows. Hence case 1 is better than case 2. Then the
rule can be defined as follows.

**Rule 1:** For all objects $O$ belonging to class $C$, resource IDs of objects $O$ should be
less than resource ID of their class $C$.

(Note: Transactions will request either only one or all objects, ordering among the
objects of a class is not necessary)

**Partial ordering on base class and its inherited sub classes:**

Here also resource IDs can be assigned in two ways:

- **Case 1:** Base Classes are assigned lower resource IDs than their subclasses.
- **Case 2:** Sub classes are assigned lower resource IDs than their base classes.

By definition of inheritance, attributes of base class are included as attributes of
subclass. The definition and implementation of template member functions and the
definitions of hook methods of base class are also included in the subclass. Hook
methods are allowed for overriding in subclass.

**Example 2:**

Let A and B be base classes. Let a1, a2 be instances of A and b1, b2 be instances
of B. Let C be a subclass inherited from A and B. Let c1, c2 be instances of C. Let a1
and b1 be associated base class objects for subclass object c1. Similarly, a2 and b2 be
associated base class objects for c2.

Let T1 requests {A}; T2 requests {B}; T3 requests {C}.

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
<th>Transaction T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get A</td>
<td>GET B</td>
<td>Wait for A, B, C</td>
</tr>
<tr>
<td>Execute</td>
<td>Execute</td>
<td>Get A, B, C</td>
</tr>
<tr>
<td>Release A</td>
<td>Release B</td>
<td>Execute</td>
</tr>
</tbody>
</table>

Let T1 requests {A}; T2 requests {B}; T3 requests {C}.

Then the resource set for T3 = \{A, B, C\}. (By point 2 in 2.2)
The resource set of T1 = \{a1, a2\}. (By point 1 in 2.2)
The resource set of T2 = \{b1, b2\}. (By point 1 in 2.2)
The resource set of T3 = \{a1, a2, b1, b2, c1, c2\} (by point 1 in 2.2)

**Case 1**
### Case 2

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
<th>Transaction T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for A</td>
<td>Wait for B</td>
<td>Get A, B, C</td>
</tr>
<tr>
<td>Get A</td>
<td>Get B</td>
<td>Execute</td>
</tr>
<tr>
<td>Execute</td>
<td>Execute</td>
<td>Release A, B, C</td>
</tr>
<tr>
<td>Release A</td>
<td>Release B</td>
<td></td>
</tr>
</tbody>
</table>

If case 1 is considered, then T1 and T2 get more priority than T3. Then concurrency is improved in case 1 than case 2. Therefore, resource IDs for base classes (BC) should be less than resource IDs of sub classes (SC) and represented as follows.

**Rule 2:** For all base classes BC, if a sub class SC is inherited from BC, then resource ID of base class BC should be less than the resource ID of subclass SC.

(Note: Since a transaction will request only one of the sub classes at a time, the ordering among sub classes of a parent class is not necessary.)

**Partial ordering on component objects and their composite object:**

Here also resources can be ordered in two ways:

**Case 1:** Component objects are assigned lower resource IDs than their composite object.

**Case 2:** Composite object is assigned lower resource ID than its component objects.

Aggregation is an object relationship. By definition of aggregation, component objects are part of composite object. Composite object avails the service of component objects, to satisfy the transaction request.

Example 3:

Let A and B be component classes. Let a be an instance of A and b be an instance of B. C is a composite class constituted from A and B. c is an instance of C. Let a and b be associated component class objects for composite class object c.

Let T1 requests \{a\}; Let T2 requests \{b\}; Let T3 requests \{c\}.

Then the resource set for T3 = \{a, b, c\} \hspace{1cm} (By point 3 in 2.2)

If case 1 is considered, T1 and T2 get priority over T3. As their resource sets are mutually exclusive, they can be concurrently executed. If case 2 is considered, T3 gets priority over T1 and T2. Case 2 gives higher priority to dependent classes over independent classes, where as case 1 improves concurrency.
Case 1

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Transaction</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>Get a</td>
<td>Get b</td>
<td>Wait for a, b</td>
</tr>
<tr>
<td>Execute</td>
<td>Execute</td>
<td>:</td>
</tr>
<tr>
<td>Release a</td>
<td>Release b</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get a, b, c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Execute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Release a, b, c</td>
</tr>
</tbody>
</table>


Case 2

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Transaction</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>Wait for a</td>
<td>Wait for b</td>
<td>Get a, b, c</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>Execute</td>
</tr>
<tr>
<td>Get a</td>
<td>Get b</td>
<td>Release a, b, c</td>
</tr>
<tr>
<td>Execute</td>
<td>Release b</td>
<td></td>
</tr>
</tbody>
</table>

Therefore, the partial ordering in the case of composition relationship among component class (CC) and composite class (CM) will be,

**Rule 3:** For all composite classes CM, if a component class CC is a part of CM, then resource ID of component class CC should be less than the resource ID of composite class CM.

(Note: Again, the ordering among component objects is immaterial, since they will be accessed mutually exclusively).

**Partial ordering of associative objects and their associated objects:** Association relationship is also an object relationship. Aggregation relationship is a subset of association. By extending rule 3 to association, the partial ordering among associative objects (TO) and their associated objects (AO) will be as follows:

**Rule 4:** For all associative class (TC), if an associative class (TC) is associated with associated class (AC), then resource ID of associative class TC should be less than associated class AC.

**Total resource ordering:**

The partial ordering of resources defined above is applicable when objects are related by only one relationship. However, in a business domain, objects may have complex relationships by having combination of above relationships. Hence, total
ordering of resources that can have a combination of above relationships has to be defined. By combining the partial ordering rules proposed earlier, the total ordering of resources can be defined as follows:

**Case 1:** When a class has inheritance relationship

Here the ordering needs to be done on base class, its objects, sub class and its objects. This can be done by combining rule 1 and 2 in the previous section. There are two options here

**Option 1:** Resource ID of Base Class Objects < Resource ID of Base Class < Resource ID of Sub Class Objects < Resource ID of Sub Class.

**Option 2:** Resource ID of Base Class Objects < Resource ID of Sub Class Objects < Resource ID of Base Class < Resource ID of Sub Class

Here all the objects are given lower IDs and then their classes are given higher IDs. From examples 1 and 2, it is obvious that option 2 has higher concurrency than option 1. Hence, the base class objects and sub class objects are given lower IDs and their classes are given higher IDs.

**Rule 5:** For all base class objects BCO belonging to BC and all sub class objects SCO belonging to SC, if a sub class SC is inherited from base class BC, then Resource ID (BCO) < Resource ID (SCO) < Resource ID (BC) < Resource ID (SC).

**Case 2:** When a class has aggregation relationship

There are two options here

**Option 1:** Resource ID of Component Class Objects < Resource ID of Component Class < Resource ID of Composite Class Objects < Resource ID of Composite Class

**Option 2:** Resource ID of Component Class Objects < Resource ID of Composite Class objects < Resource ID of Component Class < Resource ID of Composite Class

Here by examples 1 and 3, it is clear that option 2 is better than option 1. Hence component objects and composite objects are given lower IDs and their classes are given higher IDs.

**Rule 6:** For all component class objects CCO belonging to CC and all composite class objects CMO belonging to CM, if a component class CC is a part of composite class CM, then Resource ID (CCO) < Resource ID (CMO) < Resource ID (CM) < Resource ID (CM).

**Case 3:** When a class has association relationship

There are two options here
Option 1: Resource ID of Associative Class Objects < Resource ID of Associative Class < Resource ID of Associated Class Objects < Resource ID of Associated Class

Option 2: Resource ID of Associated Class Objects < Resource ID of Associative Class objects < Resource ID of Associated Class < Resource ID of Associative Class

It is clear that option 2 is better than option 1 by extending examples 1 and 3 to association. Hence, associative objects and associated objects are given lower IDs and their classes are given higher IDs.

Rule 7: For all associative class objects TCO belonging to TC and all associated class objects ACO belonging to associated class AC, if an associative class TC is associated with associated class AC, then Resource ID (ACO) < Resource ID (TCO) < Resource ID (AC) < Resource ID (TC).

Case 4: When a class has inheritance, association and aggregation relationships:

A class can have all the relationships in any order. For example, an inherited sub class can be component object of another class. Similarly, a class can be inherited from composite object class. It implies that the class relationships can be in any order. Hence ordering cannot be based on relationship alone. The class diagram is partitioned horizontally by various levels. In a class diagram, as the level increases, dependency increases and concurrency decreases. Concurrency decreases, because more no of resources are required for a transaction requesting high-level object or class to maintain consistency. Hence, lower resource IDs can be assigned for lower level, and higher resource IDs are assigned for higher level.

Rule 8: For all objects OA belonging to A, for all objects OB belonging to B, for all classes A in level i and for all classes B in level j of the class diagram where i < j, then Resource ID (OA) < Resource ID (OB) < Resource ID (A) < Resource ID (B).

The above rules can be formally defined using predicate calculus.

6.2.2 Formal model for resource ordering using predicate calculus

Let the class diagram representing the domain be the Universal set U. U is a collection of classes C representing the domain. Let O be the collection of objects instantiated from the classes C. Then U can be represented as U(C (O)). The classes are related to each other by inheritance, aggregation and association relationships.

Let BC be base class and SC be sub class related by inheritance. Let BCO be base class objects and SCO be subclass objects. Let CC be component class and CM be
composite class related by aggregation. Let CCO be component class objects and CMO be composite class objects. Let TC be associative class and AC be associated class related by association. Let TCO be associative class objects and ACO be associated class objects.

Let Rid (X) be a function that returns resource id for a resource X. Let Inherit-from (SC, BC) be a predicate that means SC is inherited from BC. Let Part-of (CC, CM) be a predicate that means CC is a part of CM. Let Associated-with (TC, AC) be a predicate that means TC is associated with AC. Let Level (Yi) be a predicate that means class Y is in level i. Let LL(i, j) be a predicate that means level i is less than level j.

Then rules 1-8 can be written in predicate calculus as follows:

**Rule 1:** (O) (C) (O ∈ C ⇔ Rid (O) < Rid (C)).

**Rule 2:** (BC) (SC) (Inherit-from (SC, BC) ⇔ Rid (BC) < Rid (SC)).

**Rule 3:** (CC) (CM) (Part-of (CC, CM) ⇔ Rid (CC) < Rid (CM)).

**Rule 4:** (TC) (AC) (Associated-with (TC, AC) ⇔ Rid (AC) < Rid (TC)).

**Rule 5:** (BCO) (BC) (SCO) (SC) ((BCO ∈ BC) ∧ (SCO ∈ SC) ∧ Inherit-from (SC, BC) ⇔ Rid (BCO) < Rid (SCO) < Rid (BC) < Rid (SC)).

**Rule 6:** (CCO) (CC) (CMO) (CM) ((CCO ∈ CC) ∧ (CMO ∈ CM) ∧ Part-of (CC, CM) ⇔ Rid (CCO) < Rid (CMO) < Rid (CC) < Rid (CM)).

**Rule 7:** (TCO) (TC) (ACO) (AC) ((TCO ∈ TC) ∧ (ACO ∈ AC) ∧ Associated-with (TC, AC) ⇔ Rid (ACO) < Rid (TCO) < Rid (AC) < Rid (TC)).

**Rule 8:** (i) (j) [LL(i, j) ∧ Level (Ci) ∧ Level (Cj) ⇒ {(Oi ∈ Ci) (Oj ∈ Cj) → Rid (Oi) < Rid (Oj) < Rid (Ci) < Rid (Cj)}].

### 6.2.3 System Model

Fig 6.1 shows the architecture of the proposed system. It is assumed that the issues relating to partitioning of objects [Huang1990] in OODS are resolved. The Message Handler receives the client request (transaction) that is usually a call to an object method and derives method type and properties [Riehle2000a, Riehle2000b]. The transaction manager identifies the type of lock. Further, the granularity of the lock is decided by the type and properties of methods as in previous section. Since the resources are objects in OODS, resource manager in each of the sites is called as
object manager. The transaction manager needs to get the resources from object manager to execute the transaction. The lock manager is responsible for maintaining the lock status of the resources in the site.

The resources granted to a transaction have to be locked before usage. The lock modes are shared (S) and exclusive (X). This is done to enforce synchronization and concurrency control through serialization. Once the execution is over, the lock is released and the resource can be utilized by the next waiting transaction. The object subsystem is responsible for the lifecycle of objects. It is also responsible for providing object persistence and garbage collection and other related activities. The deadlock preventor module checks for the presence of circular wait condition. If circular wait condition is not present, transactions can get the resources and execute.

![Architecture diagram of the proposed system](image)

**Figure 6.1** Architecture diagram of the proposed system

### 6.2.4 Deadlock prevention modules description

#### Preparing the system

The algorithm is based on prevention of circular wait condition. Prevention of circular wait condition is achieved using resource ordering and access ordering. Hence, DPA requires some preprocessing to be done before accepting the client transactions. The preprocessing steps are:

1. Global ordering of the partitioned resources in the system.
2. Deployment of resources to various sites.

#### Global ordering

As the proposed algorithm is based on resource ordering, it is necessary to order the resources globally to avoid access conflicts. Since objects are the resources and
their global relationship is known from the class diagram, the ordering is done on the class diagram. The class diagram is partitioned into levels based on the rules of horizontal partitioning. Then resource ordering is done by applying the total ordering rules specified in section 6.2.1.

**Resource ordering method**

1. Let $<$ be the initial total order of all the objects. This ordering starts from level 1 up to level n.
2. For every set of objects $O_1$ belonging to classes $C_1$ in level 1, add the objects arbitrarily.
3. For objects in level 2 to level n, order the objects in increasing dependency such that resource id of objects in level $i$ is less than resource id of objects in level $j$, where $i < j$. Define the function $\text{max\_object\_Rid}(O)$, to return the maximum of resource ID of all objects $O$ participating in the system which will be the resource id of last object in level n.
4. For all classes in level 1, i.e. independent classes (classes that need not lock any other classes to preserve consistency, like base classes and component classes), add the classes arbitrarily to create the total order such that $\text{max\_object\_Rid}(O)$ is always less than resource IDs of independent classes i.e. classes in level 1.
5. For classes in level 2 to n, order the classes in increasing dependency, such that resource ID of classes $C_i$ in level $i$ is less than resource IDs of classes $C_j$, where $i < j$.
6. Let be the smallest transitively – closed order that is compatible with steps 1-5.

![Sample class diagram after resource ordering](image)

**Figure 6.2** Sample class diagram after resource ordering
Deployment

Figure 6.2 shows the resource ordering of the sample class diagram. In this class diagram, there are four levels. The objects and classes are ordered from level 1 to level 4 using the resource ordering technique proposed earlier. The maximum resource id for objects is 24. Figure 6.3 shows the class diagram after deployment of classes, objects and associated database fragments to various servers. The transactions may come to any of these servers. The class diagram is horizontally partitioned and assigned to four sites S1 to S4.

![Class Diagram](image)

**Figure 6.3** Deployment of sample class diagram

6.2.5 **Deadlock prevention algorithm**

The deadlock prevention algorithm imposes the following constraints:

- All transactions are time stamped.
- A transaction can request only one resource at a time. It can request for the next resource only when previous resource is granted.
- A transaction cannot request for resource of higher ID without getting the resource of lower ID.

It involves the following modules.

1. **Determine the explicit resources needed**

When a transaction enters the system, it request for resource. The resource that is determined from the transaction directly is called explicit resource. The resource type and granularity should be determined first. If the transaction maps onto instance method, the resource will be an object and its ID is known from the system. If it is a class method, it means all objects of the class needs to be locked. Here instead of locking objects, class has to be locked. Then the resource ID will be the ID of the class whose method is called in the transaction.
2. **Determine the implicit resources needed**

This step is needed mainly to maintain consistency of the system. As already mentioned, both base class and sub classes should be locked simultaneously, when sub class is requested by a client. Since base classes and their sub classes are mapped on to the same database, simultaneous access may lead to inconsistency. Similarly, a composite object and its component objects are not allowed simultaneous access. This is due to the ‘part of ‘relationship shared between component objects and composite objects. This is also extended to association.

In this module, the resource ID of the explicit (dependent) resource is checked for its relationship with other classes. This information can be documented using C++doc or Javadoc. If it is a subclass, all its parent classes need to be locked. Similarly, in the case of composite objects, all its component objects need to be locked. Then parent classes and/or component classes are implicit (independent) resources for the request.

Finally, this module will enumerate all the resources needed for satisfying the transaction request.

3. **Request Ordering**

First, the list of resources for every transaction, obtained from the previous module is sorted in increasing order of resource IDs. The objective of this request ordering is to enforce the rule that resources of higher IDs can be requested only after obtaining resources of lower IDs. The location of the servers for these distributed requests need to be identified, as the resources are partitioned to several sites.

4. **Access Ordering**

Any good DPA should prevent deadlocks and starvation. Starvation is abortion of same transaction repeatedly. If the transactions are allowed to access the resources based on their arrival time by FIFO basis, starvation in poverty will happen. If transactions are allowed to access the resources based on priority, starvation in wealth will happen. Hence, the proposed scheme provides a combination of FIFO ordering and priority ordering as given below. Here the priority is based on whether the transaction requests for dependent or independent resources. As the resource ID requested by a transaction increases, its priority decreases. This priority policy is chosen to improve concurrency.

Since horizontal portioning is followed, all the transactions will be accessing the sites in same order. First, all the transactions will request the resource of smaller IDs.
Due to horizontal partitioning, this will localize the transactions to the same site. The following rules are applied to grant a resource.

**RULE 1:** Transactions $T_i$ and $T_j$ are granted resources independently of their arrival time, if they satisfy 1 and 2 or 1 and 3 of the following conditions:

1. The resources requested by $T_i$ and $T_j$ are different  
   \[ \text{i.e. } \text{Rid (ReqTi)} \cap \text{Rid (ReqTj)} = \emptyset. \]

2. Both $T_i$ and $T_j$ have their resource requests i.e. $\text{Rid (ReqTi)}$ and $\text{Rid (ReqTj)} \leq \text{max_object_Rid (O)}$.

3. Both $T_i$ and $T_j$ have requests i.e. $\text{Rid (ReqTi)}$ and $\text{Rid (ReqTj)}> \text{max_object_Rid (O)}$.

**RULE 2:** Transactions $T_i$ and $T_j$ are granted resources on FIFO basis when the resource IDs are conflicting i.e. $\text{Rid(ReqTi)} \cap \text{Rid (ReqTj)} \neq \emptyset$.

**RULE 3:** Transaction $T_i$ has higher priority over $T_j$ when

1. The resources requested by $T_i$ and $T_j$ are different  
   \[ \text{i.e. } \text{Rid(ReqTi)} \cap \text{Rid (ReqTj)} = \emptyset \]

2. $T_i$ has its $\text{Rid(ReqTi)} \leq \text{max_object_Rid (objects)}$ and $T_j$ has its $\text{Rid (ReqTj)}> \text{max_object_Rid (objects)}$.

### 6.2.6 Informal Proof

The following scenarios are considered to show the working of proposed DPA by taking typical sample scenarios. The transactions can request the resources only in two granularities namely classes and objects. So it is enough if the access ordering is done for transactions requesting objects, classes and objects and classes.

**Scenario 1:** This is a scenario where rule 1 is applied. *(Maximum Concurrency)*

Rule 1 ensures maximum concurrency. The transactions are allowed to execute in parallel as long as their resource requests are mutually exclusive.

For example with reference to the above sample class diagram Let $T_i$ requests \{d1\} and $T_j$ requests \{f1\}.

Resource requests for $T_i$ = \{a1, d1\}.

(Where a1 is the associated base class object of d1)

Requested resource IDs for $T_i$ = \{1, 7\}.

Similarly resource requests for $T_j$ = \{b1, f1\}.

(Where b1 is the associated base class object of f1)

Then resource request IDs for $T_j$ = \{3, 11\}
They satisfy Rule 1, since \( \{1, 7\} \cap \{3, 11\} = \emptyset \)
And both resource request sets \( \{1, 7\} \) and \( \{3, 11\} \) are < 24 (max_object_Rid (objects))
Then Ti and Tj can execute in parallel.

**Scenario 2: This is a scenario where rule 2 is applied. (Starvation in Poverty)**

This eliminates starvation in poverty. When the resource requests are conflicting, they are served on FIFO basis, as both transactions want some or all the resources that the other transaction also needs. Then younger transaction will have to wound and wait. Wound and wait is better than wait and die, because of rollback and restart costs.

For example

Ti requests \{h1\}; Tj requests \{d1\}.
Then resource request for Ti = \{a1, d1, h1\}. Resource IDs for Ti are \{1, 7, 15\}.
Tj request list is \{a1, d1\}. And request resource IDs are \{1, 7\}

\[ \text{Rid (ReqTi)} \cap \text{Rid (ReqTj)} = \{1, 7, 15\} \cap \{1, 7\} \neq \emptyset \]
Then Ti and Tj are serialized based on their arrival time.

**Scenario 3: This is a scenario where rule 3 is applied. (Starvation in Wealth)**

This eliminates starvation in wealth. When two transactions implicitly conflict, then priority is given to transaction that needs lesser number of resources as an expedient scheduling measure. This will improve concurrency.

For example

Ti requests \{d1\}; Tj requests \{D\} (class locking implies it needs all its objects)
Resource requests of Ti = \{a1, d1\}; And requested resource IDs for Ti = \{1, 7\}
(\text{where } a1 \text{ is the associated base class object of } d1). 
Resource requests of Tj = \{a1, a2, d1, d2\}
(\text{where } a1, a2 \text{ are the associated base class objects of } d1 \text{ and } d2.)
Then resource request ID for Tj = \{32\}
They satisfy Rule 1, since \( \{1, 7\} \cap \{32\} = \emptyset \)

But implicitly Tj needs resources \{1, 2, 7, 8\}. This can be inferred by the second condition, Ti’s resource request sets \{1, 7\} <24 and Tj’s request ID \{32\} > 24 (max_object_Rid (O)). Then Ti gets higher priority over Tj.

**6.2.7 Formal Proof**

**Theorem 1:** The resource ordering < done using the proposed resource ordering method is total and meets rules 1-8.
orders each pair of resources \( r_i \) and \( r_j \): If the resources \( r_i \) and \( r_j \) are not already ordered, by step 2 and 3 all objects are ordered. Then at least one of them should be a class.

**Case 1:** \( r_i \) is an object and \( r_j \) is its parent class (Rule 1)

Steps 2 and 3 orders all objects by their dependency. Without loss of generality, by step 3 and 4, max_object_Rid (\( r_i \)) < Rid (\( r_j \)). By transitivity of <, it is proved that \( r_i < r_j \).

**Case 2:** \( r_i \) is a base class and \( r_j \) is its sub class (Rule 2)

If \( r_i \) is in level \( i \) and \( r_j \) is in level \( j \), and levels \( i < j \), then in general by steps 4 and 5, Rid (\( r_i \)) <Rid (\( r_j \)). By definition of inheritance, base classes will be always in lower level than their subclasses. Hence, it is proved.

**Case 3:** \( r_i \) is a component class and \( r_j \) is its composite class (Rule 3)

If \( r_i \) is in level \( i \) and \( r_j \) is in level \( j \), and levels \( i < j \), then in general by steps 4 and 5,Rid (\( r_i \)) < Rid (\( r_j \)). By definition of composition, component classes will be always in lower level than their composite classes. Hence, it is proved.

**Case 4:** \( r_i \) is an associative class and \( r_j \) is associated class (Rule 4)

If \( r_i \) is in level \( i \) and \( r_j \) is in level \( j \), and levels \( i < j \), then in general by steps 4 and 5, Rid (\( r_i \)) < Rid (\( r_j \)). By definition of association, associated classes will be always in lower level than their associated classes. Hence, it is proved.

**Case 5:** Proof of constraint Rule 5

From cases 1 & 2 by transitivity, rule 5 holds good for all resources \( r_i \) and \( r_j \).

**Case 6:** Proof of constraint Rule 6

From cases 1 & 3, it is inferred that rule 6 holds good for all resources \( r_i \) and \( r_j \).

**Case 7:** Proof of constraint Rule 7

From cases 1 & 4, it is inferred that rule 7 holds good for all resources \( r_i \) and \( r_j \).

**Case 8:** Proof of constraint Rule 8

\( r_i \) is in level \( i \), \( r_j \) is in level \( j \) and \( i < j \). By steps 1-6, from cases 5, 6 and 7, it is proved that they are totally ordered.

**Theorem 2:** Ordering < is well defined, if \( r_i \) is less than \( r_j \), then \( r_j < r_i \) does not hold.

Let \( \zeta_\prec \) be defined as \{\( R, E \)\}, where resources \( R \) represent nodes and \( E \) is defined as set of edges linking those resources that are related by the ordering \( r_i < r_j \), where \( r_i, r_j \in R \) and < defines \( E \), then it is enough to show that \( \zeta_\prec \) contains no cycles.
The resources can be either objects or classes. All objects \( r_i \) and \( r_j \) are ordered by step 1 and 2 and edges \( r_i \rightarrow r_j \) are added for all objects at these steps. If \( r_i \) and \( r_j \) are classes related by inheritance and/or composition and/or association, then by steps 4 and 5 they are ordered and \( r_i \rightarrow r_j \) are added in this step. So it is clear that so far there are no back edges \( r_j \rightarrow r_i \) added, as the objects and classes are ordered individually. In step 3 objects and classes are ordered by defining \( \max_{object} \text{Rid}(object) \) to be less than the resource IDs for all classes. So, it is clear by transitivity, that all objects and classes are ordered. The edge \( r_i \rightarrow r_j \) is added where \( r_i \) is an object with maximum resource ID and \( r_j \) is a class. Since edges are added for each type of resource exclusively, there will be no cycles.

### 6.2.8 Summary

Any DPA is expected to handle 3 conditions namely (1) Deadlock (2) Starvation on Poverty (3) Starvation on Wealth.

1. is solved in our algorithm by access ordering. Transactions can access the resources only in specific order, i.e. resources from lower IDs to higher IDs. Since FIFO ordering is followed, younger transactions wait on older transactions. Hence, circular wait is broken. This ensures breaking of cycles and hence deadlock is prevented.

2. Transactions are satisfied on FIFO basis. This eliminates starvation in poverty. Starvation in poverty generally occurs when larger requests are kept pending permanently. This is because smaller transactions are favored over bigger transactions to increase throughput. However, in our algorithm, the transactions are served in FIFO basis. Hence, starvation in poverty is eliminated.

3. Though, our algorithm favors FIFO ordering, when two transactions arrive at the same time, it favors the transaction requesting least number of resources. Hence, an expedient strategy is followed to speed up the computation, improve the resource utilization and alleviate starvation of wealth.

Thus our algorithm has shown that deadlock prevention algorithm is possible for distributed object oriented systems.

### 6.3 Fault Informant Probe based Distributed Deadlock Detection Algorithm

Probe based DDDR algorithm is widely used in distributed systems because of its simplicity. It detects the deadlock by sending a probe through the Wait-for-Edges in GWFG. If the probe returns back to the initiator, deadlock is deduced. This algorithm
expects the network to be fault free. Practically networks are fault prone, hardware, network and software failures are bound to occur.

Though there are several fault tolerance algorithms for distributed systems, generally it is not expected by DDDR to provide fault tolerance. Hence much research work is not done in this area. However the DDDR algorithm should facilitate for the initiator of deadlock detection to infer whether it is due to live lock or site failure (where deadlock may be present) that the probe does not come back to it. If it is due to live lock, then the transactions can wait for finite time and then start sending resources’ requests again. If the probe does not come back due to site failure, then the system needs to be reconfigured to continue. But if it is actually due to deadlock, and probe does not return to initiator due to site failure, then it is a serious problem.

Since there is no ideal fault tolerance algorithm and site failures are bound to happen, the initiator needs to get the probe back always despite whether it is live lock or deadlock. Hence this chapter aims to propose a fault informant algorithm that sends colored probes to initiator indicating the sites’ status. It detects at most two site failures per deadlock cycle. The proposed algorithm uses the following colors in probe messages to indicate the status.

RED: Indicates deadlock and there is no site failure.
ORANGE: Indicates site failure. In this deadlock/live lock status is unknown due to site failure.
WHITE: Indicates live lock and there is no site failure.

In the proposed algorithm, transaction uses forward and backward probe messages to detect the reason for not getting the resource. Initially the color of the probe message is RED. The messages travel along the wait for edges and on the opposite sides by traversing node by node. A node that is receiving the probe message should send an acknowledgement to its sender. This is used to inform the active status of receiver to the sender.

If sender does not receive acknowledgement message before timeout, it infers that the receiver site has failed. Then sender will change the color of forward/backward probe messages into ORANGE after updating the bit in fault vector corresponding to the faulty site. The sender will send a return probe which is addressed always to the initiator with the color of forward/backward probe along with updated fault vector and fault site ID. The initiator will broadcast the faulty state
of the site to all the sites. This faulty status is modified only when the faulty site broadcasts awake message.

If the faulty site ID in both forward and backward probe messages is same, it can be inferred that it is one site failure. If they are different, then it is “two site failure” situation.

If there is no site failure, both the forward and backward probe messages will reach the initiator and by the RED color of the probe, the initiator will infer the presence of deadlock and will start deadlock resolution phase.

If there is no failure, but the wait for graph terminates at some node which does not have wait for edge, then receiver will not be able to send message any further. It will send acknowledgement message to sender indicating that it is active. Then receiver will change the color of the forward/backward probe into WHITE, and return probe with forward/backward probe color is sent to initiator. When the initiator realizes it is live lock, it will wait for some more time.

In the next section, the data structures and message formats used in the algorithm are defined. In section 6.3.2, the system model is described. In section 6.3.3, the proposed fault informant probe based DDDR algorithm is explained with pseudo code. In section 6.3.4, formal proof for the algorithm is given and section 6.3.5 summarizes the paper.

6.3.1 Definitions

Definition 1: Wait for Graph (WFG (N, E)) is a directed graph where nodes N represent transactions currently participating in the system and E is a finite set of edges representing the transaction dependency on resources. Ti→Tj e E where Ti is waiting on Tj for the resource held by Tj. So Tj is successor of Ti and Ti is predecessor of Tj.

Definition 2: A Deadlock is identified by a directed cycle in the WFG.

Definition 3: Forward probe (Forward_Probe (Initiator, Sender, Receiver, Forward_Probe_Color)) is a traversal of dependency edges in WFG from initiator and propagates until it reaches back initiator or terminates when there is no dependency edge for a transaction in the path i.e. TI → T1→T2…Tn, where {Tn = TI or Tn has no dependency edge | TI ,T1,T2….Tn e N}. The probe color is RED if there is a deadlock and WHITE if there is live lock in fault free environment. This probe will not reach the initiator in faulty environment.
Definition 4: Backward probe (Backward_Probe (Initiator, Sender, Receiver, Backward_Probe_Color)) is a traversal from initiator and propagates backwards along the dependency edges in WFG, i.e. T1 ← T1←T2…Tn, where \( \{T_n = T_1 \text{ or } T_n \text{ has no dependency edge } | T_1 \rightarrow T_1 \rightarrow T_2…T_n \text{ are directed edges } \in E \text{ and } T_1,T_2…,T_n \in N \} \). The objective of using both probes is to identify at most 2 site failures in a deadlock cycle than 1 site failure as in[1]. The probe color is RED if there is a deadlock and WHITE if there is live lock in fault free environment. This probe will not reach the initiator in faulty environment.

Definition 5: A Fault Vector (FaultVector) \( V = S_1S_2...S_n \), where \( S_1, S_2, ...S_n \) denotes the N sites participating in the system domain. \( S_i = 1 \), if site i is faulty; \( S_i = 0 \), if site i is non faulty. Instead of PMC diagnosis model, the site fault is identified by message response from the neighboring sites.

Definition 6: Return probe (Return_Probe (Initiator, Sender, Forward/Backward_Probe_Color, FaultVector, FaultSiteID,)) is the probe forwarded by the site Si holding transaction Ti to the initiator about its successor faulty site Sj holding transaction Tj, where Ti→Tj \( \in E \). This probe updates the status of site Sj in fault vector and sends it to the initiator. The initiator updates the status of Sj and broadcasts to all the other nodes for future requests. It stays unchanged until the awake message is received from the faulty site Sj. This is done during forward probe. In backward probe, if predecessor Tj is faulty, then this return probe is forwarded by the successor. The return probe color is WHITE if there is live lock in fault free environment. The return probe color is ORANGE if there is site failure. The return probe will have FaultVector and FaultSiteID only under faulty environment.

Definition 7: Acknowledgement message (Ack_msg (Receiver, Sender)):- Every site on receiving the probe message from its sender should send an acknowledgement message to its sender. If this message is not received by the sender, by time out period, it assumes that receiver is faulty. Then sender sends return probe to initiator updating fault vector about this faulty site.

Definition 8: Clean message (Clean_message) is to broadcast all the sites to clean the probes sent by victim which is in faulty site.

Definition 9: Victim is the lowest priority transaction which will be aborted to break the cycle. Here initiator is the victim.
Definition 10: Awake message (awake_mesg (SiteID)) is a message sent by all sites on startup or after fault recovery. This message is needed to update its status in fault vector and include it for further transaction requests.

6.3.2 System Model

The system is assumed to be free of congestion for timely delivery and messages are received in the order in which they are delivered. Further priority based DDDR algorithm [Mitchell1984] exists to ensure the least priority transaction in the cycle to become initiator of probe messages. This helps avoiding phantom deadlocks due to simultaneous initiation of probe messages for the same cycle. In each site it is assumed that only one transaction is running at a time. A transaction failure is also assumed as site failure. Each site is running one transaction for simplicity sake. The site index and transaction index are assumed as same.

6.3.3 Fault-Informant Probe Based DDDR Algorithm

In this proposed scheme, initiator will send forward as well as backward probes. Hence the algorithm in Li1993 uses backward probe alone and can detect only one site failure per deadlock cycle. Intuitively if we use both forward and backward probes in our algorithm, at most two failures can be detected. To improve the reliability of the system, we use both probes in our mechanism. The procedures for deadlock detection and resolution are given below.

```
Procedure Site_Initialization
If (fault_recovery or start_up)
    Broadcast awake_mesg(SiteID)
End procedure.
```

This procedure will be called whenever a site is started or recovered from failure. On receiving this message, all the other non faulty sites will update the status of this site in their fault vector.

```
Procedure Transaction_ Initialization
probe = null;
Fault_ Vector = Get_faultvector();
End Procedure
```

Any transaction that comes to the system will initially have the probe queue empty. It will get the current status of sites from the neighboring sites. Any transaction after making request for a resource will wait till time out or grant message which ever comes early. After time out, it will start sending probe messages. Any
transaction Ti that receives forward or backward probe messages will execute the following procedure.

Transaction Ti::
Do
If Receive Forward_Probe (Initiator I, Sender Ti-1, Receiver = Ti, Forward_Probe_Color = RED)
{
    Send Ack_msg to Sender Ti-1
    If there is no dependency edge from Ti
    {
        Send Return-Probe (Initiator, Sender, Forward_Probe_Color = WHITE);
        Exit;
    }
    else
    {
        Send Forward_Probe (Initiator I, Sender=Ti, Receiver = Ti+1, Forward_Probe_Color = RED)
        Until timeout
        {
            Wait for Ack_msg from Ti+1
            If Receive Ack_msg from Ti+1 break;
        }
        Update FaultVector[Si+1] = 1;
        Update FaultSiteID = S i+1;
        Send Return-Probe (Initiator I, Sender Ti, Forward_Probe_Color = ORANGE, FaultVector, FaultSiteID);
    }
}

If Receive Backward_Probe (Initiator I, Sender Ti+1, Receiver = Ti, Backward_Probe_Color = RED)
{
    Send Ack_msg to Sender Ti+1
    If there is no dependency edge from Ti
    {
        Send Return-Probe (Initiator I, Sender Ti, Backward_Probe_Color = WHITE);
        Exit;
    }
    else
    {
        Send Backward_Probe Initiator I, Sender=Ti, Receiver = Ti-1, Backward_Probe_Color=RED)
        Until timeout
        {
            Wait for Ack_msg from Ti-1
            If Receive Ack_msg from Ti-1 break;
        }
        Update Faultvector[Si-1] = 1;
Update FaultSiteID = Si-1;
Send Return- Probe (Initiator =I,
Sender=Ti,Backward_Probe_Color=ORANGE,
FaultVector, FaultSiteID);

The existing DDDR algorithm [Chowdary1989, Roseler1988, Sinha1985] determines the lowest priority transaction in a deadlock cycle and nominates it as initiator. Initiator will send forward probe along dependency edges and backward probe along the opposite direction of dependency edges.

Initiator:
Switch on case
{
Case 1: //Livelock/Deadlock–INITIATOR’S NEIGHBORING SITE(S) FAILURE
{
Send Forward_Probe (Initiator I,Sender=I,
Receiver=Ti, Forward_Probe_Color=RED)
{
Until timeout
{
Wait for Ack_msg from Receiver Ti;
If Receive Ack_msg from Receiver Ti break;
}
Update Faultvector[Si] = 1;
Update FaultSiteID = Si;
Forward_Probe_Color = ORANGE;
// Declare Live lock or Deadlock due to site failure Si
Broadcast Clean_message to roll back transaction Ti in the faulty site Si;
}
Send Backward_Probe (Initiator I, Sender I,
Receiver = Tj, Backward_Probe_Color = RED)
{
Until timeout
{
Wait for Ack_msg from Tj
If Receive Ack_msg from Tj break;
}
Update Faultvector[Sj] = 1;
Update FaultsiteID = Sj;
Backward_Probe_Color = ORANGE;
// Declare Live lock or Deadlock due to site failure Sj;
Broadcast Clean message to roll back transaction Tj in the faulty site Sj;
}
Case 2: // Deadlock—NO SITE FAILURE; Probe comes back to the initiator

If Receive Forward_Probe (Initiator I, Sender Ti, receiver = Initiator, Forward_Probe_Color = RED) AND Receive Backward_Probe (Initiator I, Sender Tj, receiver = Initiator, Backward_Probe_Color = RED)
{
    Call Deadlock Resolution Algorithm;
    // Declare Deadlock;
}

Case 3:    // Livelock – NO SITE FAILURE
If Receive Return_Probe (Initiator I, Sender Ti, Receiver = Initiator, Forward_Probe_Color = WHITE) AND Receive Return_Probe (Initiator I, Sender Tj, Receiver = Initiator, Backward_Probe_Color = WHITE)
{
    Wait until timeout;       // Declare Livelock;
}

Case 4:   // Deadlock/ Livelock in 1 / 2 SITE FAILURES

If Receive Return_Probe (Initiator I, Sender Ti, Receiver = Initiator, Forward_Probe_Color = ORANGE, FaultVector, FaultSiteID) AND Receive Return_Probe (Initiator, Sender Tj, Receiver = Initiator, Backward_Probe_Color = ORANGE, FaultVector, FaultSiteID)

If FaultSiteID in Forward probe== FaultSiteID in Backward probe
{
    // Declare Live lock or Deadlock due to 1 site failure
    Broadcast Clean message to roll back faulty transaction Ta in the faulty site Sa;
}

If FaultSiteID in Forward probe <> FaultSiteID in Backward probe
{
    // Declare Live lock or Deadlock due to 2 site failures.
    Broadcast Clean message to roll back transactions Ta and Tb in the faulty sites Sa and Sb;
}
}

EndCase.
6.3.4 Formal Proof

The algorithm is proved correct under the following assumptions:

1. Transactions use single request model for requesting the resource.
2. No transaction in deadlock aborts unless it is victimized in resolution phase.
3. There are at most 2 site failures in a cycle.

**Theorem 1**: The algorithm detects deadlock only if there is a deadlock.

**Proof**: This algorithm detects a deadlock only when it receives both forward and backward probes and their colors are red. In that case, no site failure was there, when the probe was traversing. If a site had not sent acknowledgement message to its sender, the initiator would have received return probe messages. See figure 6.4.a for the traversal of probes in deadlock situation. See fig 2b for the probe traversal in faulty environment.

**Scenarios**

These scenarios are considered to give informal proof to the algorithm. They show how the algorithm works under both fault free and faulty environment. They also show how deadlock and live lock status are detected. In fault diagnosis model, it is shown that 2t+1 processors are needed to detect t failures. In our proposed algorithm, it is shown that 2t-1 processors are enough to detect t failures through messaging.

![Figure 6.4a](image)

**Figure 6.4a** Deadlock in fault free environment

Since the proposed algorithm also can detect at most 2 site failures, minimum number of nodes are taken to show the working of the proposed algorithm.
Figure 6.4 shows various possible scenarios in a distributed system during deadlock detection phase.

**Scenario 1:** Figure 6.4a depicts the scenario when deadlock occurs in fault free environment. Initiator T1 sends red colored forward and backward probes along dependency edge and opposite direction of dependency edge. When the initiator receives back both probes, it infers the presence of deadlock and that there is no site failure along the path. Then initiator (lowest priority) is victim. The color of the probe is RED.

![Figure 6.4a](image)

**Figure 6.4b:** Live lock in fault free environment - case 1

**Scenario 2:** In figure 6.4b, let us assume that T1-the initiator does not have predecessor. Then backward probe will not be sent back to the initiator. As T4 does not have any dependency edge, forward probe terminates at T4. T4 changes the forward probe to WHITE indicating live lock status and T4 is active. This is also a scenario in fault free environment.

![Figure 6.4b](image)

**Figure 6.4c** Live lock in fault free environment – case 2

**Scenario 3:** Figure 6.4c also depicts live lock status in fault free environment. Here T2 is assumed to be having least priority. Hence it becomes the initiator.T3 and T1
send forward and backward probes indicating live lock. So they change the color of the forward and backward probes WHITE and send return probe back to initiator.

**Scenario 4:** In figure 6.5a, let us assume T4 is faulty. T3 waits until time out for acknowledgement message from T4. If there is no acknowledgement message from T4, then it updates the fault vector for site 4, changes the color of probe message to ORANGE indicating site failure and sends return probe to initiator. T1 also sends backward probe. T4 sends no acknowledgement even after timeout. T1 concludes T4 faulty. It is confirmed by ORANGE forward probe from T3. Since the fault site id in return probe from T3 matches with the faulty site deduced by initiator, it concludes one site failure and it may be a live lock or deadlock situation.

**Scenario 5:** In figure 6.5b, T2 is initiator. It sends forward probe to T3 and backward probe to T1. T4 is faulty. So T1 and T3 will not receive acknowledgement messages. They will change the probe color to ORANGE and send back to initiator after updating fault vector on T4. Since both T1 and T3 will have their fault site ID same, it is concluded that it is live lock/ deadlock due to single site failure.

---

**Figure 6.5a** Live lock/Deadlock in faulty environment (1 site failure)- case1

**Figure 6.5b**: Live lock / Deadlock in faulty environment (1 site failure)- case 2
**Scenario 6:** In figure 6.5c, Assume T3 and T4 are both faulty. T2 will send forward probe to T3. T3 will not send acknowledgement message. So T3 is updated as faulty on time out. T2 sends T1 backward probe. T1 forwards backward probe to T4. As T4 is also faulty, it will not send acknowledgement to T1. So T1 changes return probe to ORANGE and updates T4 status. Since the fault side IDs updated by initiator and T1 are different, T2 will understand both T3 and T4 are both faulty and infer that it is two site failures scenario.

![Figure 6.5c](image)

**Figure 6.5c** Live lock /Deadlock in faulty environment (2 site failure)- case1

**Scenario 7:** This is the worst case for 2 site failures with 4 sites in picture. Let T1 and T3 are faulty. Initiator sends forward and backward probes to them. On time out, since it does not receive acknowledgement messages from neither T1 nor T3, it deduces that T1 and T3 are both faulty. However the status of T4 is unknown, as it is unreachable by both T1 as well as T3. Since this algorithm can only detect at most 2 site failures, it cannot be inferred. However in Preparata1967, it is stated that at least 2t+1 processors are needed to detect t failures using PMC diagnosis model. However in our case it can be inferred that t failures can be detected with 2t – 1 processors.

![Figure 6.5d](image)

**Figure 6.5d** Live lock/Deadlock in faulty environment (2 site failure) – case 2
6.3.5 Summary

A new fault tolerant algorithm for Distributed Deadlock Detection and Resolution is proposed with the following improvements:

- Initiator always knows the status of probe whether deadlock or live lock or site failure.
- In Li1993 every non-faulty site tests other sites periodically for site failures. In the proposed algorithm the site failure is decided by acknowledgement messages. This improves the throughput of non-faulty sites.
- Checking whether faulty sites are rectified is known by awake message. This situation is not handled separately in Li1993.
- Only one site failure is handled in Li1993. This paper however handles at most 2 site failure which improves fault tolerance.
- The color of the probe is used to indicate the status of the system. Red indicates deadlock with no site failure. Orange indicates live lock or deadlock due to site failure. White indicates live lock due to a transaction having no dependency edge.
- The worst case message complexity is $4n$ where $n$ is the number of transactions. This occurs when there is no site failure and deadlock occurs. The four messages are the forward probe message and backward probe message to next nodes and acknowledgement messages for both forward and backward messages to senders (see figure 6.4a).
- Further fault identification is better than fault diagnosis model, which needs $2t+1$ processors to identify $t$ failures. In the messaging mechanism, $2t-1$ processors are enough to identify $t$ failures.

6.4. Weight Based Victim Selection Algorithm

Deadlock resolution phase follows deadlock detection phase. In this a transaction is chosen (called as victim) for abortion to break the circular wait that is causing the deadlock. Several victim selection algorithms have been analyzed in chapter 2.4.3.

6.4.1 Performance of existing victim selection algorithm

Based on the definition of the victim selection algorithms, their characteristics along with their time complexity can be summarized as in table 6.1. The time complexity helps to determine the deadlock resolution latency and defined in terms of ‘n’- the number of transactions. From the table, it is worth noting certain points.
Table 6.1: Comparison of various victim selection policies

<table>
<thead>
<tr>
<th>Victim Selection Policy</th>
<th>Optimal in</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Youngest [Agarwal1987]</td>
<td>Fairness</td>
<td>O(n)</td>
</tr>
<tr>
<td>Min. History [Agarwal1987]</td>
<td>Fairness</td>
<td>O(n)</td>
</tr>
<tr>
<td>Least Static Priority [Newton1979]</td>
<td>Response time</td>
<td>O(n)</td>
</tr>
<tr>
<td>Maximum Size[Chow1991]</td>
<td>Throughput</td>
<td>O(n)</td>
</tr>
<tr>
<td>Min. no. of locks[Agarwal1987]</td>
<td>Resource Utilization</td>
<td>O(n)</td>
</tr>
<tr>
<td>Max. no of cycles[Singhal1989]</td>
<td>Throughput</td>
<td>O(n^2 x k + 2) k– max. cycle size</td>
</tr>
<tr>
<td>Minimum cost[Srivastava2007] Max. abortion</td>
<td>Resource Utilization</td>
<td>O (n3)</td>
</tr>
<tr>
<td>Max. Edges [Singhal1989]</td>
<td>Resource Utilization</td>
<td>O (n4)</td>
</tr>
<tr>
<td>Blocker [Agarwal1987]</td>
<td>Resolution latency</td>
<td>O(1)</td>
</tr>
<tr>
<td>Initiator</td>
<td>Resolution latency</td>
<td>O(1)</td>
</tr>
<tr>
<td>Max. release set[Moon1997]</td>
<td>Resource utilization, throughput</td>
<td>O (n3m) m- no of resources</td>
</tr>
<tr>
<td>Min. work done so far[Holt1972]</td>
<td>Resource Utilization</td>
<td>O (n3)</td>
</tr>
<tr>
<td>Priority + resource priority + min. work done[Garey1979]</td>
<td>Resource utilization, throughput</td>
<td>O(n4.m ) m- no of resources</td>
</tr>
</tbody>
</table>

Youngest is fair in giving priority to the transactions based on their arrival time. So this will eliminate starvation in poverty [Holt1972]. But this might introduce starvation in wealth [Parnas1972]. Static priority lets the user to configure the priorities of the transactions participating in the system. This eliminates starvation in wealth. This policy is ideal for real time systems. The transactions can be prioritized based on the need of immediate or delayed response time. But resolution using history eliminates both starvation in wealth and starvation in poverty. History based resolution is also fair in the sense that it does not penalize any transaction again and again. Resolution based on transaction size will increase the number of transactions completed per unit time i.e. throughput.

Resolution policies like minimum number of locks, minimum work done and minimum abortion cost focus on lower rollback cost of a transaction, while algorithms like initiator, blocker, maximum edges and maximum release set choose a victim based on overall better performance of the system than on concerned individual transactions. The victims chosen using these algorithms might have already acquired all the required resources, completed maximum amount of execution or had been aborted again and again in the past. So they are not fair on individual transactions.

Blocker and initiator can be lower priority transactions and their only benefit is better deadlock resolution latency, especially in distributed systems. While all the
above mentioned algorithms abort one transaction per cycle, resolution based on maximum number of cycles tries to reduce this. So number of transactions executed per unit time i.e. throughput increases in this case.

A simulation experiment has been made to study their characteristics with respect to other attributes. To study these desirable characteristics, attributes of 500 transactions are randomly generated and tested. From the given victim selection algorithms, blocker and initiator algorithms are not considered because, they are optimal only in deadlock resolution latency and poor in other aspects. Victim selection algorithm by Srivastava2007 takes maximum deadlock resolution latency, hence it is also not considered.

![Figure 6.6 Performance for number of transactions versus throughput](image)

Figure 6.6 Performance for number of transactions versus throughput

In figure 6.6, it can be noticed that algorithm choosing victim based on maximum number of cycles provide maximum throughput i.e. more than 96%. This is because it aborts at most ‘n’ transactions, when there are ‘n’ cycles, whereas other algorithms abort atleast ‘n’ transactions. Then selection on maximum size provides better throughput i.e. 94%. This is because smaller transactions finish in time when the transactions are relatively smaller in size. In max edge algorithm, by aborting one transaction many transactions can proceed. Therefore the throughput is more in this case also. The performance of other resolution algorithms also depend on attributes of participating transactions and are bound to vary.
In figure 6.7, resource utilization is compared by varying number of transactions. While throughput is measured in terms of number of transactions, resource utilization is measured in terms of resources. Maximum resource utilization happens when there is minimal rollback. Resource utilization is maximized in resolution algorithms of minimum number of locks and maximum size. It is also noticeable that minimum abortion cost based on arrival time and minimum number of operations has made the algorithm suboptimal in both aspects. But it is better than algorithms considering single transactional attribute.

Figure 6.7 Performance for number of transactions versus resource utilization
In figure 6.8, fairness is compared with number of transactions. Fairness can be viewed in two aspects: based on age and starvation. The fairness considered in figure 6.8 is based on arrival time. While throughput, deadlock resolution latency and resource utilization are desirable attributes of the system, non-starvation and fairness are desirable attributes of individual transactions.

### 6.4.2 Proposed weight based victim selection algorithm

The victim selection algorithms are based on transaction attributes and resource attributes. The common transaction attributes or characteristics are age, history, code size, priority, resource utilized so far and its attributes in WFG like in-degree, number of edges, out-degree and cycle participation. Apart from static priority, transactions are usually dynamically prioritized based on the attributes given above. Victim selection algorithm based on resource attributes could be based on resource priority, number of units of a particular resource, costly resource, resource most sought after etc.
The desirable attributes that could improve the system are higher throughput, better resource utilization and lesser deadlock latency. The desirable attributes in individual transaction execution is lower response time, fairness, no starvation and minimum roll back cost. It can be noticed that while each victim selection algorithm is optimal in one aspect, it is suboptimal in other aspects.

Hence the proposed algorithm is based on assigning weights which can be configured based on user requirement. For example in real time systems, response time is more important than other attributes. Similarly throughput is important in batch processing systems.

In the proposed algorithm, each transaction is expected to possess an attribute list to maintain its rank in various aspects. The attribute list of a transaction is as table 2a. The attribute list is created for every transaction arriving at the system. Attribute lists of all transactions whose execution are completed are deleted. Attribute lists of all live transactions whose execution are not completed, are also updated whenever a new transaction arrives or an old transaction leaves the system.

The rank of a transaction with respect to a particular attribute is based on its value relative to other transactions with respect to that factor. For example, if a transaction has arrived third among the active transactions in the system, then its rank with respect to age is fixed as 3. In general, the ranks are determined based on the seniority of the transaction.

Table 6.2a  Transaction attribute list

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td></td>
</tr>
<tr>
<td>History</td>
<td></td>
</tr>
<tr>
<td>No of resources requested</td>
<td></td>
</tr>
<tr>
<td>No of resources granted</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td></td>
</tr>
<tr>
<td>Static priority</td>
<td></td>
</tr>
</tbody>
</table>

The weights of desirable attributes in the system can be configured so that \( \Sigma (G, F, L, T, R) = 100\% \). This is done based on the nature of the distributed system. The weights can be in the range 0 to 100\%. The ranking of transactions in a centralized system is easy. However the deadlock detection and selection of a victim for resolution in distributed system is very tedious. The candidate victims are distributed in various sites. Selecting a victim transaction for abortion at each local site is
suboptimal and affects the performance of the system. Hence global selection of
victim needs propagation of transaction attributes to all sites. The sites in a distributed
system communicate through messages. Hence probe based deadlock detection in
Chandy1983 is one of the best distributed deadlock detection algorithms.

<table>
<thead>
<tr>
<th>Desirable system attribute</th>
<th>Wt as %</th>
<th>Rank of Transaction attribute to be favored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>G</td>
<td>Size</td>
</tr>
<tr>
<td>Fairness</td>
<td>F</td>
<td>Age, history</td>
</tr>
<tr>
<td>Resolution latency</td>
<td>L</td>
<td>Initiator, random blocker</td>
</tr>
<tr>
<td>Response time</td>
<td>T</td>
<td>Static priority</td>
</tr>
<tr>
<td>Resource utilization</td>
<td>R</td>
<td>No of resources requested</td>
</tr>
</tbody>
</table>

In this algorithm, the transactions waiting for grant message will start sending
probe message after time out. The probe is sent along the wait for edges of a Global
Wait for Graph (GWFG). The probe message has the fields such as initiator
(transaction initiating the probe), sender (transaction forwarding the probe), receiver
(transaction receiving the probe). This algorithm is used to detect cycle which
indicates the presence of deadlock. Two new fields’ namely current victim’s
transaction ID and its attribute list can be added to the probe for propagation for
global victim selection. At each site, the rank of the current victim is compared with
the locally selected victim. If the rank of local victim is higher than the current victim,
the current victim can be replaced before forwarding the probe. When the probe
reaches the initiator, the Transaction ID which is to be aborted will be known. Then
command can be sent to abort the victim to break the cycle and restart the system.

6.4.3 Summary

Resolution is an important phase in handling deadlocks. Selection of a victim
influences the performance of the system strongly. The desirable performance
parameters of the system like throughput, resource utilization are influenced by the
victim selection. The desirable parameters of the transaction like fairness, resolution
latency and response time are to be considered while selecting a victim. The weight
based victim selection scheme balances the desirable parameters of transactions and
system.