Optimal Fault-Tolerant Resource Allocation in Dynamic Distributed Systems

(Extended Abstract)

Injong Rhee
Department of Computer Science
University of Warwick
Coventry, CV4 7AL
England
rhee@dcs.warwick.ac.uk

Abstract

This paper presents a fault-tolerant resource allocation algorithm in a dynamic distributed message passing system, where concurrent processes sharing system resources can be created or terminated dynamically. The degree of fault-tolerance is measured by the failure locality that is the maximum number of processes whose liveness conditions (e.g., starvation freedom) cannot be satisfied because of a single process failure. The algorithm guarantees the optimal failure locality.

1 Introduction

Distributed systems commonly include resources (I/O channels, buffers, space, data files, etc.) which must be managed so that no two processes access the resources at the same time, while avoiding starved or deadlocked processes. The scheduling of processes with various resource requirements in this type of system is generally known as resource allocation. Processes that require the same resource are said to be in conflict. We consider the problem in the message passing model. A typical efficiency metric is the response time needed for a process to acquire all its required resources.

Examples of resource allocation can be found in many applications including distributed database systems and distributed (or parallel) I/O systems. In distributed database systems, a transaction accesses multiple data records during its operation. To maintain the consistency of the accessed data, a transaction usually locks the data records so that no other transactions can access them during its access. The locks on data records can be considered as resources to which a transaction needs to have exclusive access. In this system, efficient resource allocation schemes can be used to synchronize concurrent transactions.

This paper focuses fault-tolerant resource allocation techniques that can effectively localize an undetectable failure of a process. We measure the degree of fault-tolerance by the failure locality, which was first introduced by Choy and Singh [CS92]. The failure locality is the maximum number of processes whose liveness conditions (e.g., starvation) cannot be satisfied because of the crash failure of a single process.

The failure locality is an important measure in distributed systems because it measures the degree of distributedness of a concurrent system. For example, a centralized protocol has failure locality $n$ where $n$ is the total number of processes in the system. Although an algorithm is seemingly “distributed”, its failure locality may not be optimal.

To motivate our work, let us examine one of the early resource allocation algorithms, which was developed by Havender [Ha68].

In Havender’s algorithm, there exists a total ordering of priorities among resources, and processes request resources in the order of the priorities to avoid deadlock. Suppose that there are processes $p_1, p_2, \ldots, p_n$ contending for resources $r_1, r_2, \ldots, r_n, r_{n+1}$ which are in the descending order of priorities with $r_1$ being the resource with the highest priority, and each process $p_i$ requires $r_1$ and $r_{i+1}$. An execution may include a situation where each $p_i$ first acquires resource $r_1$, and then attempts to acquire $r_{i+1}$. Only $p_n$ is successful in the attempt because $r_{n+1}$ has not been taken by any process while others ($p_i$'s) are not because their requested resources ($r_{i+1}$'s) were already taken (by $p_{i+1}$'s). A waiting chain of $p_1, p_2, \ldots, p_n$, where $p_i$ waits for $p_{i+1}$ to free resource $r_{i+1}$, is formed (see Figure 1).

Now suppose that $p_n$ crashes while using resources $r_n$ and $r_{n+1}$. Since there is no way for $p_{n-1}$ to decide in the asynchronous system whether $p_n$ is simply slow or failed, $p_{n-1}$ keeps on waiting and so do others according to the algorithm. Therefore, the starvation of all processes occurs due to the failure of $p_n$. This algorithm has failure locality $n$. It shows that although an algorithm is seemingly “distributed”, it might have “the heel of Achilles” that when faulty, can bring the whole system to a halt.

Choy and Singh [CS93] showed the tight upper and lower bound ($\delta^*$) on the failure locality for a static version of this problem, namely the dining philosophers problem,
The system structure of our solution is similar to those in [WPP91, CS93, R95], in that there is one designated process for each resource, called the resource manager, that doles out resources to the requesting processes or maintains information about conflicting processes. In our solution, the resource managers also participate in scheduling accesses to resources. In Sections 2 and 3, we give a description of the model and problem, and in Section 4, present our algorithm. Section 5 gives the correctness proof.

Figure 1: A waiting chain of $n$. A solid lined arrow from resource $r_i$ to $r_j$ represents resource $r_i$ is held by $p_j$. A dotted lined arrow from $p_j$ to $r_{i+1}$ represents that $p_j$ is currently waiting for $r_{i+1}$.

Our definition of the dynamic resource allocation problem is more general in that it doesn’t assume a priori knowledge about the conflict processes.

The main contribution of our paper is the presentation of a dynamic resource allocation algorithm with optimal failure locality $\delta^*$. Our algorithm is based on an abstraction of bankers. From the example in Figure 1, we can see that a waiting chain of processes is formed when a process waits for a resource while it is holding another. Our algorithm breaks this chain in such a way that when a process cannot immediately use the credits (i.e., resources) that it is currently granted, it lends the credits to the other processes that might need them. Later, it collects the credits when it finally acquires all the credits that it has waited for. This way, a process may not be blocked because of some process that is also blocked.

The response time of this algorithm is $O(n(c + d))$. We leave as an open problem to find a dynamic algorithm with the optimal failure locality as well as a response time independent of $n$. However, it can be proved that if the message delay and the local computation are assumed to be negligible, our algorithm has response time $\delta^* c$. This measure bounds the amount of concurrency that one process can have: the execution of the critical region of one process cannot be concurrent with those of at most $\delta^2$ processes. This, in fact, upper-limits the impact of the critical region time ($c$) to the overall response time. Reducing the impact of $c$ on the response time can improve the overall performance of a system because $c$ can often be very large. For example, in such applications as distributed database or file systems, an access to each resource in the critical region requires I/O operations because shared resources are typically in secondary storage. In a typical local area network, I/O latency is more than 20 times larger than message delay. It is more evident in parallel processor systems, where I/O latency is considerably larger than interprocessor communication latency.

The system structure of our solution is similar to those in [WPP91, CS93, R95], in that there is one designated process for each resource, called the resource manager, that doles out resources to the requesting processes or maintains information about conflicting processes. In our solution, the resource managers also participate in scheduling accesses to resources.

In Sections 2 and 3, we give a description of the model and problem, and in Section 4, present our algorithm. Section 5 gives the correctness proof.

$$\log^* n = \min \{k : \log^k n \leq 1\}.$$
<table>
<thead>
<tr>
<th>Authors</th>
<th>Response Time</th>
<th>Failure Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Awerbuch and Saks [AS90]</td>
<td>$O(\delta c + \delta' \log</td>
<td>H</td>
</tr>
<tr>
<td>Weidman et al. [WPP91]</td>
<td>$O(\alpha c + nd)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Choy and Singh [CS93]</td>
<td>$O(\delta' c + (\delta' + 1) \log^*</td>
<td>H</td>
</tr>
<tr>
<td>Rhee [R95]</td>
<td>$O(\delta c + (\delta' + \delta \log^*</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 1: Previously known dynamic resource allocation algorithms.

2 The System Model

There exists a (finite or infinite) set of processes $P = \{p_1, p_2, p_3, \ldots\}$. Processes communicate by passing messages. There are four types of process steps: send, receive, local step and fail step. Send and receive are communication primitives and a local step changes local variables of processes. A fail step doesn’t involve any communication. Each process $p_i$ is modeled by a finite state automaton with state $Q_i$. The state set $Q_i$ includes an initial state $q_{0,i}$ and fail. The automaton for each process is specified by a single guarded command set $\{B_1 = A_1 \circ B_2 = A_2 \circ \ldots \circ B_m = A_m\}$. Each $B_i \rightarrow A_i$ is a guarded command, where a guard $B_i$ is a boolean expression and/or a message reception (receive step), and a finite list of action statements $A_i$ that consist of multiple local steps and/or one send step.

We model communication by a special process, called network, which schedules the delivery of messages sent among the other processes. Each process owns a buffer of infinite size. A buffer is an atomic read-modify-variable, so that a process can add a message to the contents of the buffer atomically. A send step adds a set of messages to the buffer of the network. A receive step of a process removes some messages from the buffer of the process. The receive step of the network removes some messages from its buffer and the send step of the network adds the messages to the buffers of their destination processes. (In this case, we say that the message is delivered to the destination). A process can send the same message to a finite set of processes in one step, i.e., a send step can add multiple copies of the same message with different destinations to the buffer of the network in one step.

A configuration is a vector $C = (q_1, q_2, \ldots)$ where $q_i$ is the local state of $p_i$ for each $p_i \in P$ (including the network); denote $state_i(C) = q_i$. A guarded command is enabled in a configuration if its associated boolean expression is true and associated receive, if any, can return nonempty messages, i.e., the messages specified in the receive are in the buffer of its process. An execution of a guarded command involves an atomic execution of all the steps in the action statements of the guarded command. It results in simultaneous changes to the state of the process of the guarded command based on the state of the process, and possibly the state of the network if the guarded command involves a send step. A guarded command enabled in a configuration $C$ can be applied to $C$ to yield a new configuration $C'$ as a result of the execution of the guarded command.

A system is specified by describing $P$, an initial configuration $C_0 = (q_{0,1}, q_{0,2}, \ldots)$, and the automaton of all processes in $P$. An execution sequence of a system is an infinite alternating sequence of configurations and enabled guarded commands $a = C_0, \pi_1, C_1, \pi_2, C_2, \ldots, C_i \pi_i, \ldots$, where $\pi_i$ is an enabled guarded command in $C_i$ and a fail step; $C_i$ is obtained by applying $\pi_i$ to $C_{i-1}$. If $\pi_i$ is not a fail step, then $q_i = fail$ in $C_{i-1}$ if and only if $q_i = fail$ in $C_i$. This means that the automaton of a process cannot cause the process to leave the fail state, and the automaton cannot cause a process to enter the fail state. We say that in an execution sequence, a guarded command is continuously enabled from $C_i$ to $C_j$, $i < j$ if the guarded command is enabled in every configuration from $C_i$ to $C_j$ and is not applied to any configuration in between $C_i$ and $C_{i+1}$. We also say that in an execution sequence $\sigma$, a guarded command $g$ is enabled before a guarded command $g'$ in $C_i$ if there exists a sequence of configurations $C_i, C_k$ in $\sigma$ such that $g$ is continuously enabled from $C_i$ to $C_k$, but $g'$ is not continuously enabled from $C_i$ to $C_j$.

An execution $(\sigma, T) = C_0, (\pi_1, t_1), \ldots, (\pi_j, t_j), \ldots$ satisfies the followings: (1) $\sigma = C_0, \pi_1, C_1, \pi_2, C_2, \ldots$ is an execution sequence; (2) if $\pi_i$ is a guarded command of process $p$ (that is not the network) and applied to $C_i$, then there is no guarded command of $p$ that is enabled before $\pi_i$ in $C_i$; (3) if $\pi_i$ is a fail step and applied to a process $j$, then $state_j(C_{i-1}) \neq fail$ and $state_j(C_i) = fail$; (4) all enabled guarded command will be executed eventually; (5) messages are received in the order that they are sent (i.e., FIFO) and all messages sent are received eventually; (6) $T$ is a mapping from guarded commands to real numbers that associates a real time with each guarded command in the execution. The sequence $t_0, t_1, \ldots, t_i \ldots$ is nondecreasing and unbounded.

Note that the system model here is completely asynchronous because there are no constraints on relative timing of process steps and message delays.

3 The Dynamic Resource Allocation Problem

We now specialize the general system model in Section 2 for resource allocation problem. Since we don’t put any restriction on the behavior of resources and we treat them
as passive entity\(^3\), we don’t model resources as physical objects in our system. We simply treat them as an additional system parameter. Let \( R \) be the set of all resources in systems. There exists a set of processes \( U (\subseteq P) \) called the users that need a subset of \( R \) for their execution at various times. Let \( R_i(t) \) be the resource requirement of user \( i \) at time \( t \).

Each user’s local states are partitioned into four regions. In the trying region, the user requests its required resources. Once acquiring the resources, the user enters the critical region. It remains in the region for a finite time using the resources. When the user is finished with the resources, it enters the exit region, where it relinquishes the resources. Otherwise, the user is in the remainder region.

Once a user enters the trying region, it doesn’t change its resource requirement until it enters the following remainder region. A user requests resources only in the trying region. At some time \( t \), if users \( i \) and \( j \) are in the trying or critical region, and \( R_i(t) \cap R_j(t) \neq \emptyset \), then we say that user \( i \) conflicts with user \( j \) at time \( t \).

A dynamic resource allocation algorithm is a specification of a system with an infinite set \( U \) where for all users \( i \), \( R_i \) is a priori unknown to all processes (i.e., \( R_i \) cannot be built into the code of processes). The algorithm must satisfy the following two conditions: (1) (exclusion) in any execution of the algorithm, if users \( i \) and \( j \) are both in the critical region at time \( t \), then \( i \) and \( j \) do not conflict with each other; (2) (no-lockout) in any execution of the algorithm that no process fails in, if a user is in the trying or exit region, then it leaves its current region in finite time.

The failure locality of an algorithm is defined as follows. Let \( \mathcal{E} \) be the set of executions of the algorithm in which only one process fails. Let \( f_e \) be the number of users that remain in the trying region forever in an execution \( e \in \mathcal{E} \). The failure locality is \( \max \{ f_e : \forall e \in \mathcal{E} \} \). The response time of an algorithm is defined to be the maximum time period between when a user enters the trying region and when the user enters the following critical region for all executions of the algorithm.

4 Algorithm

4.1 A Priority Assignment Algorithm

Our algorithm is based on a partial order relation \( <_{pr} \) on users, representing priority. If \( i <_{pr} j \) for two users \( i \) and \( j \), we say that \( i \) has a lower priority than \( j \), or \( j \) has a higher priority than \( i \). \( <_{pr} \) should ensure that (i) (uniqueness) any two conflicting users have different priorities, and the priority relation is closed under transitivity, and (ii) (finiteness) there exists a finite time \( t \) such that if a user is in the trying region for time \( t \), then it has a higher priority than any user entering the trying region after time \( t + t' \) where \( t' \) is the time that \( i \) entered the trying region last time.

We present a simple priority assignment algorithm that satisfies the uniqueness and finiteness conditions. It is based on Lamport’s logical clock [La78].

A user \( i \) entering the trying region sends a get-priority message to each resource manager \( rm_r \), \( r \in R_i \). Upon receiving a get-priority, \( rm_r \) increments its counter, which is initially set to 0, and then sends back the value of its counter. Receiving the counter values of all resource managers \( rm_r \), \( r \in R_i \) sets its priority value to the maximum of those values. Then it sends back its new priority value to the resource managers. Resource managers reset their counter to the received value only if the value is bigger than its current counter value. We say \( i <_{pr} j \) only if the priority value of \( i \) is bigger than that of a conflicting user \( j \). The symmetry between any two conflicting users with the same priority value can be broken by their IDs which are assumed to be unique. Once set, the priority value of a user is fixed until the user enters the trying region again.

Since each user chooses only one value for its priority and doesn’t change it until the next time it enters the trying region, it ensures the uniqueness condition. Any conflicting user \( j \) of \( i \) entering the trying region after time \( 3d \) since \( i \) did must have a higher priority than \( i \) because the counter values of a \( rm_r \), \( r \in R_i \cap R_j \), must be at least equal to the priority value of \( i \) by the time that \( j \) enters the trying region. Thus, it guarantees the finiteness condition.

The algorithm can be replaced by a simple time stamping algorithm using approximately synchronized clocks (if they are available). The presented algorithm, however, is designed to work in asynchronous systems.

4.2 A Simple Dynamic Resource Allocation Algorithm

We first present a simple dynamic resource allocation algorithm with failure locality \( n \). Our optimal algorithm (described in Section 4.3) is modified from this algorithm. We assume that the priority of each user entering the trying region is assigned by the algorithm in Section 4.1.

A user \( i \) entering the trying region sends a request message to each resource manager \( rm_r \), \( r \in R_i \). Upon receiving the request message, \( rm_r \) adds the (ID of) user \( i \) to its queue \( q_r \). Then, if no user is currently granted resource \( r \), it sends a grant message immediately to \( i \). Receiving a grant message from \( rm_r \), \( i \) adds \( r \) to its grant-set. If there is a user that is currently granted the resource, let \( j \) be the user. If \( j <_{pr} i \), then \( rm_r \) sends a preempt message to \( j \). If \( i <_{pr} j \), it does nothing. If \( j \) is not in the critical region when it received the preempt, \( j \) returns resource \( r \) to \( rm_r \) by sending a return message and removing \( r \) from its grant-set. Upon receiving the return message, \( rm_r \) sends a grant message to a user with the highest priority among the users in the queue. A user leaving the critical region sends a release message to each resource managers of the resources that it used, and sets its grant-set to the empty set. Receiving a release message from \( j \), \( rm_r \) removes \( j \) from its queue and then, sends a grant message to a user with the highest priority among the users in the queue. When receiving grant messages from \( rm_r \), \( r \in R_i \) (i.e., when grant-set \( = R_j \)), \( i \) enters the critical region.
Because of the finiteness condition of the priority assignment algorithm, there will be a finite set of conflicting users that have higher priorities than a user \(i\) after some time (3d) since \(i\) entered the trying region. It can be shown that a user with the highest priority among all its conflicting users eventually gets grant messages from all the resource managers of the resources that it requires. An induction on the size of the set can show that user \(i\) eventually gets grant messages from all the resource managers in \(R_i\), and enters the critical region. This guarantees the no-lockout condition. It is also straightforward to show that the algorithm satisfies the mutual exclusion condition.

The failure locality of the algorithm is \(n\). To see this, suppose that there are users \(i\), \(j\) and \(k\) such that \(k <_{pr} j <_{pr} i\); \(i\) requires resource \(a\); \(j\) requires resources \(a\) and \(b\); and \(k\) requires resources \(b\) and \(c\). A situation can occur in which \(i\) gets granted resource \(a\) and is in the critical region; \(j\) holds resource \(b\) and waiting for resource \(a\); and \(k\) holds resource \(c\) waiting for resource \(b\). Since \(k <_{pr} j\), \(k\) cannot preempt \(b\) from \(j\). Figure 2 shows the queues of resources \(a\), \(b\) and \(c\) (the users in the queues are in order of their priorities). The waiting chain where \(k\) waits for \(j\), and \(j\) waits for \(i\) may grow to involve all users in the system. Clearly, if the user at the front of this chain fails in the critical region, then every other user will starve.

4.3 An Optimal Algorithm: Distributed Bankers

We now modify the simple algorithm in Section 4.2 to obtain an optimal failure locality algorithm.

A problem with the simple algorithm is that a user is allowed to wait for a resource while holding other resources. For example, in Figure 2, user \(j\) waits for resource \(a\) while holding resource \(b\), and therefore, blocks user \(k\) unnecessarily. This unnecessary blocking may cause the length of a waiting chain to grow up to \(\Omega(n)\). Clearly, if \(i\) fails in the critical region, then it is impossible for \(j\) to use resource \(b\) because it cannot decide whether \(i\) is simply being slow or failed. But, it should still be possible for \(k\) to use resource \(b\) if we can make \(j\) give up \(b\).

As we stated in the introduction, our optimal algorithm uses an abstraction of bankers. We use resource managers as bankers as follows. When a user \(i\) receives a grant message from resource manager \(rm_i\), if there is at least one resource manager that it hasn’t received a grant message from, then user \(i\) deposits the resource back to \(rm_i\) by sending a deposit message; removing \(r\) from grant-set; and adding it to its deposit-set. Upon receiving a deposit message, \(rm_i\) records the fact by adding \(i\) in its lender-set.

Then it grants the resource to the next pending user with the highest priority among the users that haven’t been granted the resource (i.e., those not in its lender-set, but in its queue). When a user \(i\) has finally received a grant message from each \(rm\), (i.e., grant-set \(\cup\) deposit-set = \(R_i\)), it starts collecting the resources in deposit-set by sending request messages to the managers of the resources in deposit-set. However, a problem may arise if users start to collect resources at the same time. Suppose that users \(i\), \(j\), and \(k\) \((k <_{pr} j <_{pr} i)\) require resources \(a\) and \(b\), resources \(c\) and \(d\), and resources \(e\) and \(f\) respectively. We can imagine a situation illustrated in Figure 3(a) where resource \(a\) is granted to some user (with a higher priority than \(i\)) and \(i\) is waiting for resource \(e\); the same is true with \(j\) and \(k\) involving resources \(d\) and \(e\) respectively. Of the waiting, \(i\), \(j\), and \(k\) have to deposit resources \(b\) and \(c\) when the resources were granted to them. Some time later, as shown in Figure 3(b), when \(i\), \(j\), and \(k\) are finally granted resources \(e\), \(d\), and \(e\) respectively. Users \(i\), \(j\), and \(k\) try to collect their deposited resources. Since \(j <_{pr} i\), and \(k <_{pr} j\), \(i\) and \(j\) get resources \(b\) and \(e\) respectively. Now, \(i\) enters the critical region while \(k\) and \(j\) cannot because \(j\) doesn’t have \(b\) and \(k\) doesn’t have \(c\). This time, according to the algorithm described so far, because grant-set \(\cup\) deposit-set = \(R_i\), \(j\) doesn’t deposit resources in its grant-set. Neither does \(k\). As a result, \(j\) is waiting for \(i\) to release \(b\) while holding \(c\), and \(k\) is waiting for \(j\) to release \(e\) while holding \(c\). This chain may grow to involve other users (for example, some user \(m\) in \(q\). in Figure 3(b)).

To solve the problem, we again turn to the abstraction of bankers. When a user \(j\) is granted a resource which is deposited by another user \(i\) with a higher priority, it should be considered that \(j\) is borrowing the credit (resource) from \(i\). Thus, when \(i\) collects the credit, \(j\) is no longer a lender (because it lost its credit to the original lender).

As a solution, when a resource manager \(rm_j\) grants a resource \(r\) to some user \(i\), \(rm_j\) sends preempt messages to all users in its lender-set that have a lower priority than \(i\) (in the example, to \(j\)). Then \(rm_i\) removes the users from its lender-set. A user receiving a preempt message should not think any more that it has granted and deposited the resource. So it removes \(r\) from its deposit-set. In order to get resource \(r\), it has to request the resource again. In Figure 3(b), based on the new scheme, deposit-set and grant-set do not contain resource \(b\) any more (i.e., grant-set \(\cup\) deposit-set \(\neq R_i\)). Thus, \(j\) has to deposit \(c\), making it available to \(k\). \(k\) now has \(c\) and \(e\), and enters the critical region. The long waiting chain is finally broken.

The failure locality of the above algorithm is \(\Theta^2\). The worst case which leads to \(\Theta^2\) failure locality can happen when a user waits for a lower priority user to leave the critical region. Imagine a situation as in Figure 4, where \(i\) is waiting for resource \(a\) while \(j\) is in the critical region using \(a\) and \(j <_{pr} i\). Since \(i\) is currently granted \(b\) and
Theorem 5.1 (Exclusion) In any execution of the algorithm, if users \( i \) and \( j \) are both in the critical region at time \( t \), then \( i \) and \( j \) do not conflict with each other.

Proof: (Sketch) according to the algorithm, it can be shown that for a resource \( r \in R_i \cap R_j \), if \( r \notin grant-set_i \), then \( r \notin grant-set_j \). User \( i \) enters the critical region only when \( grant-set_i \) is equal to \( R_i \) and \( grant-set_j \) is not changed while \( i \) is in the critical region. Therefore, if \( i \) and \( j \) are in conflict, they cannot be in the critical region at the same time.

To prove the no-lockout condition, we needed the following lemma. Let \( A_i \) be the set of users that have a higher priority than user \( i \).

Lemma 5.2 Let \( p \) be one of the users in \( A_i \) that have the highest priority among all its conflicting users. \( p \) enters the critical region eventually.

Proof: If \( request_p \neq R_p \), \( p \) will send a request to all resource managers of resources not in \( request \).

After receiving \( request_p \), a resource manager \( rm_r \), \( r \in R_p \), either (1) sends a grant message to \( p \) or (2) sends a preempt message to user \( p' \) if \( r \) is already granted to \( p' \) (because \( p' < p \)). After receiving the preempt message, \( p' \) will send either a release message (when it exits the region if it is in the critical region), or a return message. After receiving any of the messages, resource manager \( rm_r \) will send a grant message to \( p \). After receiving a grant message from \( rm_r \), user \( p \) will deposit \( r \) if \( grant-set_p \cup deposit-set_p \neq R_p \) and deposit-set will contain \( k \). Because \( p \) is the highest priority user, eventually \( grant-set_p \cup deposit-set_p = R_p \).

Then, according to the algorithm, if \( grant-set_p \cup deposit-set_p = R_p \), then \( p \) sends request messages for resources in \( deposit-set_p \). This time, \( p \) does not deposit those resources that are in its \( grant-set_p \) and are granted after sending the request messages. Because \( p \) has the highest priority among all its conflicting user and by the same argument as the above, eventually \( grant-set_p \) will be equal to \( R_p \) and \( p \) enters the critical region.

Theorem 5.3 (No-lockout) In any execution of the algorithm that no process fails in, if a user is in the trying or exit region, then it leaves its current region in finite time.


**Proof:** (Sketch) By the finiteness condition of the priority assignment algorithm, there is a finite set \( A_i \) of the users that have a higher priority than a user \( i \) in the trying region, and no new user will be added to \( A_i \) eventually. Let \( p \) be one of the users in \( A_i \) that have the highest priority among all its conflicting users (there is at least one such user because of the transitivity of \(<_p\)). By Lemma 5.2, \( p \) enters the critical region eventually. \( p \) also leaves the critical region eventually because \( p \) is not faulty, and we can take \( p \) out of \( A_i \). An induction on the size of \( A_i \) proves the theorem.

To prove the optimal failure locality, we needed the following lemma.

**Lemma 5.4** If \( p \) is a user at distance bigger than 2 and whose priority is higher than any of its conflicting users at distance bigger than 2, then \( p \) eventually enters the critical region.

**Proof:** Since \( p \) is at distance at least 3, its conflicting users never fail. The only case that \( p \) cannot make progress is that there is some user \( p' \) that holds a resource \( r \in R_p \cap R_{p'} \) forever. If \( p' \) can keep holding \( r \), then \( p' \) must have a higher priority than \( p \). This is because if \( p' \) has a lower priority than \( p \), then \( p \) can get \( r \) from \( p' \) (either \( p \) preempts it from \( p' \) if \( p' \) is not in the critical region, or it is released by \( p' \) when it leaves the critical region if it is in the critical region. \( p' \) leaves the critical region eventually because \( p' \) is not faulty). If \( p' \) keeps holding \( r \), according to the algorithm, \( t \) must be that \( p <_p p' \); \( \text{deposit-set}_{p'} \cup \text{grant-set}_{p'} = R_{p'} \); \( \text{deposit-set}_{p'} \neq \emptyset \); and \( r \in \text{grant-set}_{p'} \). If there is such \( p' \), we say that \( p' \) obstructs \( p \) (through \( r \)). The lemma follows from proving that there is no user that obstructs \( p \).

By way of contradiction, assume that there is a user \( i \) that obstructs \( p \). We prove that \( \text{grant-set}_i \) eventually becomes equal to \( R_i \). This is a contradiction.

Let \( r \in R_i \) be the resource that is missing from \( \text{grant-set}_i \). Then there should be a user \( j \) such that \( r \) is in \( \text{grant-set}_j \) forever because \( i \) obstructs \( p \) by the algorithm. We prove that there is no such \( j \).

By the hypothesis of the lemma, \( i \) must be at distance 2 (because \( p <_p i \), \( p \) is at distance at least 3, and has a higher priority than any of its conflicting users whose distances are bigger than 2).

Case 1: \( i <_p j \). When a resource is granted to \( j \), \( i \) must
receive a *preempt* since $r$ is in $\text{deposit-set}_i$, and $i <_{pr} j$. As a result, $i$ will remove $r$ from $\text{deposit-set}_i$. So, it is not true that $i <_{pr} j$.

Case 2: $j <_{pr} i$.

When $i$ requests for $r$, $j$ must receive a *preempt* message from $rm_i$, according to the algorithm, since $r$ is in $\text{deposit-set}_i$, and $\text{deposit-set}_i \cup \text{grant-set}_i = R_i$. Therefore, $j$ eventually has to remove $r$ from $\text{grant-set}_j$ because $j$ is not faulty. Thus, it is not true that $i <_{pr} j$.

Therefore, it cannot be either $i <_{pr} j$ or $j <_{pr} i$. This is a contradiction of the uniqueness condition of our priority assignment technique because $i$ and $j$ are in conflict.

**Theorem 5.5** The distributed banker algorithm has failure locality $\delta^2$.

**Proof:** (Sketch) According to the algorithm, when a user with a resource $r$ is in the critical region, resource manager $rm_r$ do not send any message to other users until it receives a *release* message from the user. Thus, the failure of a resource manager $rm_r$, has the same effect on the other users as one user $i$, s.t. $R_i = \{r\}$, failing in the critical region. Therefore, it suffices to show the case for one user failure.

Before we prove the theorem, we define the following.

We define the distance of a user $p$ to be the minimum $k$ such that there is a sequence of users $p, p_1, p_2, \ldots, p_k$ where $p$ and $p_i$ are in conflict; $p_i$ and $p_{i+1}$ are in conflict; $p_i$ and $p_j, j > i + 1$, are not in conflict; $p_i, i < k$, is not faulty; and $p_k$ is faulty. To prove the theorem, we only need to show that if $p$ is a user whose distance is bigger than 2, $p$ enters the critical region eventually.

By Lemma 5.4, if $p$ is a user at distance bigger than 2 and whose priority is higher than any of its conflicting users at distance bigger than 2, then $p$ eventually enters the critical region. An Induction on the number of users whose priorities are higher than $p$ and whose distances are bigger than 2 as in the proof of the no-lockout condition proves the theorem.

**Theorem 5.6** If the message delay and the local computation time is 0, the response time of the algorithm is $\delta^2 c$.

**Proof:** (Sketch) Let $p$ be the user that enters the trying region at time $t$. It is clear from the algorithm that if $p$ doesn’t receive a *grant* message from any resource manager of resource $r \in R_p$, $r$ must be in the *grant-set* of some user $p’$. If $p’ <_{pr} p$, then $p’$ must be in the critical region. If $p <_{pr} p’$, then either $p’$ is in the critical region, or $p’$ is waiting for some other user $p''$ (and $p <_{pr} p''$) to finish the critical region while holding $r$ in its *grant-set* (in which case, $\text{deposit-set}_{p’} \cup \text{grant-set}_{p’} = R_{p’}$, $\exists t’$ s.t. $t’ \in \text{deposit-set}_{p’}$ and $t’ \in \text{grant-set}_{p’}$; and $r \in \text{grant-set}_{p’}$). Since there can be at most $\delta^2$ such users as $p’$ and $p’’$, $p$ enters the critical region within $\delta^2 c$.

**References**


