Resource Allocation on Dynamic Conflict Graphs*

Manhoi Choy and *Ambuj K. Singh*
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California 93106

Abstract

The sharing of resources among processes in a distributed system leads to a conflict graph that may change with time. Resource allocation over a static conflict graph (also called the dining philosophers problem) has been studied extensively. We seek to solve resource allocation on dynamic conflict graphs by using existing algorithms that work only for static conflict graphs. In the process we define the notion of a snapshot of a dynamic graph, specify its properties, and show how it can be combined with static graph algorithms to yield efficient solutions for dynamic graphs.

Index Terms: Distributed Algorithms, Dynamic Grpahs, Graph Coloring, Resource Allocation

1 Introduction

In modern computer systems, resources are usually distributed and shared among a number of users. Since the resources are usually used in an exclusive manner, efficient resource allocation algorithm are needed to minimize the amount of waiting and to improve the throughput of the system. The distributed job scheduling problem [2] is an abstraction of the resource allocation problem in distributed systems. This problem consists of a number of processes and a number of resources. From time to time a process issues a job request for the exclusive usage of a subset of the resources. Two jobs *conflict* with each other if they request for a common resource. A job can be started only after the required resources are granted, and once started it is assumed to finish eventually. The basic requirement of the problem is that every job must be started eventually.

The job scheduling problem is crucial in the efficient implementation of concurrent object-oriented systems. For example, in extending the object oriented language *Eiffel* for concurrent execution [11], resource requesters (called *features* in Eiffel or *methods* in object oriented database systems) may be concurrently requesting for the use of a set of objects to start their execution. The required objects must then be allocated to the requesters in an exclusive manner. This allows the design methodologies for sequential procedures (or features) to be used for the design of concurrent programs. However, due to the dynamic creation of the objects as well as of the requesters, the allocation of objects in an efficient and starvation-free manner is difficult.

Resources are usually controlled by resource managers. In a small and centralized system, it is usually more efficient to use a single resource manager to control the use of all the resources. However, in a large or a distributed system, the use of a single resource manager may become the bottleneck of the system and hinder the concurrent execution of independent

*A preliminary version of this paper appears in the Proceedings of the 7th International Workshop on Distributed Algorithms.

jobs. In this paper, we assume that there is one resource manager per resource. The user processes communicate with each other and with the resource managers for the scheduling of the resources. Processes and resource managers communicate with each other by sending messages on reliable FIFO communication channels. In order to measure the responsiveness of the system, we assume that there is an upper bound ν on the message delivery time and an upper bound τ on the duration of a job. These bounds are used only in the performance analysis of the algorithms.

The graph induced by the conflict relation of jobs is called the *conflict graph*. The degree of conflict of a job p, denoted by δ_p, is the maximum number of other jobs that are currently conflicting with p at any time during which p exists in the system. The degree of the conflict graph, denoted by δ, is the maximum among all the δ_p's. Since job requests may enter the system at any time, conflict graphs are *dynamic*. This is in contrary to the assumption of the dining/drinking philosophers problems [6, 8, 10] in which the conflict graph is fixed in advance. Although the conflict graph changes with time, the number of jobs in the system is bounded by the total number of processes N at any time.

A number of performance measures exist in the literature for the resource allocation problem. The ones that are considered in this paper are *response time* and *message complexity*. Response time measures the time delay between a job request and its servicing by the granting of desired resources. Message complexity measures the number of messages sent or received for a job request. While it is clear that response time depends on the values of ν and τ, it may also depend on the values of δ and N. Assuming ν and τ are constants, an obvious lower bound of $\Omega(\delta)$ on the response time of any solution to the job scheduling problem is obtained in the case where there are $\delta+1$ mutually conflicting jobs. A less obvious lower bound of $\Omega(\log^* N)$ can also be shown by reducing the $(\delta+1)$-*coloring* problem to the job scheduling problem. In this reduction, the graph to be colored is used as the conflict graph in the job scheduling problem. When a job is scheduled, a color is chosen and assigned to the corresponding node so that the colors between neighboring nodes are different. Since an $\Omega(\log^* N)$ lower bound exists for the $(\delta+1)$-*coloring* problem [9], any job scheduling algorithm also has an $\Omega(\log^* N)$ response time. In practice, jobs usually request for a small number of resources and the degree of the resulting conflict graph is small compared with the value of N. It is therefore desirable to have solutions to the distributed job scheduling problem whose response time depends on δ but not on N.

In the existing literature on the dining philosophers problem, Lynch [10] is perhaps the first one who considered response time explicitly. Lynch's algorithm, which uses a precomputed edge coloring, achieves a response time of $O(c^\delta)$ where c is the number of colors used to color the edges of the underlying conflict graph. The algorithm has a message complexity of $O(\delta)$. Styer and Peterson [13] extended the edge coloring algorithm of Lynch and gave solutions in which the longest waiting chain of processes that can exist in the system is of $O(log\ \delta)$. However, the response time and the message complexity of their solution are both $O(\delta^{log \delta + 1})$. Awerbuch and Saks [2] achieved a breakthrough in response time. They used a distributed queue to solve the distributed job scheduling problem achieving a response time of $O(\delta^2 log\ N)$ and a message complexity of $O(\delta^2 log\ N)$. Choy and Singh [7] presented a solution to the dining philosophers problem with a response time of $O(\delta^2)$ and a message complexity of $O(\delta)$ assuming an initial $(\delta+1)$-*coloring*, that assigned a legal color in the range $0 \cdots \delta$ to each node of the conflict graph. Recently, Bar-Ilan and Peleg [4] modified the distributed queue algorithm of Awerbuch and Saks and obtained an algorithm for the distributed job scheduling problem with a response time of $O(\delta \log N)$ and a message complexity of $O(\delta \log N)$ in a *synchronous complete* communication network. Another recent solution to the dining philosophers problem by Page, Jacob, and Chern [12] achieves a response time of $O(\delta^2)$ if the dependence on ν is ignored.

In this paper, we present a new method for solving dynamic graph problems in which the input graph is allowed to change dynamically. The new technique is based on the idea of *snapshots*. This terminology should not be confused with the snapshot of a distributed computation as considered by Chandy and Lamport [5]. In the problem that they consider, the goal is to compute a possible global state of a given underlying distributed computation occurring over a static graph of processes. On the other hand, a snapshot in our terminology is a stable subgraph of a dynamic graph of processes. A snapshot is composed of a set of local snapshots where the local snapshot of a node is simply a set of neighboring nodes. In general, once a snapshot is taken on a dynamically changing graph, static algorithms can be applied on the snapshot. Thus, computing on a *snapshot* allows us to use existing distributed algorithms to solve problems for dynamic graphs. Using this method, we design an algorithm for the distributed job scheduling problem with a response time of $O(\delta^3 + \delta \log^* N)$ and message complexity of $O(\delta^2 + \delta \log^* N)$. The response time of our algorithm improves that of the best known result, due to Awerbuch and Saks [2], when δ is less than $\log N$, and the message complexity of our algorithm improves that of Awerbuch and Saks's algorithm by a factor of $\log N$. Most importantly however, our algorithm minimizes the dependence of the complexity measures on the global parameters.

This paper is organized as follows. In Section 2, we specify the snapshot problem and in Section 3, we present two algorithms for the problem. In Section 4, we use the idea of snapshots to solve the distributed job scheduling problem. Concluding remarks appear in Section 5 and the appendix includes the proofs of some theorems used in the paper.

2 The Snapshot Problem

In a dynamic graph problem, processes are assigned to the nodes of a graph $G = (V, E)$. Each process has a computation to be performed that depends on the topology of the graph. Graph G is dynamic as new nodes (processes) may be added to the graph at any time. Also, a node (a process) may be deleted from the graph after the computation is finished. The addition of a node may also require the addition of a number of edges. When a node is deleted from G, all its adjoining edges are also deleted.

Formally, a variable $state_p$ that records the current state of process p is associated with each process p. This variable can take one of four different values: *idle*, *trying*, *executing*, and *completed*. The state transition diagram of a process p is shown in Figure 1. Each iteration of a process p through the above sequence of four states defines a job. A process may be associated with a number of different jobs during its lifetime. However, at any given time, it is associated with at most one job. We use upper case letters to denote jobs; for example, P denotes a job associated with process p, Q denotes a job associated with process q, and so on. A job P is said to be *active* if it corresponds to the current iteration of process p; otherwise, it is said to be *inactive*. The state of an active job is specified by the state of the process with which it is associated.

In addition to variable $state_p$, another variable $Conf_p$ that stores the set of neighboring processes of process p is associated with process p. Initially, a process is at *idle* and $Conf_p$ is empty. When the process decides to issue a job request, it obtains a set of neighboring processes $Conf_p$, sends $\langle add \rangle$ messages to each process in $Conf_p$, and transits to *trying*. On receiving an $\langle add \rangle$ message from p, process q adds p to set $Conf_q$. At state *trying*, a process

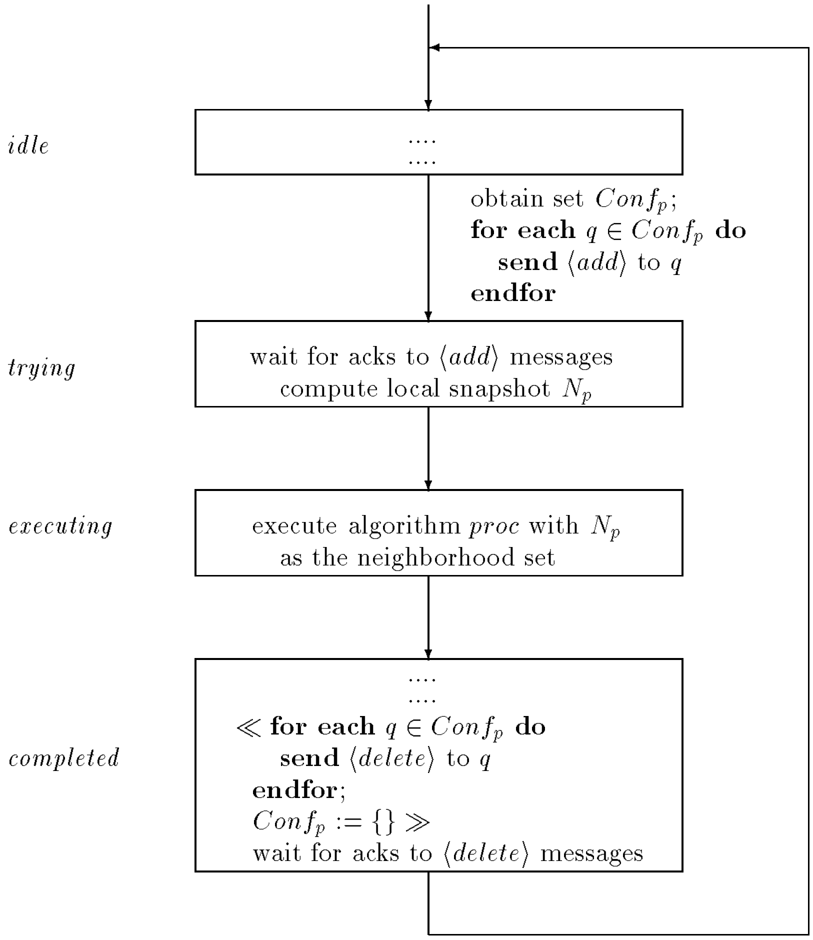

Figure 1: User process p

waits until all its $\langle add \rangle$ messages are received and then computes its *local snapshot* N_p. Then it transits to *executing* and executes a user-specified algorithm *proc* with N_p as the neighborhood. Upon completion of *proc*, process p transits to *completed*. At this state process p sends $\langle delete \rangle$ messages to each process in $Conf_p$, and resets $Conf_p$ to empty in an atomic step. Then process p waits until all the $\langle delete \rangle$ messages are received, and transits to *idle*. On receiving a $\langle delete \rangle$ message from p, process q removes p from set $Conf_q$. For simplicity, the message handlers for $\langle add \rangle$ and $\langle delete \rangle$ messages are omitted.

The set of processes that process p obtains in $Conf_p$ while transiting to *trying* includes all processes from which it has received $\langle add \rangle$ messages but no $\langle delete \rangle$ messages, as well as processes that it finds to be conflicting with itself by querying the resource managers. This set changes dynamically by the reception of $\langle add \rangle$ messages from new processes and $\langle delete \rangle$ messages from processes that are already in $Conf_p$, Thus, $Conf_p$ defines the neighbors of p in the dynamic graph at any given time. The maximum cardinality of the set $Conf_p$ at any time is denoted by δ_p.

The input to the snapshot problem consist of a set of user processes of the form shown in Figure 1. The requirement of the snapshot problem is to define the computation of *local snapshots* by a process at *trying* and the appropriate message handlers. Before we present the formal specification, we state a few theorems based on the user outline in Figure 1. The proofs of these theorems can be found in the appendix.

Theorem 1 *For any two active jobs P and Q, if $q \in Conf_p$ as a result of receiving an $\langle add \rangle$*

message from process q after P transited to trying, and P has not sent any $\langle delete \rangle$ messages then job Q sent the $\langle add \rangle$ message to job P.

Theorem 2 *For any two active jobs P and Q that are at trying, executing or completed, and that have not sent any $\langle delete \rangle$ messages, if $q \in Conf_p$ is true then it holds until either P or Q send $\langle delete \rangle$ messages.*

Theorem 3 *If P is a job at trying, executing, or completed and Q is a job at executing or completed, and if the two jobs have not sent any $\langle delete \rangle$ messages then $p \in Conf_q$ implies $q \in Conf_p$.*

Theorem 4 *For any two active jobs P and Q that are at executing or completed and that have not sent any $\langle delete \rangle$ messages, if $q \in Conf_p$ then $p \in Conf_q \wedge q \in Conf_p$ holds after both P and Q transit to executing.*

For the formal specification of the snapshot problem, we first state two safety properties that relate $Conf_p$ and N_p. The first property called *non-triviality* states that two processes that are neighbors of each other in the dynamic graph must include each other in their local snapshots if they are at *executing* simultaneously. This property bounds the local snapshots from below and ensures that all necessary conflicts are reflected in the local snapshots.

Property 1 (*non-triviality) At any global state, if P and Q are two jobs at executing such that $p \in Conf_q \wedge q \in Conf_p$, then $p \in N_q \wedge q \in N_p$.*

In order to relate snapshots to the topology of the dynamic graph, we introduce a second safety property called *validity*. It bounds the local snapshots from above ($|N_p| \leq \delta_p$) and ensures that no unneeded conflicts are captured by the local snapshots.

Property 2 (*validity) At any global state, if P is a job at executing such that $q \in N_p$ then there exists a previous global state at which P is at trying, q is at trying, $p \in Conf_q$ and $q \in Conf_p$.*

The specification has so far included only safety properties. A snapshot algorithm should also satisfy certain progress conditions concerning the termination of snapshot computation. The seemingly obvious requirement that each process eventually complete its snapshot computation (i.e., a process at *trying* eventually transits to *executing*) is too strong. It is possible to show that such a progress requirement is unsatisfiable when considered in conjunction with *non-triviality* and *validity*. For a proof consider a history in which job P transits to *executing* before job Q transits to *trying* with $p \in Conf_q$. Assume that P remains at *executing* forever. Now, if an algorithm can ensure that Q eventually transits to *executing* then at the resulting state $p \in Conf_q \ \wedge \ q \in Conf_p$ holds. Therefore, by *non-triviality*, $p \in N_q \ \wedge \ q \in N_p$ should hold. However, this violates *validity* as the computation of local snapshots of P and Q did not overlap.

On account of the above impossibility result, we need to weaken the requirement of local progress. There are a number of ways to weaken this requirement based on the possible of instances of the user-specified computation *proc*. In our investigation, we have encountered two classes of user-specified computations – one in which a user process p at *executing* transits to *completed* irrespective of the states of the processes in its local snapshot and another in which this transition can only occur after all processes in the local snapshot of p have completed the computation of their local snapshots. An instance of the former kind is a dining philosophers algorithm where a process becomes *hungry* only after computation of the local snapshot and an instance of the latter kind is a node coloring algorithm where a process $q \in N_p$ needs to communicate its color to process p in order for process p to be colored correctly. For the former class of problems, progress is stated as the following conditional property.

Property 3 (*conditional local progress) If every job at executing eventually transits to completed then every job at trying eventually transits to executing.*

For the latter class of problems, we require a combination of *global progress* and *no blocking* conditions. The *global progress* condition states that in every neighborhood of the conflict graph, some process eventually completes the computation of its local snapshot.

Property 4 (*global progress) For any process p at trying there eventually exists a process* $\in Conf_p \cup \{p\}$ *at* executing.

The *no blocking* condition requires that a process, which is in the local snapshot of another process executing *proc*, also begins executing *proc* eventually. It is stated as follows.

Property 5 (*no blocking) For any job P at executing and any job Q at trying such that $q \in N_p$, if $q \in Conf_p$ has held continuously since P transited to executing, then eventually either P transits to completed or Q transits to executing.*

Note that if job P in the above statement waits at *executing* until Q transits to *executing* then the no blocking condition guarantees that Q eventually transits to *executing*. In the next section, we present two algorithms for the snapshot problem, one satisfying *conditional local progress* and the other satisfying all three progress conditions.

3 Algorithms for the Snapshot Problem

We first present a simple algorithm $S1$ that satisfies the conditional local progress condition. Later, we present a more complex algorithm $S2$ that satisfies the *global progress* and *no blocking* conditions in addition to *conditional local progress*. Algorithm $S1$ is interesting and useful because it satisfies the *conditional local progress* condition in a more efficient manner.

3.1 Snapshot Algorithm $S1$

Figure 2 describes an algorithm for the snapshot problem satisfying the conditional local progress condition. On transiting to *trying*, process p copies $Conf_p$ to N_p and sends a $\langle req \rangle$ message to all processes in N_p. Then it waits until a $\langle req \rangle$ message is received from each process in set N_p. Upon receiving a $\langle req \rangle$ message from process q, if process p is at *trying* and q is not in N_p, then process q is added to N_p and a $\langle req \rangle$ message is returned to process q. Otherwise, if process p is at *idle* or *completed* then a $\langle done \rangle$ message is returned. Process p also sends $\langle done \rangle$ messages to all processes in $Conf_p$ when it transits to *completed*. Upon receiving this message or a $\langle delete \rangle$ message from process p, process q removes p from N_q if q is at *trying*.

Snapshot computation of process p at *trying*
wait until a $\langle req \rangle$ is received from each process in N_p;

Events handlers
On transiting to *trying* $N_p := Conf_p$; $\langle \forall q \in N_p :$ **send** $\langle req \rangle$ to $q \rangle$;
On *receiving message* $\langle req \rangle$ *from* q **if** $state_p = trying \;\wedge\; q \notin N_p$ **then** $N_p := N_p \cup \{q\}$ **send** $\langle req \rangle$ to q **elsif** $(state_p = idle \;\vee\; state_p = completed)$ **then** **send** $\langle done \rangle$ to q **endif**
On transiting to *completed* $temp_p := Conf_p$; $\langle \forall q \in temp_p :$ **send** $\langle done \rangle$ to $q \rangle$
On *receiving message* $\langle done \rangle$ *or* $\langle delete \rangle$ *from* q **if** $state_p = trying \;\wedge\; q \in N_p$ **then** $N_p := N_p - \{q\}$

Figure 2: Snapshot Algorithm $S1$

3.1.1 Proof of Correctness

Theorem 5 *Property non-triviality is satisfied.*

Proof: Assume that jobs P and Q are at *executing* and that $q \in Conf_p \wedge p \in Conf_q$ holds. There are two cases to consider: either P received an $\langle add \rangle$ message from process q after transiting to *trying* or $q \in Conf_p$ has held continuously since P transited to *trying*. In the former case, P received an $\langle add \rangle$ message from process q while at *trying*. By Theorem 1, this message was sent by job Q. Thus, $p \in Conf_q$ and $p \in N_q$ when Q transits to *trying*. Job Q does not receive a $\langle done \rangle$ or a $\langle delete \rangle$ message from process p while at *trying* since such a message would have been sent by a predecessor of job P and in that case P would not have received the $\langle req \rangle$ message from Q. Since Q is currently at *executing*, $p \in N_q$ and Q must have received a $\langle req \rangle$ message from process p while at *trying*. This message could not have been sent by a predecessor of P since in that case P would have not received the $\langle req \rangle$ message from Q. Therefore, P sent a $\langle req \rangle$ message to Q in response to a $\langle req \rangle$ message. From algorithm $S1$, P must have received the $\langle req \rangle$ message from Q while at *trying* and added q to N_p. This completes the proof of the first case.

In the latter case, $q \in Conf_p$ and $q \in N_p$ when P transited to *trying*. Since $q \in Conf_p$, job P does not receive a $\langle delete \rangle$ message from process q while at *trying* or at *executing*. Job P also does not receive a $\langle done \rangle$ message from process q while at *trying* since this message would have been sent by a predecessor of Q and in that case P would have also received a $\langle delete \rangle$ message from process q while at *trying* or at *executing*. Therefore, $q \in N_p$ also holds when P transits to executing. We can repeat the reasoning of the first case on account of symmetry to meet the proof obligation if $p \in Conf_q$ as a result of receiving an $\langle add \rangle$ from process p. So, assume that $p \in Conf_q$ has held continuously since Q transited to *trying*. Then $p \in N_q$ when Q transits to *trying*. By repeating the reasoning of the second case, $p \in N_q$ also holds when Q transits to executing. Thus, $p \in N_q \;\wedge\; q \in N_p$ holds while both P and Q are at *executing*. This completes the proof. □

Theorem 6 *The property of validity is satisfied.*

Proof: Consider a global state at which P is at *executing* and $q \in N_p$. Let e_p be the global state prior to the transition of P to *executing*. From algorithm $S1$, P received a $\langle req \rangle$ message from process q while at *trying* at some global state r_p. Let Q be the job that sent the $\langle req \rangle$ message to P. Since Q sends $\langle req \rangle$ messages after sending $\langle add \rangle$ messages and before sending $\langle delete \rangle$ messages, $q \in Conf_p$ at global state r_p. From algorithm $S1$, Q sent this message prior to some global state r'_p at which Q was at *trying*. Also, $p \in Conf_q \ \wedge \ p \in N_q$ holds at r'_p. Job Q does not receive a $\langle delete \rangle$ or a $\langle done \rangle$ message from process p after state r'_p since such a message would have been sent by a predecessor of P and in that case P would not have received a $\langle req \rangle$ message from Q. Therefore, $p \in Conf_q \ \wedge \ p \in N_q$ holds from r'_p until Q transits to *executing*. If Q is currently at *trying*, then the proof obligation follows by considering the state r_p. Otherwise, let e_q be the global state prior to the transition of Q to *executing*. From algorithm $S1$, Q received a $\langle req \rangle$ message from p at some global state r_q prior to e_q while at *trying*. This message must have been sent by P since P would not have received a $\langle req \rangle$ message from Q otherwise. Let r'_q be the global state following the transmission of this message. By repeating the reasoning above, it can be shown that $q \in Conf_p \ \wedge \ q \in N_p$ holds from state r'_q until state e_p.

If P sends the $\langle req \rangle$ message to Q in response to the $\langle req \rangle$ message by Q then the desired global state of the theorem is given by r'_q. Similarly, if Q sends the $\langle req \rangle$ message to P in response to a $\langle req \rangle$ message by P then the desired global state of the theorem is given by r'_p. Finally, consider the remaining case in which P and Q send a $\langle req \rangle$ message to each other while transiting to *trying*. The desired global state in this case is given by considering the latter of the two states r'_p and r'_q. $\square$

The following lemma is proved in the appendix.

Lemma 1 *For any job P at trying and any process $q \in N_p$, if q is included in N_p when P transited to trying and q is not at executing when it receives P's $\langle req \rangle$ message then P receives a $\langle req \rangle$ message, a $\langle done \rangle$ message, or a $\langle delete \rangle$ message from q within $O(1)$ unit of time.*

In order to show that algorithm $S1$ satisfies conditional local progress, we prove the following theorem.

Theorem 7 *If each job at executing transits to completed within T units of time then a job at trying transits to executing within $O(1) + T$ units of time.*

Proof: From algorithm $S1$, a job P waits at *trying* until a $\langle req \rangle$ or a $\langle done \rangle$ message is received from each process q in N_p. By Lemma 1, P receives a $\langle req \rangle$ message, a $\langle done \rangle$ message, or a $\langle delete \rangle$ message within $O(1)$ unit of time unless the neighbor is at *executing*. By the assumption in the theorem, such a neighbor at *executing* transits to *completed* and sends a $\langle done \rangle$ message within T units of time. The theorem follows. $\square$

Algorithm $S1$ can be shown to satisfy the *global progress* condition. However, it does not satisfy the *no blocking* condition. The counterexample can be constructed by considering three *idle* jobs P,Q, and R. Suppose that P and R transit to *trying* with $Conf_p = N_p = \{r\}$ and $Conf_r = N_r = \{\}$. Job R transits to *executing* with $N_r = \{p\}$. Then, Q transits to *trying* with $Conf_q = N_q = \{p, r\}$ and sends $\langle req \rangle$ messages to p and r. Job P responds to this message by sending a $\langle req \rangle$ message and transits to *executing* with $Conf_p = N_p = \{q, r\}$. However, job R does not respond to the $\langle req \rangle$ message from Q. Thus, we have P and R at *executing* with $q \in N_p \cap Conf_p$ holds continuously while P is at *executing*, and Q at *trying* unable to transit to *executing* unless R completes the user-specified algorithm *proc* at *executing*. If the computation of *proc* does not terminate then q cannot transit to *executing* and the no blocking condition is violated. Worse yet, if the completion of *proc* at R requires the participation of P which in turn requires Q to begin executing *proc* (as in the case of node coloring) then a deadlock may be reached.

3.2 Snapshot Algorithm $S2$

Figure 3 describes the second snapshot algorithm that satisfies all three progress conditions. Set N_p is initialized to empty and variable $collecting_p$ is initialized to *false*. For each process q in $Conf_p$, a local variable $blocking_{pq}$ is also introduced. If $q \in Conf_p$ when p transits to *trying* then $blocking_{pq}$ is initialized to *true*. Otherwise, if q is included in $Conf_p$ as a result of receiving an $\langle add \rangle$ message after process p transits to *trying*, $blocking_{pq}$ is initialized to *false*.

Process p waits at *trying* until $blocking_{pq}$ is *false* for each process q in $Conf_p$. Then it sets variable $collecting_p$ to *true*, sends a $\langle req \rangle$ message to each process in $Conf_p$, and copies the set $Conf_p$ in variable $temp_p$ in a single atomic step. This atomicity requirement is needed since $Conf_p$, which appears in the range of the quantification, may change continuously. By making the entire group of statements atomic, a process p waits for a neighbor q iff it observed $blocking_{pq}$ to be *false* and if it sent a $\langle req \rangle$ message to q. Afterwards process p waits until an acknowledgement in the form of a $\langle req \rangle$ or a $\langle no \rangle$ message is received from each process in $temp_p$. When all the processes in $temp_p$ have replied, p transits to *executing* and sets variable $collecting_p$ to *false*.

Variable $blocking_{pq}$ is set to the value of $flag$ when a $\langle m_{flag} \rangle$ message is received and is set to *true* when a $\langle req \rangle$ message is received. If process p finds $collecting_p$ to be *true* when it receives a $\langle req \rangle$ message from process q, q is inserted into N_p. Otherwise, a $\langle no \rangle$ message is returned to q. Processes keep track of the arrival of $\langle no \rangle$ messages. Messages $\langle m_{flag} \rangle$ are sent in two situations. First, when process q is added to set $Conf_p$ while process p is not at *idle*, variable $blocking_{pq}$ is set to *false* and a message $\langle m_{flag} \rangle$, with $flag := (collecting_p \vee state_p = executing)$, is sent to q. Second, when process p transits to *completed*, a message $\langle m_{false} \rangle$ is sent to every process in set $Conf_p$.

3.2.1 Correctness

We following two lemmas concerning the state at which a job P sets variable $collecting_p$ to *true* are proved in the appendix.

Lemma 2 *If there exists a global state at which job P and process q are at executing and $q \in Conf_p$ then $q \in Conf_p$ when P sets $collecting_p$ to true.*

Lemma 3 *If there exists a global state at which jobs P and Q are at executing and $p \in Conf_q$ then $q \in Conf_p$ when P sets $collecting_p$ to true.*

It follows from the above two lemmas that if jobs P and Q are at *executing* simultaneously and if $q \in Conf_p$ then the jobs sent $\langle req \rangle$ messages to each other when they set *collecting* to *true*. The following lemma, also proved in the appendix, considers the arrival of these $\langle req \rangle$ messages.

Lemma 4 *If there exists a global state at which jobs P and Q are at executing and $q \in Conf_p$ then the $\langle req \rangle$ message sent from q to p (when Q set $collecting_q$ to true) arrives at p while $collecting_p$ is true and P is at trying.*

Theorem 8 *Property non-triviality is satisfied.*

Snapshot computation of process p at $trying$

$\ll$**wait until** $\langle \forall q \in Conf_p : \neg blocking_{pq} \rangle$;
    $collecting_p := true$;
    $\langle \forall q \in Conf_p :$ **send** $\langle req \rangle$ to $q \rangle$;
    $temp_p := Conf_p \gg$;
wait until a $\langle req \rangle$ or a $\langle no \rangle$ message is received from each process in $temp_p$;

Events handlers
On *receiving message* $\langle m_{flag} \rangle$ *from* q $blocking_{pq} := flag$

On *receiving message* $\langle req \rangle$ *from* q
    $blocking_{pq} := true$;
    if $collecting_p$ **then**
        $N_p := N_p \cup \{q\}$
    else send $\langle no \rangle$ to q **endif**

On *receiving message* $\langle no \rangle$ *from* q record the arrival of the message

On *adding* q *to* $Conf_p$
    if $state_p \neq idle$ **then** $blocking_{pq} := false$;
    $flag := (collecting_p \vee state_p = executing)$;
    send $\langle m_{flag} \rangle$ to q

On *deleting* q *from* $Conf_p$ **if** $state_p = trying$ **then** $N_p := N_p - \{q\}$
On transiting to $executing$ $collecting_p := false$

On transiting to *completed*
    $temp_p := Conf_p$;
    $\langle \forall q \in temp_p :$ **send** $\langle m_{false} \rangle$ to $q \rangle$

Figure 3: Snapshot Algorithm $S2$

Proof: Assume that jobs P and Q are at *executing* and that $q \in Conf_p \wedge p \in Conf_q$ holds. By Lemma 3, Q sends a $\langle req \rangle$ message to process p and by Lemma 4, this message is received at process p while $collecting_p$ is *true* and when P is at *trying*. Therefore, q is added to N_p when P is at *trying*. Job P does not receive a $\langle delete \rangle$ message from q before it transits to *executing* since P would not have received the $\langle req \rangle$ from Q otherwise. Consequently, $q \in N_p$ continues to hold while P is at *executing*. Similarly, we also have $p \in N_q$ while Q is at *executing*. The proof follows. □

The following three lemmas are needed for the proof of *validity* and the progress conditions. Their proofs appear in the appendix.

Lemma 5 *For any job P at executing, local snapshot N_p is a subset of $temp_p$, the copy of $Conf_p$ when P set $collecting_p$ to true.*

We say that a message m from q is an *acknowledgement* of a $\langle req \rangle$ message sent by a job P to q if between the states P sent the message and received m, $collecting_p$ is continuously $true$. We also say that a $\langle req \rangle$ message *triggers* a $\langle no \rangle$ message if the $\langle no \rangle$ message is sent in response to the $\langle req \rangle$ message.

Lemma 6 *For any two jobs P and Q, if a $\langle req \rangle$ message from P is received by Q then the first acknowledgement to this message, if any, is either a $\langle req \rangle$ message sent by Q or a $\langle no \rangle$*

message triggered by P's $\langle req \rangle$ message.

Lemma 7 *For any $\langle req \rangle$ message, there is exactly one acknowledgement.*

Theorem 9 *The property of validity is satisfied.*

Proof: Assume a state at which P is at *executing* and $q \in N_p$. From algorithm $S2$, P received a $\langle req \rangle$ message from a job Q of process q when P is at *trying* and $collecting_p$ is $true$. By Lemma 7, exactly one acknowledgement for the $\langle req \rangle$ message is received by Q. Since the $\langle req \rangle$ message is received when $collecting_p$ is $true$ it does not trigger a $\langle no \rangle$ message. Therefore, by Lemma 6 the acknowledgement of the $\langle req \rangle$ message must be a $\langle req \rangle$ message sent by P when it set $collecting_p$ to $true$. Also, we have $q \in Conf_p$ when P sends its $\langle req \rangle$ message. Job P does not receive any $\langle delete \rangle$ messages from q between the state in which it sends its $\langle req \rangle$ message and the state in which it receives Q's $\langle req \rangle$ message. (Otherwise, it would not have received Q's $\langle req \rangle$ message.) Thus, $q \in Conf_p$ holds continuously between these two states. Similarly, $p \in Conf_q$ is true when Q sends its $\langle req \rangle$ message and $p \in Conf_q$ holds continuously until Q receives P's $\langle req \rangle$ message. Since the two $\langle req \rangle$ messages are sent concurrently, *validity* follows. □

To show that the global progress and the no blocking conditions are satisfied, we prove the following theorem.

Theorem 10 *From any global state at which job P is at trying, some process in the set $Conf_p \cup \{p\}$ is at executing within $O(1)$ unit of time. Also, for any job P at executing and any job Q at trying such that $q \in N_p$, if $q \in Conf_p$ has held continuously since P transited to executing, then within $O(1)$ unit of time, either P transits to completed or Q transits to executing.*

Proof: Assume that P is at $trying$ and that no process in the set $Conf_p$ has set *collecting* to *true* or has reached *executing*. On account of the delivery of the $\langle m_{flag} \rangle$ messages and the setting of the variables *blocking*, within 2ν time units P or one of the processes in $Conf_p$ will find the first waiting condition at *trying* to be *true* and proceed to set *collecting* to *true*. From Figure 3, a job P with variable $collecting_p$ set to *true* sends a $\langle req \rangle$ message to each process $q \in temp_p$. By Lemma 7, an acknowledgement is received for each such $\langle req \rangle$ message. By Lemma 6, these acknowledgements are received by P within 2ν units of time so that P transits to *executing* within 2ν units of time. The first statement of the theorem follows.

For the proof of the second statement, assume a state s at which P is at *executing*, Q is at *trying*, $q \in N_p$, and assume that $q \in Conf_p$ held continuously since P transited to *executing*. Since $q \in N_p$, P received a $\langle req \rangle$ message from Q or a predecessor Q' of Q. In the former case, by repeating the reasoning about global progress, Q will receive an acknowledgement for each $\langle req \rangle$ message it sent within 2ν units of time and transits to *executing* within 2ν units of time. Thus the no blocking condition holds in this case. To completes the proof of the theorem, we show that the latter case is not possible. According to algorithm $S2$, $p \in Conf_q$ is true when Q' sends its $\langle req \rangle$ message. If Q' receives a $\langle delete \rangle$ message from a predecessor P' of P then since P' waits for acknowledgements for $\langle delete \rangle$ messages, P does not receive the $\langle req \rangle$ message from Q'. Also Q' does not receive a $\langle delete \rangle$ message from P. Therefore Q' does not receive a $\langle delete \rangle$ message from p and according to the user outline in Figure 1, Q' sends a $\langle delete \rangle$ message to p at *completed*. Since Q' waits for acknowledgements for its $\langle delete \rangle$ messages, P must have received the $\langle delete \rangle$ message from Q' before state s. If this $\langle delete \rangle$ message is received before P transits to *executing* then the assumption that $q \in N_p$ as a result of receiving a $\langle req \rangle$ message from Q' is violated. Otherwise, if the $\langle delete \rangle$ message is received after P transits to *executing* then q is removed from $Conf_p$ and the condition $q \in Conf_p$ does not hold continuously while P is at *executing*. The proof follows. □

To show that the conditional local progress condition is satisfied, we prove the following theorem.

Theorem 11 *If each job at executing transits to completed within T units of time then a job at trying transits to executing within $O(\delta * T)$ units of time.*

Proof: A job P that has passed the first waiting statement sends $\langle req \rangle$ messages and obtains acknowledgements within $O(1)$ time units. After that, such a job transits to *executing* within $O(1)$ time units. Therefore, the time complexity of the *trying* to *executing* transition is determined by the time that a job takes to complete its first waiting statement. We next show that this time is $O(\delta * T)$ time units.

Consider a job P that transits to *trying*. It can be blocked at the first waiting statement by a neighbor q if $blocking_{pq}$ is true. Now, $blocking_{pq}$ is set to true only if q has passed its own first waiting statement and has not reached *completed*. Since, such a process transits to *completed* within $O(1) + T$ time units and its subsequent jobs do not set $blocking_{pq}$ to true, $blocking_{pq}$ can be true for a maximum of $O(1) + T$ units of time. The theorem follows by summing this quantity over all neighbors of P. □

4 Solutions to Distributed Job Scheduling

In this section we present two solutions to the distributed job scheduling problem based on the snapshot algorithms described in the previous section. Our first algorithm, $J1$, is obtained by combining a dining philosophers algorithm with snapshot algorithm $S1$.

Given an underlying dining philosophers algorithm with response time[1] $T(\nu, \tau)$ and message complexity M, the resulting algorithm has a response time of $O(1) + 2 * T(\nu, \tau)$ and a message complexity of $O(\delta) + M$. Though any dining philosophers solution can be used in algorithm $J1$, the setting of the initial state requires special care. For example, consider the dining philosophers solution due to Chandy and Misra [6]. As a part of the initial state, this solution requires an acyclic graph connecting all the processes. The process IDs can be used to obtain this initial acyclic graph by directing the edge between two processes towards the one with the higher ID. If the dining philosophers solution is based on node colors then the process IDs need to be used as the node colors. In other words, the setting of the initial state needs to be performed by using the process IDs and no prior initialization viz., node colors or logical structures such as trees, can be assumed. In case an initial state that is not based just on process IDs is desired, it needs to be computed explicitly. Such an explicit initialization in the form of a node coloring is considered by our second algorithm $J2$. It is obtained by combining a $(\delta + 1)$-*coloring* scheme with a dining philosophers solution based on node coloring. The $(\delta + 1)$-*coloring* scheme is used to color the nodes of the conflict graph in the first phase and once the nodes have been colored, the dining philosophers algorithm is used to resolve conflicts in the second phase. Snapshots of the conflict graph are taken during each phase. The snapshot computation during the first phase requires global progress and no blocking conditions, and is thus implemented using snapshot algorithm $S1$. The snapshot computation during the second phase requires only conditional local progress and is thus implemented using snapshot algorithm $S2$. If the response time of a $(\delta + 1)$-*coloring* algorithm (which is the time it takes for any node to be colored) is $T_\delta(\nu)$ and the response time of the dining philosophers algorithm

[1] The local computation time is ignored in the calculation of response time.

is $T_{dining}(\nu, \tau)$ then the resulting response time for the distributed job scheduling algorithm is given by $O(\delta) * T_\delta(\nu) + O(1) * T_{dining}(\nu, \tau)$. Similarly, if the message complexity of the $(\delta+1)$-*coloring* algorithm and the dining philosophers algorithm are M_δ and M_{dining} respectively then the message complexity of the resulting algorithm is given by $O(\delta) + M_\delta + M_{dining}$. In particular, the algorithm obtained by combining the $(\delta+1)$-*coloring* algorithm of Goldberg, Plotkin, and Shannon [9] with a dining philosophers algorithm in [7] has a response time of $O((\delta^3 + \delta \log^* N)\nu + O(\delta^2)\tau)$ and a message complexity of $O(\delta^2 + \delta \log^* N)$.

4.1 Algorithm $J1$

Figure 4 shows the sequence of steps executed by a process p at *executing* and *completed*. The execution of a process at *idle* and *trying* is as defined by the user outline in Figure 1. On transiting to *executing*, a process also transits to a *hungry* state with respect to the dining philosophers algorithm. Its transition to the *eating* state is defined by the dining philosophers algorithm. Once it completes eating, it transits to the *thinking* state and also to *completed*. On account of *non-triviality*, two conflicting *hungry* processes are in each other's local snapshot and therefore, the arbitration between conflicting processes is carried out correctly. A process that is not at *executing* is in the *thinking* state and does not contend for the resources. By the correctness of the underlying dining philosophers solution, a hungry process eventually completes eating and therefore, every process at *executing* eventually transits to *completed*. A *hungry* process p that finds a process $q \in N_p$ at *trying* does not need to wait for q to transit to *executing* since process q follows the protocol of a *thinking* process. Therefore, *conditional local progress* is the only progress condition that is needed and algorithm $S1$ can be used. Upon transiting to state *completed*, process p copies $Conf_p$ in a local variable $temp_p$ and waits until each process in $temp_p$ has been observed to be at *idle* or *completed*. This ensures that any active job Q that conflicts with an uncompleted job P remains active until P completes.

executing

execute *dining philosophers algorithm*
with N_p as the neighborhood;

↓

completed

$temp_p := Conf_p$;
wait until $\langle \forall q \in temp_p :$
$\quad state_q = idle \lor state_q = completed \rangle$;
....
....

Figure 4: Algorithm $J1$ for distributed job scheduling

4.1.1 Correctness

The following theorem establishes that conflicting jobs are not executed together in algorithm $J1$ provided the underlying dining philosophers algorithm satisfies mutual exclusion.

Theorem 12 *For any two jobs P and Q at executing, if $q \in Conf_p$ holds then $q \in N_p$ and $p \in N_q$.*

Proof: By Theorem 3, $q \in Conf_p \wedge p \in Conf_q$ holds. The desired proof obligation follows from *non-triviality.* □

Theorem 13 *If the underlying dining philosophers algorithm has a response time of* T_{dining} *and a message complexity of* M_{dining} *then the response time and the message complexity of algorithm* $J1$ *are* $2 * T_{dining}$ *and* $O(\delta) + M_{dining}$ *respectively.*

Proof: The response time of the algorithm follows by applying Theorem 7. In measuring the message complexity, we ignore the number of messages needed to obtain the set $Conf$ from the resource managers since this incurs a lower bound of at least one message per resource required. The message complexity is obtained by adding the following: $O(\delta)$ messages for each of $\langle add \rangle$, $\langle m_{flag} \rangle$, $\langle req \rangle$, $\langle no \rangle$, and $\langle delete \rangle$, and M_{dining} messages for the dining philosophers algorithm. □

4.2 Algorithm $J2$

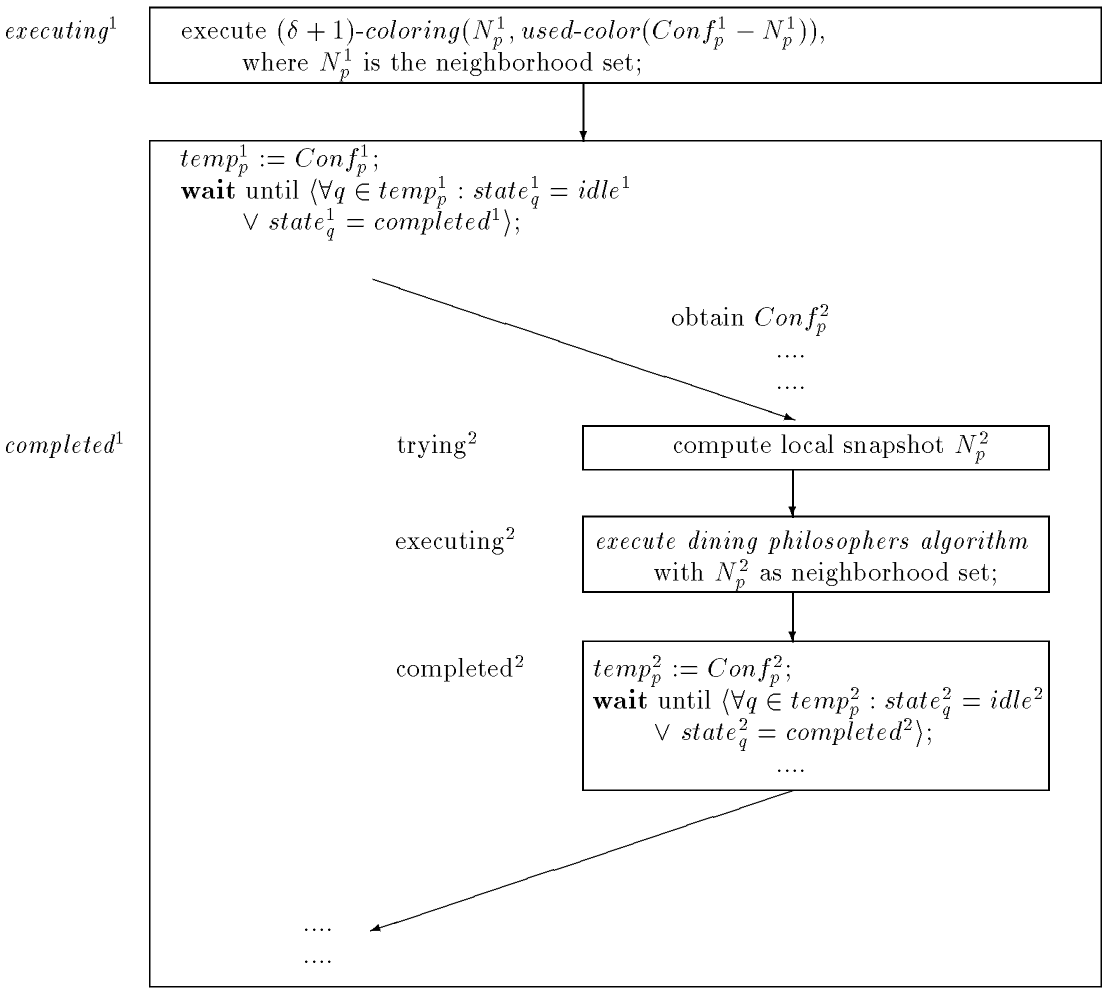

Figure 5: Algorithm $J2$ for distributed job scheduling

As shown in Figure 5, algorithm $J2$ combines a $(\delta+1)$-*coloring* scheme with a given dining philosophers solution based on node coloring. Each job takes two snapshots, one for the

$(\delta+1)$-*coloring* and another for the dining philosophers solution. Though the execution of the underlying dining philosophers solution could also be based on the first snapshot, using two separate snapshots improves the efficiency of the combined solution. In order to distinguish the states and the variables of the two snapshot algorithms, superscripts of 1 and 2 are used for the first and the second snapshots respectively. The second snapshot algorithm is nested within the first snapshot algorithm and $Conf_p^2 \subseteq Conf_p^1$. Initially a process is at $idle^1$ and $idle^2$ and the states $trying^2$, $executing^2$, and $completed^2$ are all nested within $completed^1$. At state $executing^1$, a $(\delta+1)$-*coloring* algorithm is executed on the static graph defined by the local snapshots N_p^1. The $(\delta+1)$-*coloring* algorithm is invoked with a parameter *used-colors*$(Conf_p^1 - N_p^1)$ which returns the set of colors currently used by the set of processes in $Conf_p^1 - N_p^1$. When the coloring algorithm completes, a process p is colored with a color different from the colors of the processes in N_p^1 and from colors in set *used-colors*$(Conf_p^1 - N_p^1)$. Then the process transits to $completed^1$. At this state, set $Conf_p^1$ is copied to a local variable $temp_p^1$ and process p waits until it observes each process in $temp_p^1$ to be at $idle^1$ or at $completed^1$. After that the process sets $Conf_p^2$ to the subset of processes in $temp_p^1$ that are observed to be at $completed^1$ and the set of processes from which it has received $\langle add^2 \rangle$ messages but no corresponding $\langle delete^2 \rangle$ messages. Then it sends out appropriate $\langle add^2 \rangle$ messages to processes in $Conf_p^2$ and transits to $trying^2$. Once a local snapshot N_p^2 is obtained, the process executes the dining philosophers algorithm with N_p^2 as the neighborhood set as in algorithm $J1$. Then the process copies $Conf_p^2$ to a local variable $temp_p^2$ and waits until it observes each process in $temp_p^2$ to be at *idle* or *completed*. After that the process transits to $idle^2$ and continues with the remainder of the code at $completed^1$.

As in algorithm $J1$, only *conditional local progress* condition is required for the execution of the dining philosophers algorithm at $executing^2$ and consequently, algorithm $S1$ can be used at $trying^2$. However, the computation of colors requires the participation of neighboring processes implying the need for the no blocking and global progress conditions. Consequently, snapshot algorithm $S2$ is used at $trying^1$. A brief account of the $(\delta+1)$-*coloring* is given below. The dining philosophers algorithm used here is algorithm 1 of [7] and its description is omitted.

4.2.1 $(\delta+1)$-*coloring*

The $(\delta+1)$-*coloring* algorithm used here is based on the algorithm by Goldberg, Plotkin, and Shannon [9]. However, our algorithm (shown in Figure 6) is implemented in a distributed message passing environment instead of a shared memory environment and the validity of a coloring is slightly different. A node invoking our coloring algorithm has two parameters R and *used* such that $|R| + |used| \leq \delta$. Parameter R is the set of neighbors of the node and *used* is a set of colors that the node should not be colored with. A coloring is valid if neighboring nodes are not colored with the same color and a node is not colored with any color in set *used*.

The $(\delta+1)$-*coloring* algorithm is divided into four phases. In the first phase, the static graph $G = (V, E)$ is partitioned into at most δ *pseudo forests* $G_k = (V, E_k)$, $k \in [1..\delta]$, which are directed graphs in which the out-degree of any node is at most one. In our implementation in Figure 6, process p assigns a distinct number $k_{pq} \in [1..\delta]$ to each adjacent edge in the graph. An edge e between p and q belongs to pseudo forest G_k if k is the smaller of the numbers assigned by p and q. The direction of the edge points to the process that assigns a larger number to the edge or to the process with a smaller ID in case of a tie. Since an edge only points outward from a node in pseudo forest G_k if the node assigns k to the edge, the out-degree of each node is at most one.

```
Procedure (δ+1)-coloring(R,used)
    /* phase 1 */
    assign a distinct number k_pq ∈ [1..|R|] for each q ∈ R;
    send k_pq to q;
    wait until k_qp is received from each q ∈ R;
    for each q ∈ R do
        k := min(k_pq, k_qp);
        if k_pq < k_qp ∨ (k_pq = k_qp ∧ id_p < id_q) then
            insert (p,q) into E_k
        else
            insert (q,p) into E_k
        endif
    endfor;
    /* phase 2 */
    obtain 6-colorings for each pseudo forest G_j, j ∈ [1..|R|];
    /* phase 3 */
    obtain 3-colorings from the 6-colorings;
    /* phase 4 */
    reduce the 3-colorings to a valid (δ+1)-coloring that
        does not use colors in set used;
```

Figure 6: Procedure $(\delta+1)$-coloring

In the second phase of the algorithm, the 6-*coloring* algorithm described in [9] is executed on each pseudo forest G_k. Each node p in a pseudo forest is colored by its ID in the beginning and is repeatedly recolored with a color equal to the concatenation of ith bit of its previous color and the binary representation of i, where i is the least significant bit of p's color that is different from the corresponding bit of the color of p's parent. (For a root node, i is taken as zero.) It is shown in [9] that a 6-*coloring* is obtained for each pseudo forest after the recoloring procedure is repeated for $O(\log^* N)$ steps where N is the maximum value of the IDs.

In the third phase of the algorithm, each 6-*coloring* obtained in phase 2 is reduced into a 3-*coloring* in 3 successive stages, one for each color $\in 5, 4, 3$. At each stage, colors are shifted down from parent nodes to child nodes, while a root node is recolored with a color different from its current color. Then a color c is removed by recoloring every node of color c with the smallest color different from c, its parent's color, and the color of its children.

Finally, in the fourth phase, the 3-*colorings* obtained for each pseudo forest G_k are combined into a valid $(\delta+1)$-*coloring*. Let the color of a node p in the 3-*coloring* of pseudo forest G_k be c_{pk}. Define graphs $g_k = (V, \cup_{i=1}^{i=k} E_i)$. A node p's color c_p is initialized to 0 which is a valid coloring for graph g_0. The algorithm then proceeds in δ iterations. After the kth iteration, c_p is reassigned so that it becomes valid on g_k. In the kth iteration, a node p with a smaller value for (c_p, c_{pk}) than its neighbors chooses a color different from each of its neighbors, assigns it to c_p, and sends its new color to each of its neighbors. Then the node with the next smallest pair (c_p, c_{pk}) obtains a new color in the same fashion. The kth iteration ends when every node is recolored during the iteration. Colors in set *used* are not chosen during this recoloring procedure.

Theorem 14 *The $(\delta+1)$-coloring algorithm in Figure 6 has a time complexity of $O(\delta^2 + \log^* N)$ and a message complexity of $O(\delta^2 + \delta \log^* N)$.*

4.3 Correctness

We first show that the underlying static $(\delta+1)$-*coloring* algorithm when combined with any snapshot protocol that satisfies the properties of *non-triviality*, *validity*, *global progress*, and *no blocking* conditions properly colors the dynamic conflict graph.

Lemma 8 *For any two jobs P and Q at executing^2, $q \in Conf_p^1 \Rightarrow color_p \neq color_q$.*

Proof: Consider the intervals where P and Q are at $executing^1$. If they overlap then by Theorem 4, $q \in Conf_p^1 \ \wedge \ p \in Conf_q^1$ holds while both P and Q are at $executing^1$. Consequently, by *non-triviality*, $q \in N_p^1 \ \wedge \ p \in N_q^1$ holds and P and Q choose different colors during $(\delta+1)$-*coloring*. If the intervals do not overlap then by Theorem 4, the job that transits to $executing^1$ later finds the other process in its $Conf^1$ set, obtains the color used by the other process, and chooses a different color. □

Lemma 9 *For any job P with a maximum of δ_p conflicting neighbors, $color_p \in [0..\delta_p]$.*

Proof: By the correctness of the $(\delta+1)$-*coloring* algorithm, it is sufficient to show that for a job P at $executing^1$, N_p^1 is a subset of $Conf_p^1$. Let q be any process in N_p^1. By *validity*, there exists a previous global state s at which q is at $trying^1$, $q \in Conf_p^1$, and $p \in Conf_q^1$. Let Q be job of process q corresponding to that particular iteration of process q. By Theorem 2, $q \in Conf_p^1 \ \wedge \ p \in Conf_q^1$ holds until either P or Q send $\langle delete^1 \rangle$ messages. From algorithm $J2$, P and Q wait at *completed*^{1} and do not send $\langle delete^1 \rangle$ messages until the other job transits to *completed*^{1}. The proof follows. □

For the correct execution of the dining philosophers algorithm, we first prove the following lemma that shows that all processes in the neighborhood of a process p executing the dining philosophers algorithm are colored and contained in the conflict set $Conf_p^1$.

Lemma 10 *For any job P at executing^2, if $q \in N_p^2$ then $q \in Conf_p^2$ and q is at trying^2, executing^2, or completed^2.*

Proof: Assume that P is at $executing^2$ and $q \in N_p^2$. By *validity*, there is a previous global state s at which job P and process q are at $trying^2$ and $p \in Conf_q^2 \ \wedge \ q \in Conf_p^2$ holds. Let Q be the job corresponding to that particular iteration of process q. By Theorem 2, $q \in Conf_p^2 \ \wedge \ p \in Conf_q^2$ holds until either P or Q send $\langle delete^2 \rangle$ messages. From algorithm $J2$, P and Q wait at *completed*^{2} and do not send $\langle delete^2 \rangle$ messages until the other job transits to *completed*^{2}. This completes the proof. □

The following theorem establishes that conflicting jobs are not executed together in algorithm $J2$ provided the underlying dining philosophers algorithm satisfies mutual exclusion.

Theorem 15 *For any two jobs P and Q at executing^2, if $q \in Conf_p^1$ holds then $q \in N_p^2$ and $p \in N_q^2$.*

Proof: We show that $q \in Conf_p^2 \ \wedge \ p \in Conf_q^2$ holds. The desired proof obligation then follows from *non-triviality*. By Theorem 4, $p \in Conf_q^1 \ \wedge \ q \in Conf_p^1$ holds after both p and Q transit to $executing^1$. Consider the last of the two jobs that transits to *completed*^{1}. Let it be R and let R' be the other job. From algorithm $J2$, R will find $r' \in Conf_r^1$ and will observe the state of r' to be *completed*^{1} and add r' to $Conf_r^2$. If R receives a $\langle delete^2 \rangle$ message from process r' then it also receive a $\langle delete^1 \rangle$ message from process r', contradicting the stability of $p \in Conf_q^1 \ \wedge \ q \in Conf_p^1$. Therefore, $r' \in Conf_r^2$ holds when both R and R' are at $executing^2$.

By Theorem 3, $p \in Conf_q^2 \ \wedge \ q \in Conf_p^2$ holds when both P and Q are at $executing^2$. The proof follows. □

In order to calculate the response time of algorithm $J2$, we first prove the following lemma. We say that a job Q *overtakes* another job P if at some global state $p \in Conf_q^1$ and P, Q are at state $trying^1$, and at a later global state P is still at state $trying^1$ and Q is at $executing^1$.

Lemma 11 *At most δ_p jobs overtake job P.*

Proof: Let Q be a job that overtakes P. On account of Theorem 2, $p \in Conf_q^1$ holds until either P or Q send $\langle delete^1 \rangle$ messages. By algorithm $J2$, Q waits at *completed*^{1} and does not send any $\langle delete^1 \rangle$ messages to p until P transits to *completed*^{1}. On account of Theorem 3, $q \in Conf_p^1$ at the state when Q transits to $executing^1$. Again by Theorem 2, $q \in Conf_p^1$ holds until either P or Q send $\langle delete^1 \rangle$ messages. Consequently, $q \in Conf_p^1$ until P transits to *completed*^{1}. Each such process q that overtakes job P is a member of the set $Conf_p$ and remains in the set until P transits to $executing^1$. Since $|Conf_p^1| \leq \delta_p$, the lemma follows. □

The response time of algorithm $J2$ is calculated by considering the time taken for each state transition. Let $T_{t,e}^i$ and $T_{e,c}^i$ denote the maximum time a process takes to transit from *trying*^{i} to *executing*^{i} and from *executing*^{i} to *completed*^{i}, respectively, for $i = 1$ or 2. Note that $T_{e,c}^1 = O(1) + T_\delta$ on account of Theorem 10.

Lemma 12 $T_{t,e}^1 = O(\delta * T_{e,c}^1) = O(\delta * T_\delta)$.

Proof: Let p be a process waiting at *trying*^{1}. By *global progress* of snapshot algorithm $S2$, process p or one of its neighbors will eventually be at $executing^1$. (For the snapshot algorithm $S2$ given in Section 2, the time taken for this transition is $O(1)$.) This process will enter *completed*^{1} within time $T_{e,c}^1$. From Lemma 11, process p has to wait for at most δ_p such processes to transit to *completed*^{1}. Consequently, within $O(\delta * T_{e,c}^1)$ time process p transits to $executing^1$. The lemma follows as $T_{e,c}^1 = O(1) + T_\delta$. □

Theorem 16 *The response time of algorithm $J2$ is $O(\delta * T_\delta + T_{dining})$ and the message complexity is $O(\delta) + M_\delta + M_{dining}$, where T_δ and M_δ are the time and message complexities of the $(\delta + 1)$-$coloring$ algorithm.*

Proof: From Theorem 7, $T_{t,c}^2 = O(1) + 2 * T_{dining}$. The response time of algorithm $J2$ follows by adding this to the quantities $T_{t,e}^1$ and $T_{e,c}^1$ computed earlier. The proof of the message complexity is similar to that of Theorem 13. □

Corollary 1 *The response time of algorithm $J2$ is $O(\delta^3 + \delta \log^* N)$ and the message complexity is $O(\delta^2 + \delta \log^* N)$ when the $(\delta + 1)$-$coloring$ algorithm in Figure 6 and the dining philosophers algorithm in [7] are used.*

Note that if only one snapshot is used and the dining philosophers algorithm is invoked after $(\delta + 1)$-$coloring$ at $executing^1$ then the total response time would be $\delta *($ response time of $(\delta + 1)$-$coloring$ + response time of dining philosophers). By using two snapshots, we multiply only the response time of $(\delta + 1)$-$coloring$ by a factor of δ.

5 Conclusion

The idea of taking snapshots on a dynamic graph provides us with a new method for solving dynamic graph problems based on an underlying static graph algorithm. In the case of the distributed job scheduling problem, we designed an algorithm by combining existing solutions to $(\delta + 1)$-*coloring* and dining philosophers with our snapshot algorithms. The result is an algorithm with a response time of $O(\delta^3+\delta \log^* N)$ and a message complexity of $O(\delta^2+\delta \log^* N)$. It should be noted that both the complexity measures depend only minimally on N.

Adopting a compositional approach in the design of our algorithms allowed us to reuse existing algorithms without sacrificing performance. Moreover, it allows us to ride the "technology curve" as improvements in the efficiency of the basic building blocks such as node coloring will also lead to improvements in the time and message complexities of our algorithms. The success of such a compositional approach depends critically on a clean specification of the components. We examined a number of different specifications of the snapshots and finally settled on what appeared to be the simplest.

The idea of snapshots so far seems to be applicable to only those problems where the computation is local with certain non-local constraints or conflicts. We are examining if the ideas of this paper can also be extended to other kinds of graph problems such as shortest paths and maximum independent sets. Existing techniques to solve these problems have relied on some form of a *reset* or *restart* after each change in the underlying network [1, 3]. Once the network stablizes for a sufficiently long period of time, meaningful computations can be carried out. We are investigating whether some *resets* or *restarts* can be avoided if more information is available about the kind of computation being performed.

References

[1] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, IEEE*, pages 358–370, 1987.

[2] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 65–74, 1990.

[3] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, IEEE*, pages 206–220, 1988.

[4] J. Bar-Ilan and D. Peleg. Distributed resource allocation algorithms. In *6th International Workshop on Distributed Algorithms*, pages 277–291, 1992.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transaction on Computer Systems*, 3(1):63–75, 85.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley Pub. Co., Reading, Mass., 1988.

[7] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 593–602, 1992.

[8] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communication of ACM.*, 8(9):569, 1965.

[9] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[10] N. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 70–81, 1980.

[11] B. Meyer. Systematic concurrent object-oriented programming. *Interactive Software Engineering Inc., Technical Report TR-EI-37/SC*, 1993.

[12] I. Page, T. Jacob, and E. Chern. Fast Algorithms for Distributed Resource Allocation. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), pages 188–197, 1993.

[13] E. Styer and G. Peterson. Improved algorithms for distributed resource allocation. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–116, 1988.

Appendix

Proof of Theorem 1: The $\langle add \rangle$ message was sent either by Q or a predecessor Q' of Q. In the latter case, Q' would have also sent a $\langle delete \rangle$ message after the $\langle add \rangle$ message and waited for an acknowledgement. Consequently, P would have also received the $\langle delete \rangle$ message and q would not be in $Conf_p$, as claimed in the statement of the theorem. □

Proof of Theorem 2: It suffices to show that P does not receive a $\langle delete \rangle$ message from a predecessor of Q after a state s at which $q \in Conf_p$. The proof is by contradiction. Suppose that P receives a $\langle delete \rangle$ message from a predecessor Q' of Q after state s. Since Q' waits for an acknowledgement of the $\langle delete \rangle$ message, Q cannot be active at state s. □

Proof of Theorem 3: Since $p \in Conf_q$, either $p \in Conf_q$ has held continuously since Q transited to *trying* or Q received an $\langle add \rangle$ message from p after transiting to *trying*. In the former case, process q sends an $\langle add \rangle$ message to process p. From the user outline in Figure 1, this message is received at process p before Q transits to *executing*. Now, q can be removed from $Conf_p$ either as a result of receiving a $\langle delete \rangle$ message from q or by the resetting of $Conf_p$ to $\{\}$ by a predecessor of P. The former is not possible due to FIFO channels and since Q has not sent out any $\langle delete \rangle$ messages. The latter is also not possible since the predecessor of P would have sent out a $\langle delete \rangle$ message to q and waited for an acknowledgement. The $\langle delete \rangle$ message would be received by job Q and consequently, p would be removed from $Conf_q$. Therefore, $q \in Conf_p$ until either P or Q send out $\langle delete \rangle$ messages.

In the latter case, by Theorem 1, P must have sent the $\langle add \rangle$ message when it transited to *trying*. This means that $q \in Conf_p$ at that global state. By the user outline in Figure 1, $q \in Conf_p$ will hold until either P or Q send out $\langle delete \rangle$ messages. The proof follows. □

Proof of Theorem 4: Since $q \in Conf_p$ holds, either $q \in Conf_p$ has held continuously since P transited to *trying* or P received an $\langle add \rangle$ message from process q after transiting to *trying*. In the former case, P sent an $\langle add \rangle$ message to q. By the user outline in Figure 1, this message is received at q before P transits to *executing*. Set $Conf_q$ could not have been reset after the reception of this message as in that case P would have received a $\langle delete \rangle$ message and removed q from $Conf_p$. Therefore, $p \in Conf_q$ holds and is stable after P transits to *executing*. Consequently, $p \in Conf_q$ holds and is stable after both P and Q transit to *executing*. The desired result follows from Theorem 3.

In the latter case, P received an $\langle add \rangle$ message from process q after transiting to *trying*. By Theorem 1, this message was sent by Q. By the user outline in Figure 1, this message is received by P before Q transits to *executing*. Therefore, $q \in Conf_p$ holds and is stable after Q transits to *executing*. Consequently, $q \in Conf_p$ holds and is stable after both P and Q transit to *executing*. The desired result again follows from Theorem 3. This completes the proof. □

Proof of Lemma 1: Let P be at *trying* and q be included in N_p when P transited to *trying*. Assume that q is not at *executing* when it receives P's $\langle req \rangle$ message. Consider the state of q when it receives the $\langle req \rangle$ message from P. If q is at *idle* or at *completed* then it returns a $\langle done \rangle$ message and we are done. If q is at *trying* and $p \notin N_q$ then it sends a $\langle req \rangle$ message to P and we are also done. It remains to consider the case in which q is at *trying* and $p \in N_q$. Let Q be the job of process q at this state. Since P sends an $\langle add \rangle$ message before sending a $\langle req \rangle$ message, the $\langle add \rangle$ message is received at q before the $\langle req \rangle$ message. If the $\langle add \rangle$ message is received at q before the predecessor of Q resets $Conf_q$ to empty then a $\langle delete \rangle$ message is returned to P which removes q from N_p. Otherwise, the $\langle add \rangle$ message is received at q after the predecessor of Q resets $Conf_q$ and $p \in Conf_q$ when the $\langle req \rangle$ message is received. Now, there are two cases to consider. Either $p \in Conf_q$ has held continuously since Q transited to *trying* or Q received an $\langle add \rangle$ message from P after transiting to *trying*.

In the latter case, Q received an $\langle add \rangle$ message from P after transiting to *trying*. Since $p \in N_q$ holds, Q must have already received a $\langle req \rangle$ message from P, leading to a contradiction. So, we concentrate on the former case. Here, Q sent an $\langle add \rangle$ message followed by a $\langle req \rangle$ message to process p when transiting to *trying*. If the $\langle add \rangle$ message is received by a predecessor of P before $Conf_p$ is reset to empty then a subsequent $\langle delete \rangle$ message is received at Q and consequently p cannot belong to N_q as assumed. Therefore, the $\langle add \rangle$ message is received at p after the predecessor of P resets $Conf_p$ to empty. If the subsequent $\langle req \rangle$ message sent by Q is received at p before P transits to *trying* then a $\langle done \rangle$ message is returned to Q and p cannot belong to N_q as assumed. Therefore, the $\langle req \rangle$ message is received by P after it transits to *trying*. Since this $\langle req \rangle$ message is sent by Q before Q receives P's $\langle req \rangle$ message, Q's message arrives at P within $O(1)$ unit of time after P sends its $\langle req \rangle$ message and the lemma follows. □

Proof of Lemma 2: Since $q \in Conf_p$, either $q \in Conf_p$ has held continuously since P transited to *trying* or P received an $\langle add \rangle$ message from process q after transiting to *trying*. The proof is obvious in the former case as P sets $collecting_p$ to *true* after transiting to *trying*. For the latter case, let Q be the job sending the $\langle add \rangle$ message. If q is added to $Conf_p$ after P sets $collecting_p$ to *true*, then according to algorithm $S2$, P sends a $\langle m_{true} \rangle$ message to process q. At process q, variable $blocking_{qp}$, which is initialized to *true*, remains *true* on receiving the $\langle m_{true} \rangle$ message. Consequently, Q cannot proceed to *executing* until P has reached *completed*, contradicting the assumption that job P and process q are at *executing* at the same global state. Therefore, q is added to $Conf_p$ before $collecting_p$ is set to *true*. □

Proof of Lemma 3: Consider the state at which p is added to $Conf_q$. If p is added to $Conf_q$ when Q transits to *trying* then by repeating the argument of the second case in Lemma 2, q belongs to $Conf_p$ when P sets $collecting_p$ to *true*. Otherwise, p is added to $Conf_q$ on account of the arrival of an $\langle add \rangle$ message from process p after Q has transited to *trying*. By Theorem 1, this $\langle add \rangle$ message must have been sent by job P and therefore, $q \in Conf_p$ when P transits to *trying*. Next, we show that P does not receive any $\langle delete \rangle$ messages from q before it sets $collecting_p$ to *true*. Suppose, on the contrary, that P receives a $\langle delete \rangle$ message from q after it obtains $Conf_p$ and sends an $\langle add \rangle$ message to q, and later reaches the global state of the lemma. By the FIFO assumption, the acknowledgement of the $\langle delete \rangle$ message arrives at q after the $\langle add \rangle$ message. According to the user outline in Figure 1, $Conf_q$ is reset to empty after all the acknowledgement of the $\langle delete \rangle$ message are received. Therefore $Conf_q$ is reset to empty after the receipt of the $\langle add \rangle$ message. This contradicts the assumption that $p \in Conf_q$ is *true* at the global state of the lemma. □

Proof of Lemma 4: For the sake of contradiction, suppose that p receives q's $\langle req \rangle$ message while $collecting_p$ is *false* or P is not at *trying*. There are two cases to consider: either the $\langle req \rangle$ reaches process p before P sets $collecting_p$ to *true* or after P resets $collecting_p$ to *false* and P has transited to *executing*. In the former case, variable $blocking_{pq}$ will be set to *true* and P will not complete the execution of the first waiting statement at *trying* until Q transits to *completed*. This violates the assumption that there is a later global state at which jobs P and Q are at *executing*. In the latter case, on account of Lemma 2, P sent a $\langle req \rangle$ message to process q when it set $collecting_p$ to *true*. Using the same argument as above, this message does not reach process q before Q sets $collecting_q$ to $true$ and sends its own $\langle req \rangle$ message to process p. On account of the FIFO channel assumption, any acknowledgement to this message in the form of a $\langle no \rangle$ message reaches process p after the $\langle req \rangle$ message from Q and hence after P resets $collecting_p$ to *false*. This implies that job P transited to *executing* before waiting for an acknowledgement from process q, which is a contradiction. $\Box$

Proof of Lemma 5: Let P be at *executing* and consider a process $q \in N_p$. We need to show that $q \in temp_p$. According to algorithm $S2$, process p must have received a $\langle req \rangle$ message from q while $collecting_p$ was *true*. Let Q be the job that sent this $\langle req \rangle$ message. Again, according to the algorithm, $p \in Conf_q$ at the state when $collecting_q$ is set to $true$ by job Q. Now, either $p \in Conf_q$ has held continuously since Q transited to *trying* or q receives an $\langle add \rangle$ message from p after Q transits to *trying*. Consider these two cases separately.

In the first case Q obtained p in $Conf_q$ when it transited to *trying*. From algorithm $S2$, q sent an $\langle add \rangle$ message to p during this transition. By the FIFO property of the channels, this message arrives before the $\langle req \rangle$ message from Q arrives at p. Also, p does not receive any $\langle delete \rangle$ messages from q between the reception of the $\langle add \rangle$ message and the reception of the $\langle req \rangle$ message. So, it is sufficient to show that the $\langle add \rangle$ message is received before P sets $collecting_p$ to $true$. For the sake of a contradiction, assume that the $\langle add \rangle$ message is received after P sets $collecting_p$ to $true$. Then, according to algorithm $S2$, once $collecting_p$ is set to *true*, $flag = (collecting_p \vee state_p = executing)$ remains *true* until P reaches *completed*. Consequently, P does not send a $\langle m_{false} \rangle$ message to q until it reaches *completed*. Since variable $blocking_{qp}$ is initialized to *true* at Q, Q cannot complete the first waiting statement of *trying* until P reaches *completed*. Consequently, Q does not send any $\langle req \rangle$ message to p until after P has reached *completed*, contradicting the assumption that P receives a $\langle req \rangle$ message from Q while at *trying*.

In the second case, q receives an $\langle add \rangle$ message from p after Q transits to *trying*. It is sufficient to show that process p does not receive any $\langle delete \rangle$ messages between the transmission of this $\langle add \rangle$ message and the reception of a $\langle req \rangle$ message from job Q. The proof follows by noting that if p receives such a $\langle delete \rangle$ message then it must have been sent by a predecessor of Q and furthermore, due to the waiting for the acknowledgements of $\langle delete \rangle$ messages and the FIFO property of channels, q could not have received an $\langle add \rangle$ message from p after Q transits to *trying*. This completes the proof of the second case and the proof of the lemma. $\Box$

Proof of Lemma 6: For the sake of a contradiction, consider the first pair of jobs P and Q that violates the lemma. Let m be the $\langle req \rangle$ message that Q receives from P. Let m_1 be the first acknowledgement of m. We consider the cases depending on whether m_1 is a $\langle no \rangle$ or a $\langle req \rangle$ message.

If m_1 is a $\langle no \rangle$ message then consider the $\langle req \rangle$ message m_2 that triggers m_1. According to the algorithm, a job sends at most one $\langle req \rangle$ message to another process. Since P sends the $\langle req \rangle$ message m, the $\langle req \rangle$ message m_2 must have been sent by a job P' preceding P. Since m_2 does not violate the lemma and P' received an acknowledgement for m_2, the first acknowledgement of message m_2 is either a $\langle req \rangle$ message m_3 from q or a $\langle no \rangle$ message triggered by m_2. The latter case is not possible as the $\langle no \rangle$ message m_1 triggered by m_2 is received by P. Therefore, the first acknowledgement of m_2 must be m_3. According to the algorithm, when P' receives m_3, $collecting_p$ is *true* and m_3 does not trigger a $\langle no \rangle$ message. Since message m_3 does not violate the lemma, it is acknowledged by a $\langle req \rangle$ message that is sent by P'. In other words, m_3 is acknowledged by message m_2. Therefore when m_2 is received, $collecting_q$ is *true* and m_2 does not trigger m_1, a contradiction.

Now consider the case in which m_1 is a $\langle req \rangle$ message. Since message m violates the lemma, m_1 is sent by either a job Q' preceding Q or a job Q'' succeeding Q. In the former case, since m_1 does not violate the lemma, it is acknowledged by either a $\langle req \rangle$ message from P or a $\langle no \rangle$ message triggered by it. The second case is not possible as m_1 is the first acknowledgement of m and it does not trigger any $\langle no \rangle$ message. Thus the first case must be true and therefore, Q' receives message m as an acknowledgement to m_1. Consequently, Q does not receive m, a contradiction. In the latter case, while P is waiting for an acknowledgement, job Q is completed and job Q'' transits to *trying*. Since message m_1 is later sent by Q'', Q'' must have obtained $p \in Conf_q$ and initialized $blocking_{qp}$ to *true* when it transits to *trying*. Since P does not send any $\langle m_{false} \rangle$ message while $collecting_p$ is *true*, Q'' cannot pass the first waiting statement of *trying* and message m_1 is not sent, a contradiction. □

Proof of Lemma 7: For the sake of a contradiction, assume that the lemma is *false*. Consider the first $\langle req \rangle$ message m that violates the lemma. Assume that message m is sent from job P to job Q. Consider the two cases: either m is not acknowledged, or there are two or more acknowledgements for it. In the former case, job P remains at state *trying* forever, waiting for an acknowledgement. From algorithm $S2$, m does not trigger any $\langle no \rangle$ message, and Q receives m when $collecting_q$ is *true* and includes p in N_q. By Lemma 5, Q must have sent a $\langle req \rangle$ message m_1 to p when it set $collecting_q$ to *true*. Message m_1 must have been received by job P or job P' preceding P. If m_1 is received by job P then since m is not acknowledged, m_1 must have been received before $collecting_p$ is set to *true* and a $\langle no \rangle$ message must have been sent to q. Consequently, Q receives at least two acknowledgements for messages m_1, contradicting the assumption that m is the first message that violates the lemma. On the other hand, if m_1 is received by P' then there exists a message m sent from process p to process q which can serve as an acknowledgement to m_1. Therefore by Lemma 6, the first acknowledgement for m_1 is sent by P'. Thus, Q receives at least two acknowledgements for message m_1, contradicting the assumption that m is the first message that violates the lemma.

In the latter case, by Lemma 6, the first acknowledgement of message m is either a $\langle req \rangle$ message m_2 sent by Q or a $\langle no \rangle$ message triggered by m. In the first case, message m_2 is sent before message m is received by Q. Therefore, message m_2 has at least one acknowledgement. Also, when P receives m_2, $collecting_p$ is true and m_2 does not trigger a $\langle no \rangle$ message. Therefore, by Lemma 6, the first acknowledgement of m_2 is a $\langle req \rangle$ message sent by P. In other words, the first acknowledgement of m_2 is message m. Consequently, message m does not trigger a $\langle no \rangle$ message and the second acknowledgement of m must be a $\langle req \rangle$ message m_3 sent by a job Q'' succeeding Q. By repeating the argument as in Lemma 6, Q'' cannot pass the first waiting statement of *trying* while P is waiting at *trying*. Therefore, Q'' does not send message m_3, a contradiction. Now consider the second case in which the first acknowledgement of m is a $\langle no \rangle$ message triggered by m. Since m triggers a $\langle no \rangle$ message, the second acknowledgement of m must be a $\langle req \rangle$ message m_4 sent by either Q or a job Q'' succeeding Q. If m_4 is sent by Q'' then the argument as in Lemma 6 can be repeated again, leading to a contradiction. Otherwise, if m_4 is sent by Q then let the global state at which Q received m be s and the global state at which Q sends m_4 be s'. When Q receives message m at state s, it sets $blocking_{qp}$ to *true*. According to algorithm $S2$, P sets $collecting_p$ to *true* when it sends message m. By definition, $collecting_p$ remains *true* until the second acknowledgement m_4 is received by P. Also, P does not send any $\langle m_{false} \rangle$ message while $collecting_p$ is *true*. Therefore, according to the algorithm, $blocking_{qp}$ remains *true* at q between the states s and s'. However, since message m_4 is sent at state s', Q must have found $blocking_{qp}$ to be *false*, which is a contradiction. This completes the proof of the lemma. □