Dynamic Resource Allocation Using Local Views*

Manhoi Choy and *Ambuj K. Singh*

Department of Computer Science
The Hong Kong University
of Science and Technology
Clear Water Bay, Hong Kong

Department of Computer Science
University of California
at Santa Barbara
Santa Barbara, California 93106

Abstract

Conflicting requests to shared resources arise naturally and frequently in distributed systems. The resolution of these conflicts in a fair and timely manner is important. We consider a general model in which processes request repeatedly and asynchronously for arbitrary subsets of resources. Our solution to this dynamic problem first computes a collection of process-specific local *views* that capture the conflicts for each process, and then resolves the conflicts on these views using existing algorithms. We specify the requirements on local views, and present an algorithm satisfying the requirements. The time and message complexities of the composite algorithm for dynamic conflict resolution depend minimally on global parameters and offer a significant improvement over existing algorithms.

Keywords: distributed algorithms, conflict detection, resource allocation, synchronization.

1 Introduction

Conflicting requests to shared resources arise naturally in parallel and distributed systems. In databases, transactions need locks on shared records and files; in operating systems, processes and threads synchronize with the help of semaphores and barriers; in concurrent object-oriented programming, threads invoke methods on common objects; in parallel programming, processes request exclusive access to a set of shared variables. Timely resolution of such concurrent requests is necessary for individual progress and systemwide efficiency. Patterns of access and conflict can be categorized on the basis of two attributes. The first attribute relates to the number of resources requested by a process (or thread or transaction); either a process requests a *single* resource or it requests *multiple* resources. The second attribute describes whether requests for resources are *incremental* or *single-step*. In incremental requests, a requesting process holds onto previously granted resources while making further requests. In the *single-step* case, a process requests its resources all at once. Synchronized methods in the programming language Java [1] operate in the single-step, single-resource model; a thread requests exclusive access to an object and a monitor is used to grant such requests. Databases transactions operate in the incremental single-resource model; a

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

transaction requests its resources as needed one at a time while holding on to its previously granted resources. Some parallel programs operate in the single-step, multiple-resource model; here variables are partitioned into regions and a program requests access to a whole region at some time [11]. The simplest request model is the single-step, single-resource kind: deadlocks are not possible and resources can be allocated without any global considerations. The incremental request models (both single-resource and multiple-resource) admit the possibility of deadlocks and run-time support for detection and resolution of deadlocks is necessary. There is little to be gained by using efficient algorithms for resolving conflicts here as the overhead of deadlock prevention or resolution (e.g., copying all changed objects) can be quite high. In this paper, we assume the single-step multiple-resource model for requests. This model is useful in distributed systems and parallel programs. Careful selection among competing overlapping requests can avoid deadlocks and reduce the response time, thus emphasizing the need for efficient algorithms for this model.

Even within the adopted framework of the single-step multiple-resource model, there can be subtle variations. In the dining philosophers model, processes lie over a ring and conflicts can occur only between neighboring processes [9]. In a generalization of the problem, processes lie on the vertices of a *conflict graph* and conflicts occur between every pair of neighboring processes [6, 7]. The conflicts are determined *statically* in the sense that the conflict graph does not change over time. In another generalization, called the drinking philosophers problem, a set of resources is defined statically for each edge in the conflict graph. In each iteration (request-use-release cycle), a process is interested in a subset of the resources on its adjoining edges. However, the underlying graph itself and the set of resources associated with each edge are determined statically. We consider the case of dynamic requests in this paper: requested resources are determined completely at runtime and there is no restriction over the resources requested by a process in different iterations.

Allocating resources efficiently in an asynchronous distributed system is a difficult problem. Not only are the processes working at independent speeds making coordination difficult, but the decision making needs to be distributed and independent. Using a central coordinator for resolving conflicts impacts the performance (all requests will get queued at this site) as well as the fault-tolerance (failure of this node brings down the whole system). We are interested in algorithms which scale with the size of the conflict graph. The performance of the algorithm should be governed by the degree of conflict in the system (implying that when there are no conflicts, a process should obtain its resources immediately) and should depend as little as possible on global parameters such as the number of processes.

The resolution of conflicts among processes is especially difficult when the processes iterate asynchronously and continuously change the conflict relationship without any global coordination. Instead of developing an algorithm from first principles, our solution as illustrated in Figure 1 reduces the dynamic resource allocation problem to three subproblems. The first subproblem, *conflict detection*, extracts a consistent local view of the conflicts for a process. A process obtains this view at the beginning of each iteration (request-use-release cycle) and uses it as a static value for the iteration. The collection of these views (one per process) defines a conflict graph that is static and essentially maps the dynamic conflicts to a set of computed static conflicts. The second subproblem, *node coloring* colors the the vertices of the local views so that neighboring vertices are assigned different colors. The final subproblem, *static resource allocation* (or *dining philosophers*), resolves the conflicts over a static colored conflict graph in a fair and efficient manner. The latter two subproblems have been investigated previously by other researchers and ourselves; we use some of these existing algorithms. The *conflict detection* problem is new and the main subject of this paper.

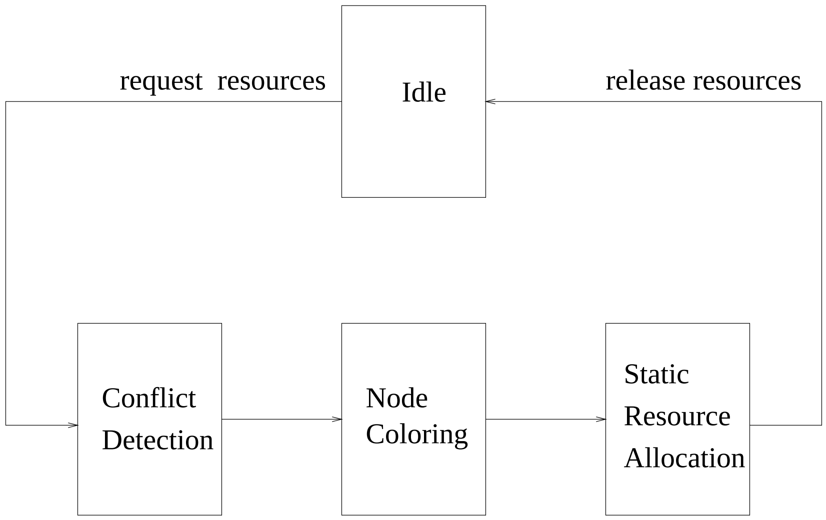

Figure 1: Structure of our resource allocation algorithm

If conflict detection has to be solved only one-time, a global snapshot algorithm such as Chandy and Lamport [5] suffices – the global snapshot algorithm is run and the conflicting processes are identified. However, in a distributed resource allocation problem environment, conflict detection is needed repeatedly. Applying global snapshot algorithms, in an attempt to detect dynamic conflicts, is not a trivial task. Fairness and performance complexity issues have to be addressed. Nevertheless, there is no need to take snapshots globally as conflicts only need to be detected among the conflicting processes. Conflict detection has also been studied (although indirectly) in the shared memory context, under the titles of atomic snapshot objects and wait-free synchronization [2]. An atomic snapshot object is a wait-free implementation of scan and update operations from low-level atomic read and atomic write operations. The result of a scan operation depends on the execution of update operations on the object. Conflict detection in a distributed resource allocation problem environment can be thought of as providing a conflict detection operation from low-level message send and receive operations. There is no explicit update operation in our distributed environment as processes simply inform each other of their resource requirements by messages.

We are interested in two performance measures: *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. In measuring response time, we assume an upper bound ν on the message delivery time and an upper bound τ on the time to finish a started job. Both bounds of τ and ν are used only for performance analysis and not by the algorithm. The maximum degree of conflict of a process p (i.e., the degree of node p in the conflict graph) is denoted by δ_p and the maximum of all δ_ps is denoted by δ. The number of processes is bounded by M.

For the static resource allocation problem, earlier solutions had a response time of either $O(M)$ or exponential in δ [7, 14]. The best known solution has a response time of $O(\delta^2)$ and a message complexity of $O(\delta)$ [8]. For the dynamic resource allocation problem, the algorithm by Weidman et al. [15] has a response of $O(M)$ and a message complexity of $O(\delta)$. Designing an algorithm with a response time and message complexity that are independent of the size of the system is a major challenge. This is particularly important when processes compete for resources with a small set of neighbors, i.e., when δ is much smaller than M. The previously best known deterministic solution was presented by Awerbuch and Saks [3]. Their solution employs a distributed queue technique and uses multiple levels of queues. Processes enter a queue at a high level and gradually move to the front of the queue. Once at the front of a queue, a process proceeds to the queue at the next lower level. Processes gradually resolve conflicts in this manner until they reach the front of the queue at the lowest level, whence they are granted the needed resources. Their algorithm achieves a response time of $O(\delta^2 log\ M)$ and a message complexity of $O(\delta^2 log\ M)$, where M is the number of processes in a system. Our algorithm presented in Section 5 has a response time of $O(\delta^2 + \log^* M)$ and a message complexity of $O(\delta^2 + \delta \log^* M)$. The response time and message complexity of our algorithm are better by a factor of $\log M$. Most importantly, however, our algorithm minimizes the dependence of the complexity measures on global parameters.

Bar-Ilan and Peleg [4] consider the dynamic resource allocation problem in a *synchronous* environment with a *complete* communication graph. According to their definitions, a communication graph is complete if every process is connected to every other process by a link on which messages can be sent and received. Communication is synchronous if all the processors work at the same rate and a message sent from processor p connected to processor q at time t arrives at time $t+1$. Unlike the asynchronous model used in this paper, they require an underlying clock synchronization protocol that signals every processor at every clock increment. They modify the distributed queue algorithm of Awerbuch and Saks to obtain an algorithm with a response time of $O(\delta \log M)$ and a message complexity of $O(\delta \log M)$. The cost due to the clock synchronization protocol is not included in these terms.

Recently, Rhee [13] considers reducing the impact of τ on the worst case response time. He proposes a modular algorithm based on a distributed queue and using any existing resource allocation algorithm. The resulting algorithm improves the existing resource allocation algorithm (asymptotically), when certain conditions relating the critical section duration, the message delay, and the degree of conflict hold. For instance, if Rhee's algorithm and our algorithm in Section 5 are combined, the resulting time complexity becomes $O(\delta\tau + (\delta^2 + \log^* M)\nu)$ instead of $O(\delta^2\tau + (\delta^2 + \log^* M)\nu)$. This improves the bound of $O(\delta\tau + (\delta^3 + \delta \log^* M)\nu)$ as demonstrated in [13]. The message complexity of the resulting algorithm remains unchanged at $O(\delta^2 + \delta \log^* M)$.

The rest of this paper is organized as follows. Section 2 defines the system model and formulates the resource allocation problem. In Section 3, we specify the conflict detection problem. In Section 4, we present an algorithm for conflict detection. Section 5 uses this algorithm to solve the resource allocation problem. Section 6 contains a summary and concluding remarks.

2 System Model

2.1 Identifying Process States

We assume an asynchronous message passing system composed of a set of processes and FIFO channels. A process iterates through a request-use-release cycle illustrated in Figure 1 in which some resources are requested, used, and then released. Each iteration of a process over the cycle defines a *job*. A job of a process identifies a set of objects, and the desired operations on them (identification of operations is necessary since commuting operations such as read-only operations on the same object can proceed in parallel). Each object is controlled by a resource manager. When submitting a job, a process communicates with the relevant resource managers by informing them of the desired operations. Each resource manager subsequently determines the set of conflicting jobs for the object and informs the process. To distinguish jobs from the processes issuing them, we use upper case letters to denote jobs and lower case letters to denote processes. For example, P denotes a job associated with process p, Q denotes a job associated with process q, and so on. We assume that there is a finite number M of processes in the system.

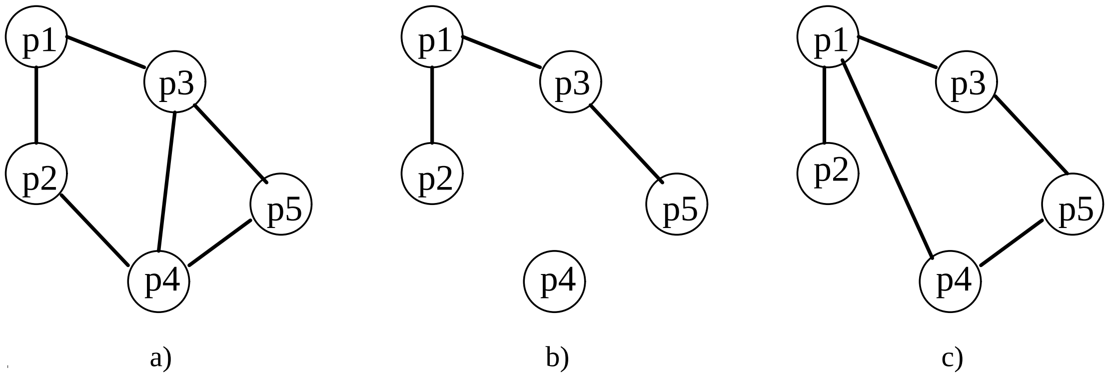

Figure 2: Example of a dynamic conflict graph

We represent conflicts between processes (or the associated jobs) in the form of a *conflict graph* with processes at the nodes and an edge between each pair of conflicting processes. Processes that are neighbors in the conflict graph conflict on some shared resource and cannot execute concurrently. Edges may be labeled by naming the resource on which the conflict occurs, but this information is typically not used in the resource allocation algorithms. Also, for a dynamic conflict graph this label would need to change as a process requests different resources. Two processes can also conflict over multiple resources; this is also captured by a single edge in the conflict graph.

Figure 2.a shows an example of a conflict graph with five processes $p1 \ldots p5$. As shown in the figure, process $p1$ conflicts with processes $p2$ and $p3$ but it does not conflict with process $p4$ or process $p5$. Similarly, process $p2$ conflicts with processes $p1$ and $p4$, process $p3$ conflicts with processes $p1$, $p4$, and $p5$, process $p4$ conflicts with processes $p2$, $p3$, and $p5$, and process $p5$ conflicts with processes $p3$ and $p4$. Since a process may submit different jobs over time, the conflict graph is dynamic: the set of nodes remains fixed but the set of edges keeps changing. For example, if the current job of process $p4$ in Figure 2.a is serviced and it releases the shared objects then all adjoining edges of process $p4$ are removed and we obtain the conflict graph shown in Figure 2.b. At this time, process $p4$ may submit a new job that conflicts with processes $p1$ and $p5$. The resulting

conflict graph is shown in Figure 2.c.

Asynchronous and independent iteration of the processes over the outline shown in Figure 1 generates a dynamic conflict graph. Since it is difficult to design conflict resolution schemes from first principles that accommodate such dynamic behavior, our solution strategy is to isolate a static graph from the dynamic graph and then apply existing solutions that work well for static graphs. This static conflict graph is called a *view* and it consists of a set of *local views*, one for each process. The static graph is then colored and then a static resource allocation such as a dining philosophers algorithm is used on the colored graph. To be useful, local views should satisfy some properties. For example, process $p4$ should find processes $p2, p3$, and $p5$ in its local view in Figure 1.a and processes $p1$,$p5$ in its local view in Figure 1.c. Similarly process $p1$ should find processes $p2$ and $p3$ in its local view in Figure 1 a). In order to specify the requirements on the local views (this specification is investigated in Section 3 and its implementation in Section 4), we first formalize the state transitions of a process.

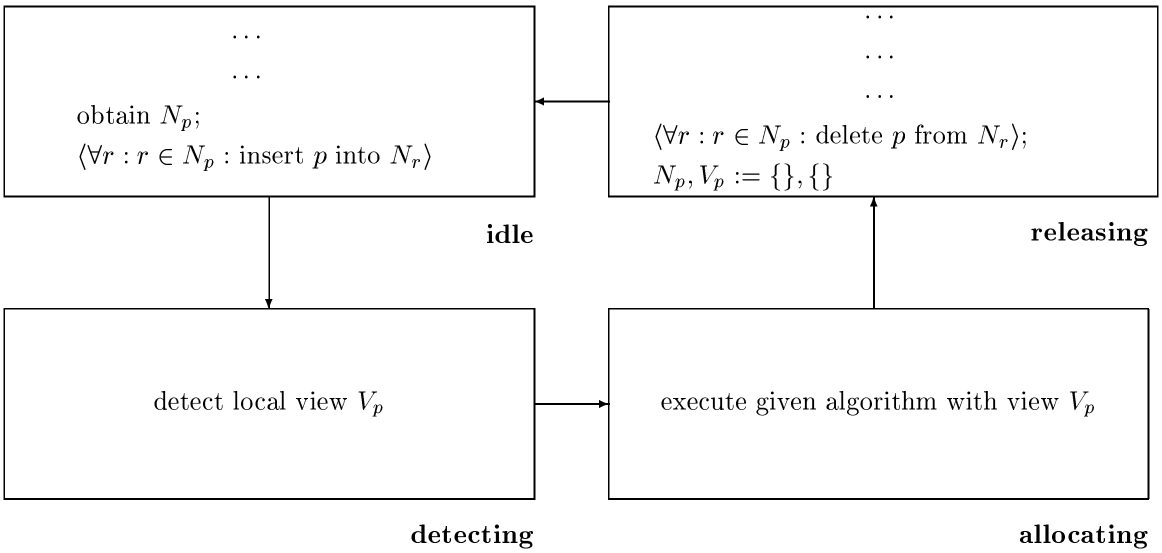

Figure 3: States of user process p

We model a process by introducing four local states (or simply states): **idle**, **detecting**, **allocating**, and **releasing** as shown in Figure 3. Initially, a process p is at **idle**. To submit a job, process p contacts the necessary resource managers and obtains the set of conflicting (or neighboring) processes N_p. It then sends an insert message to each process $r \in N_p$, informing them of the conflict. (This message is necessary since process r may have contacted the resource managers before process p and may be unaware of the conflict.) Upon reception of this message, process r adds p to N_r and returns an acknowledgement to p. Once process p receives all acknowledgements, it transits to the **detecting** state. The insert message and its response are collectively referred to as an insert operation. The **detecting** state corresponds to the construction of the local views. Process p executes a view algorithm to obtain a local view V_p and then transits to state **allocating**. Set V_p is used for the execution of a user-specified computation at this state. Typically, this user-specified computation would be a solution to the user's problem under a static environment. The technique allows the user to solve the problem in a dynamic environment based on a solution in a static environment. For example, in Figure 1 both node coloring and the static resource allocation algorithm may be invoked at this state[1]. When a process p has completed its computation at state **allocating**, it transits to state **releasing**. After finishing any application-specific computation at state **releasing** (this is indicated in Figure 3 by dots), process p deletes itself from all conflict sets. This is achieved in a manner similar to the case of insert[2]. When all delete operations are completed, process p resets N_p and V_p to empty, and transits to state **idle**.

Processes iterate asynchronously and independently through the four states indicated in Figure 3. Each iteration of a process through the four states defines a job. A job is said to be *active* if it corresponds to the current iteration; otherwise it is said to be *inactive*. Since a process is associated with exactly one active job at a time, we can refer to the state of an active job as the state of the corresponding process. The waiting for acknowledgements for insert and delete messages (Figure 3) ensures the symmetry of conflict sets of any two jobs.

Property 1 *For any two active jobs P and R that have not sent any delete messages, $p \in N_r \equiv r \in N_p$.*

The global state of the system consists of an assignment of values to all the local variables of processes and to the communication channels. Initially, all the channels are empty and all processes are at **idle**. The execution of the system is modeled by a sequence of global state transitions. A global state transition brings the system from one global state to another and consists of a local state transition, a message transmission, or a message reception. A history is formed by a sequence of global state transitions. A global state s *precedes* another global state t in a given history if s occurs prior to t in the history.

2.2 Notation

We use the following linear time temporal logic operators [12] in our specifications and proofs.

- $\Box\, p$ means that p holds currently and at all future global states.
- $\boxminus\, p$ means that p holds currently and at all previous global states.
- $\Diamond\, p$ means that p holds currently or at some future global state.
- $\diamondminus\, p$ means that p holds currently or at some previous global state.
- $\bigcirc\, p$ means that p holds in the next state.
- $p\ \mathcal{U}\ q$ means that q holds eventually and p holds until then.
- $p\ \mathcal{S}\ q$ means that q held in the past and p holds from then up to the current global state.
- $p\ \mathcal{W}\ q$ means that either $p\ \mathcal{U}\ q$ or $\Box\, p$.
- $p\ \mathcal{B}\ q$ means that either $p\ \mathcal{S}\ q$ or $\boxminus\, p$.

[1] As we will see in Section 5, we obtain the most efficient solution by performing both these tasks at **releasing** and performing a null computation at **allocating**.

[2] In order to simplify the presentation, we assume that a process delays the processing of insert messages until its pending delete messages are acknowledged.

Processes interact by sending messages of different types, e.g., an insert or a delete message. For a message type T, predicate $send(T, p, r)$ is true in the state at which p sends a message of type T to r. This message can also be explicitly named; this is then denoted as $send(T(m), p, r)$, or simply $send(m, p, r)$. Likewise, $rcv(T, p, r)$ is a state predicate that is true in the state in which r receives a message of type T from p. Also, we denote $\diamondsuit\ send(T, p, r)$ by $sent(T, p, r)$ and likewise $\diamondsuit\ rcv(T, p, r)$ by $rcvd(T, p, r)$. Instead of associating a process with a message transmission/reception in the above predicates, we sometimes name the specific jobs that sent or received the message, e.g., $send(m, P, r), send(m, p, R), send(m, P, R)$. A field may also be left unspecified as a do-not-care term in the above notation. Sometimes, we also indicate the type of a message in quantifications, e.g., $\langle \exists\ insert\ e :: \ldots \rangle$, or $\langle \forall delete\ e\ :: \ldots \rangle$.

It is convenient to explicitly name states and state intervals. When there is no ambiguity, we overload $send(T, p, r)$ and $rcv(T, p, r)$ to mean the global states at which the messages are sent and received, respectively. Also, given a history and global states s and t such that s precedes t, the open interval $(s, t]$ includes all states between s and t, excluding s and including t. The closed interval $[s, t]$, on the other hand, includes all states between s and t, including s and t. The notation $\phi\ in\ J$ means that condition ϕ holds at all states in the interval J. When the interval of interest contains only one state s, we simply say ϕ at s.

3 Specification of Conflict Detection

During conflict detection, a process obtains a local view. The local views of processes (denoted $V_p, V_r, \cdots$) are required to be based on the neighborhoods $N_p, N_r, \cdots$ of the conflict graph. Intuitively, we require the following conditions on the local views: (a) the computed local views $V_p, V_r, \cdots$ should capture a conflict when one exists based on the neighborhoods $N_p, N_r, \cdots$, (b) the computed local views should not capture a conflict when one does not exist in the neighborhoods, (c) the computed local views should be symmetric, and (d) the computation of local views should terminate.

In order to motivate the specification, consider the local view V_p of a job P of process p that is at **allocating**. Consider any other process r in the system. The only interesting cases are when r is currently at **allocating**, **releasing**, or **detecting**. The first two cases lead to isolation of safety properties and the last case leads to isolation of progress properties. We consider these three cases next.

Case 1: (Process r is at **allocating**). We consider two subcases based on whether $r \in N_p$. First, consider the subcase in which $r \in N_p$. On account of Property 1 (previous section), $(p \in N_r)$. So, there is a conflict between N_p and N_r, and this conflict should be reflected in the computed local views V_p and V_r. A desired property of view computation therefore is

Requirement 1 $p\ at$ **allocating** $\wedge\ r\ at$ **allocating** $\wedge\ p\ \in N_r \wedge r \in N_p \quad \Rightarrow \quad p \in V_r \wedge r \in V_r$.

We call this condition *non-triviality* since it enforces that local views are not trivially empty. Intuitively, this condition states that two neighboring processes (implying $p \in N_r \wedge r \in N_p$) that are simultaneously at **allocating** must include each other in the computed local views. This property bounds the size of local views from below and ensures that all necessary conflicts are reflected in the local views.

Now, consider the latter subcase, i.e., $\neg(r \in N_p)$. On account of Property 1, $\neg(p \in N_r)$. In this case, the absence of conflict in N_p and N_r should imply an absence of conflict in the computed local views V_p and V_r. We restate this using the contrapositive, i.e., a conflict in the computed local views V_p and V_r should occur only on account of a conflict between N_p and N_r. In other words, if $r \in V_p$ then there should have been a previous global state at which p and r are at **detecting**, and $p \in N_r \wedge r \in N_p$. But requiring a previous global state for process p without identifying a job (or iteration) of process p can be problematic. So, we use a specific job of process p (say P) to specify this property:

Requirement 2 $(P\ \mathit{at}\ \textbf{allocating}\ \mathit{and}\ r \in V_p) \quad \Rightarrow$
$$\diamondminus\, (P\ \mathit{at}\ \textbf{detecting}\ \wedge\ r\ \mathit{at}\ \textbf{detecting}\ \wedge p \in N_r \wedge r \in N_p).$$

Intuitively, this condition states that for a process r to be a member of the local view of a process p, the two processes must have overlapped and conflicted (i.e., $p \in N_r \wedge r \in N_p$) during the computation of local views. It ensures that no unneeded conflicts are captured by the local views, effectively bounding the size of the local views from above.

Case 2: (Process r is at **releasing**). If processes p and r were ever at **allocating** simultaneously then the needed requirements on local views follow from the two conditions defined in Case 1. Therefore, we only consider the case in which p and r are never at **allocating** simultaneously. Since process r is at **releasing** currently, this means that we need to consider the situation in which process r transited to **releasing** before process p transited to **allocating**. In this case, we require the symmetry of local views, i.e., if process p finds process r in its local view then process r should also find p in its local view. Stating this condition requires us to again name jobs instead of processes. Let P be the current job of process p that is at **allocating**. If $r \in V_p$ then we require that some job R of process r overlapped and conflicted with P during the computation of local views. Instead of stating this condition on its own, we combine it with requirement R2 stated earlier. This results in the following.

Requirement 3 $(P\ \mathit{at}\ \textbf{allocating} \wedge r \in\ V_p) \quad \Rightarrow$
$$\langle \exists R :: (P\ \mathit{at}\ \textbf{allocating}\ \wedge (R\ \mathit{at}\ \textbf{allocating}\ \vee R\ \mathit{at}\ \textbf{releasing})) \Rightarrow p \in V_r) \quad \mathcal{S}$$
$$(P\ \mathit{at}\ \textbf{detecting}\ \wedge R\ \mathit{at}\ \textbf{detecting}\ \wedge\ p \in N_r \wedge r \in N_p)\rangle.$$

Intuitively, this condition states that for a job P to find process r in its local view, some job R of process r must have overlapped and conflicted with P and this job will symmetrically find p in its local view. We term this condition *Validity*.

Case 3: (Process r is at **detecting**). In this case we should ensure that process r should complete the computation of its local view and not remain at **detecting** forever. This progress cannot be guaranteed if a neighboring process p remains at **allocating** forever. This is because process p may have preceded process r at **detecting** and the previous safety requirements prohibit the transition of process r to **allocating**. (If the processes are simultaneously at **allocating** then Requirement 1 requires that the processes include each other in their local views. But Requirement 3 then requires that the processes overlap their computation of local views; a contradiction.) So, we require progress of process r provided all other processes terminate their computation at **allocating**. This results in the following requirement.

Requirement 4 $\langle \forall P :: \; \Box\, (P \; at \; \mathbf{allocating} \Rightarrow \; \Diamond\, (P \; at \; \mathbf{releasing})) \rangle \;\; \Rightarrow$
$\langle \forall P :: \; \Box\, (P \; at \; \mathbf{detecting} \Rightarrow \; \Diamond\, (P \; at \; \mathbf{allocating})) \rangle$.

Informally, the above condition means if every job at **allocating** eventually transits to **releasing** then every job at **detecting** eventually transits to **allocating**. One way of satisfying the antecedent of the above progress property is for processes at **allocating** to engage in a wait-free computation or simply a null computation. Waiting for a process at **detecting** to transit to **allocating** would not satisfy the antecedent.

(End of case analysis)

Summarizing the above discussion, the complete specification of conflict detection is shown in Figure 4.

Non-triviality	$p \; at \; \mathbf{allocating} \wedge \; r \; at \; \mathbf{allocating} \wedge p \in N_r \wedge r \in N_p \;\; \Rightarrow$ $p \in V_r \wedge r \in V_r$
Validity	$(P \text{ at } \mathbf{allocating} \wedge r \in \; V_p) \;\; \Rightarrow$ $\langle \exists R :: (P \text{ at } \mathbf{allocating} \; \wedge (R \text{ at } \mathbf{allocating} \; \vee R \text{ at } \mathbf{releasing})) \Rightarrow p \in V_r) \quad \mathcal{S}$ $(P \text{ at } \mathbf{detecting} \; \wedge R \text{ at } \mathbf{detecting} \; \wedge \; p \in N_r \wedge r \in N_p) \rangle$
Conditional local progress	$\langle \forall P :: \; \Box\, (P \text{ at } \mathbf{allocating} \Rightarrow \; \Diamond\, (P \text{ at } \mathbf{releasing})) \rangle \;\; \Rightarrow$ $\langle \forall P :: \; \Box\, (P \text{ at } \mathbf{detecting} \Rightarrow \; \Diamond\, (P \text{ at } \mathbf{allocating})) \rangle$

Figure 4: Requirements of conflict detection

4 An Algorithm for Conflict Detection

In this section we present an algorithm for conflict detection. The algorithm is used later in Section 5 for obtaining an efficient solution to the resource allocation problem.

The algorithm is shown in Figure 5. It divides the **detecting** period into two phases. Processes synchronize in these two phases to identify the members of their local views. A Boolean variable *collecting* is introduced at each process. The variable is $false$ in the first phase and *true* in the second phase. A process waits until it is not blocked by any other process before entering the second phase and only processes in the second phase or beyond may belong in the local view of another process. A local variable $blocking_{pr}$ is introduced for each process r in N_p. This variable is false initially when p is **idle**; it is set to true for each $r \in N_p$ prior to the issuance of the insert operation to process r. Process p remains in the first phase of the **detecting** period until $blocking_{pr}$ is *false* for each process r in N_p. Then it sets variable $collecting_p$ to *true*, sends a $\langle req \rangle$ message to each process in N_p, and copies the set N_p in variable $temp_p$. These are done in a single atomic step so that no messages are received from the time the local predicate $\langle \forall r \in N_p : \neg blocking_{pr} \rangle$ becomes *true* to the time N_p is copied to $temp_p$. Besides, N_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 r iff it observed $blocking_{pr}$ to be *false* and if it sent a $\langle req \rangle$ message to r. After the atomic statement, the process transits to the second phase of the **detecting** period. Here process p waits until an acknowledgment 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, it transits to **allocating** and sets variable $collecting_p$ to *false* (handler G1).

Upon receiving a $\langle req \rangle$ message from process r (handler G2), if process p is "concurrent" with r, i.e., $collecting_p$ is *true*, then r is inserted into V_p. Otherwise process r is not "concurrent" and a $\langle no \rangle$ message is returned to r. In either case, variable $blocking_{pr}$ is set to $true$. The arrival of a $\langle no \rangle$ messages is recorded in handler G3. As in Algorithm $S1$, r is removed from V_p if a delete operation from r is *executed* at p while $collecting_p$ is *true* (handler G4).

The algorithm uses $\langle release \rangle$ messages to unblock processes that may be waiting in the first phase of **detecting**. Process p sends a $\langle release \rangle$ message when an insert operation from r is executed while process p is in the first phase of **detecting** or at **idle**, or at **releasing** (handler G6). If the insert operation is executed when p is in the second phase of **detecting** or at **allocating**, r is saved in set $pending_p$ and no $\langle release \rangle$ messages are sent to unblock r. In this case, a $\langle delete \rangle$ message is sent when process p transits to **releasing** (handler G7). Along with this state transition, set $pending_p$ is also reset. Variable $blocking_{pr}$ is set to $false$ when a $\langle release \rangle$ message is received from r (handler G5) or when an insert operation is executed in a non-idle state (handler G6). Note that r is added to V_p by a $\langle req \rangle$ message (handler G2), and removed from V_p by a delete operation (handler G4).

View computation of process p at **detecting**

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

Event handlers	
(G1)	**On** transiting to **allocating** $collecting_p := false$

(G2)	**On** *receiving message* $\langle req \rangle$ *from* r
	  $blocking_{pr} := true$;
	  if $collecting_p$ **then**
	      $V_p := V_p \cup \{r\}$
	  else send $\langle no \rangle$ to r **endif**

(G3)	**On** *receiving message* $\langle no \rangle$ *from* r record the arrival of the message
(G4)	**On** *the execution of a delete operation issued by* r **if** $state_p =$ **detecting then** $V_p := V_p - \{r\}$
(G5)	**On** *receiving message* $\langle release \rangle$ *from* r $blocking_{pr} := false$;

(G6)	**On** *the execution of an insert operation issued by* r
	  if $state_p \neq$ **idle then** $blocking_{pr} := false$;
	  if ($collecting_p \vee state_p =$ **allocating**) **then** $pending_p := pending_p \cup \{r\}$
	  else send $\langle release \rangle$ to r

(G7)	**On** transiting to **releasing** $\langle \forall r \in pending_p :$ **send** $\langle release \rangle$ to $r \rangle; pending_p := \{\}$

Figure 5: Conflict detection algorithm

4.1 Proof of Correctness

We next prove that the algorithm satisfies *non-triviality*, *validity*, and *conditional local progress*. First, we introduce some nomenclature. We say that a $\langle req \rangle$ message or a $\langle no \rangle$ message m from r is an *acknowledgement* of a $\langle req \rangle$ message sent by a job P to r if between the state immediately after P sent the $\langle req \rangle$ message and the state P received m, $collecting_p$ is continuously *true*. Formally, a $\langle req \rangle$ message or a $\langle no \rangle$ message n from r to p is an acknowledgment of a $\langle req \rangle$ message m from p to r if

$sent(m, p, r) \wedge \neg rcvd(n, r, p) \Rightarrow collecting_p$.

This is denoted as $Ack(m, n)$. Define predicate $FirstAck(m, n)$ to be true if message n is the first acknowledgment for message m, i.e., if it is received prior to any other acknowledgments. The following properties $(B1)$–$(B4)$ follow directly from the algorithm. We assume an implicit universal quantification over P, p, r.

$(B1)$ $\Box\ ((r \in V_p \wedge (P \text{ at } \textbf{allocating})) \quad \Rightarrow \quad \diamondsuit\ (collecting_p \wedge rcv(req, r, P)))$.

$(B2)$ $\Box\ (send(req, p, r) \quad \Rightarrow \quad ((P \text{ at } \textbf{detecting}) \wedge r \in N_p))$.

$(B3)$ $\Box\ (rcv(req, r, p) \wedge collecting_p \quad \Rightarrow \quad ((P \text{ at } \textbf{detecting}) \wedge r \in V_p))$.

$(B4)$ $\Box\ (((P \text{ at } \textbf{allocating}) \wedge (r \in N_p\ \mathcal{S}\ (\neg collecting_p \wedge\ \bigcirc\ collecting_p))) \Rightarrow$
$(\diamondsuit\ (send(req(m), P, r) \wedge (P \text{ at } \textbf{detecting})) \wedge$
$\diamondsuit\ (\langle \exists n : Ack(m, n) : rcv(n, r, P) \rangle \wedge (P \text{ at } \textbf{detecting}))))$

Property (B1) states that if a job P at **allocating** has included r in its local view, then it must have received a $\langle req \rangle$ message from r while $collecting_p$ is true. Property $(B2)$ states that processes send $\langle req \rangle$ messages only to conflicting neighbors at state **detecting**. Property $(B3)$ states that if the flag $collecting_p$ is true just after P receives a $\langle req \rangle$ message from r, then P must be at state **detecting** with $r \in V_p$. This property follows from handler $(G2)$. Property $(B4)$ states that for a process P at **allocating**, if $r \in N_p$ holds continuously since the setting of $collecting_p$ to true, then process P must have sent a $\langle req \rangle$ to r and this message must have been acknowledged.

Property (B5) below states that the first acknowledgment to a $\langle req \rangle$ message is sent by the job receiving the $\langle req \rangle$ message. Property (B6) is a related progress property that specifies time bounds. Both of these properties are proved in Appendix B.

$(B5)$ For any $\langle req \rangle$ message m sent from process p to r that is received by job R, if $FirstAck(m, n)$ then n is sent by R.

$(B6)$ Each $\langle req \rangle$ message is acknowledged within 2ν time units and no $\langle req \rangle$ message is acknowledged more than once.

The proofs of *non-triviality* and *validity* also make use of the following properties of user processes proved in Appendix A. These properties follow from the process outline shown in Figure 3.

Property 2 *For any job P,* $\Box\ ((\neg(P \textit{ at } \textbf{idle}) \wedge \neg\langle \exists (delete\ e) :: sent(e, P, -) \rangle) \quad \Rightarrow$
$(r \in N_p \Rightarrow p \in N_r))$.

Property 3 *For any two jobs P and R,*

$\Box\ (((P\ at\ \textbf{allocating}) \wedge (R\ at\ \textbf{allocating}) \wedge r \in N_p \wedge p \in N_r) \quad \Rightarrow \quad (\phi(P,R) \vee \phi(R,P))),$

where formula ϕ is defined as follows:

$\phi(X,Y) \equiv \langle \exists (insert\ e) :: (y \in N_x\ \mathcal{S}\ send(e,X,Y)) \wedge (x \in N_y\ \mathcal{S}\ rcv(e,X,Y)) \ \wedge$

$\quad \boxminus\ (sent(e,x,y) \Rightarrow (\neg rcv(delete,x,y) \wedge \neg rcv(delete,y,x)))\rangle$

Property 2 states that an active job P that has not sent delete messages must have informed its conflicting neighbors to include p in their conflict sets. Property 3 states that if both P and R are at **allocating** and are conflicting, then one of them must have sent insert message to the other; the message must have been received; and the process which sent the insert message must not have received any delete messages since then.

Theorem 1 *The algorithm satisfies non-triviality.*

Proof: We need to show that for any two jobs P, R,

$\Box\ (((P \text{ at } \textbf{allocating}) \wedge (R \text{ at } \textbf{allocating}) \wedge r \in N_p \wedge p \in N_r) \Rightarrow (p \in V_r \wedge r \in V_p)).$

Assuming the antecedent of the implication at any global state, we show that the consequent holds at the state. From Property 3, $\phi(P,R) \vee \phi(R,P)$, holds at s. For the rest of the proof assume w.l.o.g. that $\phi(P,R)$ holds. In other words, there exists an insert operation e such that

$(r \in N_p\ \mathcal{S}\ send(e,P,R)) \wedge (p \in N_r\ \mathcal{S}\ rcv(e,P,R)) \wedge$

$\boxminus\ (sent(e,p,r) \Rightarrow (\neg rcv(delete,p,r) \wedge \neg rcv(delete,r,p))).$ $\qquad \ldots (Q5)$

From Property $(B4)$,

$(\Diamond\!\!\!\!-\ (send(req(m),P,r) \wedge (P \text{ at } \textbf{detecting})) \ \wedge$

$\quad \Diamond\!\!\!\!-\ (\langle \exists n : Ack(m,n) : rcv(n,r,P)\rangle \wedge (P \text{ at } \textbf{detecting}))).$

Let R be the job receiving message m. Then, from Property (B5),

$(\Diamond\!\!\!\!-\ (send(req(m),P,R) \wedge (P \text{ at } \textbf{detecting})) \ \wedge$

$\quad \Diamond\!\!\!\!-\ (\langle \exists n : Ack(m,n) : rcv(n,R,P)\rangle \wedge (P \text{ at } \textbf{detecting}))).$

Consider the two cases based on the kind of the acknowledgment n.

Case 1: Message n is a $\langle req \rangle$ message. From the algorithm, P does not send any acknowledgments (in the form of a $\langle req \rangle$ or a $\langle no \rangle$ message) to r after global state $send(m,P,R)$. By Property $(B5)$, $\langle req \rangle$ message m serves as an acknowledgment to $\langle req \rangle$ message n. As a result, p is included in V_r at $rcv(m,P,R)$ and r is included in V_p at $rcv(n,R,P)$. Due to the absence of any delete operations, $r \in V_p \ \wedge \ p \in V_r$ holds currently.

Case 2: Message n is a $\langle no \rangle$ message. From the algorithm and handler $G2$, $rcv(m,P,R)$ is the same global state as $send(n,R,P)$, and $collecting_r$ is $false$ at $send(n,R,P)$. So, either R is yet to go past the first waiting statement at **detecting**, or R is at **allocating** at global state $send(n,R,P)$. We will show that both these cases are not possible by reaching a contradiction.

Case 2.1: In this case, R is yet to go past the first waiting statement at **detecting** at $send(n,R,P)$. By inspecting handler G2, $blocking_{rp}$ is set to true at $send(n,R,P)$. R waits until the variable is reset to $false$. Since R is at **allocating**, the variable is eventually reset either by an insert operation (handler G6) or by a $\langle release \rangle$ message (handler G5). Thus,

$\Diamond\!\!\!\!-\ (sent(n,R,P) \wedge (R \text{ at } \textbf{detecting}) \wedge (rcv(insert,p,R) \vee rcv(release,p,R))).$

Since the $\langle insert \rangle$ message e from P is received prior to m,

$\Diamond\!\!\!\!-\ (sent(n,R,P) \wedge (R \text{ at } \textbf{detecting}) \wedge rcv(release,p,R)).$

It is possible to show that any such $\langle release \rangle$ message that R receives is from P. So,

$\diamondminus\ (sent(n,R,P) \wedge (R \text{ at } \mathbf{detecting}) \wedge rcv(release,P,R))$.

Since $collecting_p$ is true at $send(m,P,R)$ and P is currently at **allocating**, P does not send any $\langle release\rangle$ messages to R after $send(m,P,R)$. Consequently, R does not receive any $\langle release\rangle$ messages from P after $rcv(m,P,R)$ or $send(n,R,P)$. This leads to a contradiction.

Case 2.2: In this case, R is at **allocating** at $send(n,R,P)$. Consider the state of R at global state $rcv(e,P,r)$; either R is yet to go past the first waiting statement at **detecting**, or has already done so.

Case 2.2.1: R is yet to go past the first waiting statement at **detecting** at $rcv(e,P,R)$. From Property $(Q5)$,

$p \in N_r\ \mathcal{S}\ (rcv(e,P,R))$.

From Property $(B4)$,

$(\diamondminus\ (send(req(m'),R,p) \wedge (R \text{ at } \mathbf{detecting}))\ \wedge$

$\diamondminus\ (\langle \exists n' : Ack(m',n') : rcv(n',p,R)\rangle \wedge (R \text{ at } \mathbf{detecting})))$.

It is possible to show that message m' is received by P. So, from Property $(B5)$,

$(\diamondminus\ (send(m',R,P) \wedge (R \text{ at } \mathbf{detecting}))\ \wedge$

$\diamondminus\ (\langle \exists n' : Ack(m',n') : rcv(n',P,R)\rangle \wedge (R \text{ at } \mathbf{detecting})))$.

Since m is the only $\langle req\rangle$ message that P sends to R, n' must be a $\langle no\rangle$ message. From the algorithm and handler $G2$, $rcv(m',R,P)$ is the same global state as $send(n',P,R)$, and variable $blocking_{pr}$ is set to $true$ at $rcv(m',R,P)$. P waits until the variable is reset to $false$. Since P is at **allocating**, the variable is eventually reset either by an insert operation (handler G6) or by a $\langle release\rangle$ message (handler G5). Thus,

$\diamondminus\ (sent(n',P,R) \wedge (P \text{ at } \mathbf{detecting}) \wedge (rcv(insert,r,P) \vee rcv(release,r,P)))$.

It is possible to show that no insert operations from r are executed at p after $send(n',P,R)$ and prior to the current state. So,

$\diamondminus\ (sent(n',P,R) \wedge (P \text{ at } \mathbf{detecting}) \wedge rcv(release,r,P))$.

It is possible to show that any such $\langle release\rangle$ message that P receives is from R. So,

$\diamondminus\ (sent(n',P,R) \wedge (P \text{ at } \mathbf{detecting}) \wedge rcv(release,R,P))$.

Since $collecting_r$ is true at $send(m',R,P)$ and R is currently at **allocating**, R does not send any $\langle release\rangle$ messages to P after $send(m',R,P)$. Consequently, P does not receive any $\langle release\rangle$ messages from R after $rcv(m',R,P)$ or $send(n',P,R)$. This leads to a contradiction.

Case 2.2.2: R has gone past the first waiting statement at **detecting** at $rcv(e,P,R)$. By the algorithm, p is added to $pending_r$ at $rcv(e,P,R)$, and a $\langle release\rangle$ message is not sent to P. Therefore,

$\boxminus\ (sent(e,P,R) \Rightarrow \neg rcv(release,r,P))$.

Since R has gone past the first waiting statement at **detecting** at $rcv(e,P,R)$,

$\boxminus\ (sent(e,P,R) \Rightarrow (\neg rcv(release,r,P) \wedge \neg(rcv,insert,r,P)))$.

Since variable $blocking_{pr}$ is initialized to true at $send(e,P,R)$, it is not falsified in handlers G5 and G6. So,

$\boxminus\ (sent(e,P,R) \Rightarrow \neg blocking_{pr})$.

Therefore, P cannot go past the first waiting statement at **detecting**, and consequently cannot be at **allocating** currently. This is a contradiction.

This completes the proof of the theorem. □

Theorem 2 *The algorithm satisfies validity.*

Proof: We need to show that for any job P and process r,
$(P$ at **allocating** and $r \in V_p) \Rightarrow$
$\langle \exists R :: (P$ at **allocating** $\wedge (R$ at **allocating** $\vee R$ at **releasing**$)) \Rightarrow p \in V_r) \;\; \mathcal{S}$
$(P$ at **detecting** $\wedge R$ at **detecting** $\wedge\; p \in N_r \wedge r \in N_p)\rangle$.

Assuming the antecedent at any global state, we show that the consequent holds at that state. From Property $(B1)$,

$\diamondsuit\, (collecting_p \wedge rcv(req, r, P))$.

Let m be this $\langle req \rangle$ message, and let R be the job that sent this message. From properties $(B2)$ and $(B3)$,

$\diamondsuit\, (send(m, R, P) \wedge p \in N_r \wedge (R$ at **detecting**$)) \;\wedge$
$\diamondsuit\, (collecting_p \wedge rcv(m, R, P) \wedge r \in V_p \wedge (P$ at **detecting**$))$. $\quad \ldots (Q6)$

Consider two cases depending on whether R has received an acknowledgment to message m.

Case 1: R has received an acknowledgment n. According to Property (B5), message n is sent by P. Message n cannot be a $\langle no \rangle$ message since such a $\langle no \rangle$ message can only be sent in response to message m and $collecting_p$ is true at $rcv(m, R, P)$ during the execution of handler G2. Therefore message n must be a $\langle req \rangle$ message. From the algorithm, P sent the message when crossing the first waiting statement at **detecting** and since n acknowledges m, R received n while $collecting_r$ is true. From properties $(B2)$ and $(B3)$,

$\diamondsuit\, (send(n, P, R) \wedge r \in N_p \wedge (P$ at **detecting**$)) \;\wedge$
$\diamondsuit\, (collecting_r \wedge rcv(n, P, R) \wedge p \in V_r \wedge (R$ at **detecting**$))$. $\quad \ldots (Q7)$

Since P is at **allocating**, it has not issued any delete operations. So,
$p \in V_r \;\mathcal{S}\; rcv(n, P, R)$. Therefore, R cannot cross the waiting statement at **releasing** in the user outline of Figure 3. As a result, R also has not issued any delete operations. Therefore,

$(P$ at **allocating** $\wedge (R$ at **allocating** $\vee R$ at **releasing**$)) \Rightarrow p \in V_r) \;\; \mathcal{S} \;\; rcv(n, P, R)$. $\ldots (Q8)$

Moreover, from properties $(Q6)$ and $(Q7)$,

$(r \in N_p \;\mathcal{S}\; (send(n, P, R) \wedge (P$ at **detecting**$))) \;\wedge\; \diamondsuit\, (rcv(n, P, R) \wedge (R$ at **detecting**$))$.

By a symmetric reasoning,

$(p \in N_r \;\mathcal{S}\; (send(m, R, P) \wedge (R$ at **detecting**$))) \;\wedge\; \diamondsuit\, (rcv(m, R, P) \wedge (P$ at **detecting**$))$.

The proof follows from combining the above two properties and (Q8).

Case 2: R has not received an acknowledgment to message m. Therefore,

$(R$ at **detecting**$) \;\mathcal{S}\; send(m, R, P)$.

Since P has not issued any delete operations, using Property $(Q6)$,

$((R$ at **detecting**$) \wedge p \in N_r) \;\mathcal{S}\; send(m, R, P)$.

Since R has not issued any delete operations, using Property 2,

$((R$ at **detecting**$) \wedge p \in N_r \wedge r \in N_p) \;\mathcal{S}\; send(m, R, P)$.

Since R is still at **detecting** in this case, the condition $(P$ at **allocating** $\wedge (R$ at **allocating** $\vee R$ at **releasing**$)) \Rightarrow p \in V_r)$ of *Validity* is trivially true. The proof follows from observing that $send(m, R, P)$ precedes $rcv(m, R, P)$ and that $(P$ at **detecting**$)$ at $rcv(m, R, P)$ from Property $(Q6)$.

□

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

Theorem 3 *If each job at* **allocating** *transits to* **releasing** *within T units of time then a job at* **detecting** *transits to* **allocating** *within $O(\delta * T)$ units of time.*

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

Consider a job P that transits to **detecting**. It can be blocked at the first waiting statement by a neighbor r if $blocking_{pr}$ is true. Now, $blocking_{pr}$ is set to true by a message from r only after r has gone past the first waiting statement at **detecting**. Since process r transits to **releasing** within $O(1) + T$ time units, and thereafter resets $blocking_{pr}$ to false by sending a $\langle release \rangle$ message, $blocking_{pr}$ 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. $\square$

5 A Compositional Solution to Dynamic Resource Allocation

In this section we use the idea of conflict detection to present our solution to the distributed resource allocation problem. In addition to conflict detection, the composite algorithm uses a $(\delta+1)$-*coloring* scheme and a dining philosophers solution that allocates the resources. Conflict detection is first used to isolate a static view of the conflict graph. The $(\delta + 1)$-*coloring* scheme is used to color the nodes of the views, and finally the dining philosophers algorithm is used to resolve conflicts. 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 is $T_{dining}(\nu, \tau)$ then the resulting response time for the distributed resource allocation algorithm is given by $O(1) * T_\delta(\nu) + O(1) * T_{dining}(\nu, \tau)$. 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 [10] with a dining philosophers algorithm in [8] has a response time of $O((\delta^2 + \log^* M)\nu + O(\delta^2)\tau)$ and a message complexity of $O(\delta^2 + \delta \log^* M)$.

Our resource allocation algorithm is shown in Figure 6. As mentioned earlier, it combines a $(\delta + 1)$-*coloring* algorithm with a dining philosophers algorithm based on node coloring. A process p computes a local view at **detecting** and then proceeds to **allocating**. The computation at this state is *null*. This ensures on account of the *conditional local progress* property of conflict detection that every process completes the computation of local view. At state **releasing**, a process first waits until all the processes in its local view have also progressed to **releasing**. Then a $(\delta + 1)$-*coloring* algorithm is executed on the static graph defined by the local views V_p. The $(\delta+1)$-*coloring* algorithm is invoked with a parameter $colors(N_p - V_p)$ which returns the set of colors currently used by the set of processes in $N_p - V_p$. When the coloring algorithm completes, a process p is colored

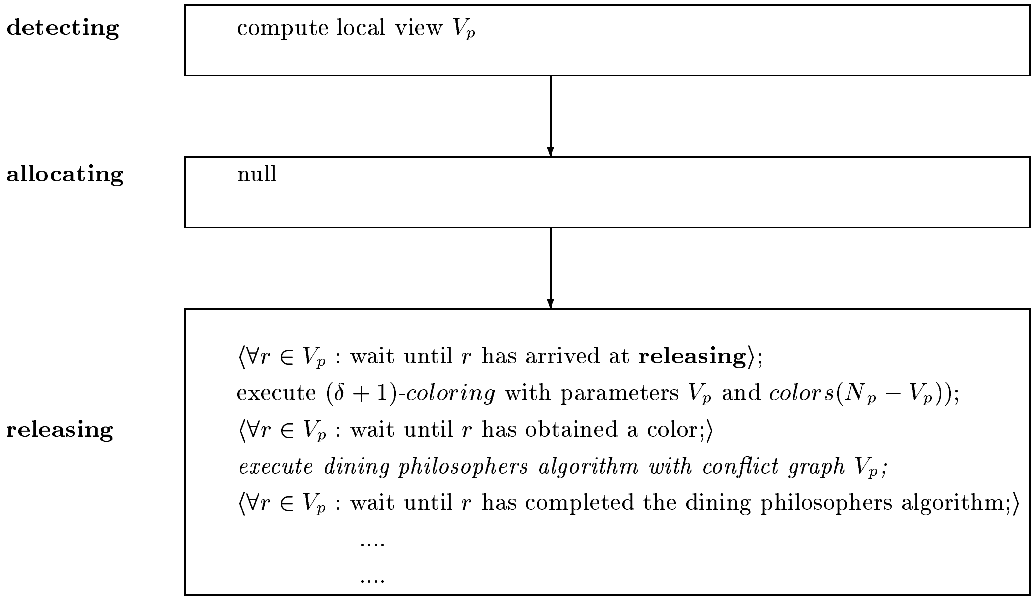

Figure 6: Composite algorithm for distributed resource allocation

with a color different from the colors of the processes in V_p and from colors in set $colors(N_p - V_p)$. Process p then waits for all processes in V_p to obtain a valid color. Then process p executes the dining philosophers algorithm with V_p as the underlying conflict graph. After that the process waits for all processes in V_p to complete the dining philosophers algorithm. Finally, it deletes itself from the neighborhood sets of all other processes in N_p. Then it resets N_p and V_p to empty and transits to **idle**.

Several barriers have been added at state **releasing**. The first barrier ensures that the node coloring is begun only after the neighboring processes are ready to participate. The second barrier ensures that the dining philosophers algorithm is begun only after the processed have obtained a valid color. The final barrier is for performance reasons. Without this barrier, a fast process may transit back to **idle** and return and compete with its slower neighbor who may still be at **detecting**. If that is the case, then the response time of the slower neighbor can become quite high.

5.1 $(\delta + 1)$-*coloring*

The $(\delta + 1)$-*coloring* algorithm used here is based on the algorithm by Goldberg, Plotkin, and Shannon [10]. However, our algorithm (shown in Figure 7) 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 P such that $|R| + |P| \leq \delta$. Parameter R is the set of neighbors of the node and P is a set of prohibited colors. A coloring is valid if neighboring nodes are not colored with the same color and a color prohibited in P is not used. A coloring is called $(\delta + 1)$-*coloring* if it uses colors in the range $[0..\delta]$.

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 7,

```
Procedure (δ + 1)-coloring(R,P)
-------------------------------
/* 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 P;
```

Figure 7: $(\delta + 1)$-coloring procedure for process p

process p (there is a 1-1 correspondence between processes and nodes) 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 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. The node with an incoming pointer is called the parent and accordingly, the other node is called a child. Since an edge points outward from a node in pseudo forest G_k only if the node assigns k to the edge, the out-degree of each node in a pseudo forest is at most one. (In other words, every node has at most parent.)

In the second phase of the algorithm, the 6-*coloring* algorithm described in [10] 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 [10] that a 6-*coloring* is obtained for each pseudo forest after the re-coloring procedure is repeated for $O(\log^* M)$ steps where M 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: the first stage reduces the number of colors to 5, the second stage reduces the number of colors to 4, and the last stage reduces the number of colors to 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 re-coloring 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 of (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 P are not chosen during this re-coloring procedure.

Theorem 4 *The $(\delta+1)$-coloring algorithm in Figure 7 has a time complexity of $O(\delta^2 + \log^* M)$ and a message complexity of $O(\delta^2 + \delta \log^* M)$.*

5.2 Correctness

We first show that the underlying $(\delta+1)$-*coloring* algorithm when combined with any view computation algorithm that satisfies the properties of *non-triviality*, *validity*, and *conditional local progress* conditions properly colors the dynamic conflict graph. Formally, we show the following two properties.

$(C1)$ For any two jobs P and R at **releasing** with computed colors, $r \in N_p \Rightarrow color_p \neq color_r$.

$(C2)$ For any job P with a maximum of δ_p conflicting neighbors and a computed color, $color_p \in [0..\delta_p]$.

For a proof of Property $(C1)$, let P and R be any two jobs with computed colors at **releasing** at a global state s and let $r \in N_p$ hold. It follows from Property 1 (of Section 2) that $p \in N_r$ holds at s. Consider the state intervals in which P and R are at **allocating**. If the intervals overlap then by *non-triviality*, $r \in V_p \;\wedge\; p \in V_r$ holds. As a result, P and R choose different colors during $(\delta+1)$-*coloring*. This completes the proof of $(C1)$ for the first case. Now, consider the other case in which the intervals do not overlap. Let P be the job that transits to **allocating** later and let u be the global state at which this transition occurs. Now, $r \in N_p \;\wedge\; p \in N_r$ holds in the interval $[u, s]$. Therefore, P will find R's color to be prohibited, and choose a different color. The proof follows.

Now consider the proof of Property $(C2)$. By the correctness of the $(\delta+1)$-*coloring* algorithm, it is sufficient to show that V_p is a subset of N_p when P transits to **allocating** at a global state v. Let r be any process in V_p at v. By *validity*, there exists a previous global state w at which r is at **detecting** and $r \in N_p \wedge p \in N_r$. The proof follows. □

Having established that the coloring of nodes is done properly, we focus our attention to the execution of the dining philosophers algorithm.

$(C3)$ For any two jobs P and R at **releasing** executing the dining philosophers algorithm,
$r \in N_p \Rightarrow (r \in V_p \wedge p \in V_r)$.

Proof: Let s be a global state at which jobs P and R at **releasing** executing the dining philosophers algorithm and let $r \in N_p$ hold at s. By Property 2, $p \in N_r$ also holds at s. Without any loss of generality, let P transit to **detecting** after R at a global state t. Due to the absence of any delete operations, $r \in N_p \;\wedge\; p \in N_r$ holds in the interval $[t, s]$. By non-triviality, $r \in V_p \;\wedge\; p \in V_r$ holds at s. □

Based on Property ($C3$), conflicting jobs at **releasing** include each other in their local views when executing the dining philosophers algorithm. Also by Property ($C1$), conflicting jobs have different colors. Consequently, by the correctness of the underlying dining philosophers algorithm, conflicting jobs do not enter critical section at the same time and mutual exclusion is satisfied.

Let T_δ be the response time of the $(\delta+1)$-*coloring* algorithm and let T_{dining} be the response time of the dining philosophers algorithm. The response time of the algorithm is given by the following theorem.

Theorem 5 *The response time of the composite resource allocation algorithm is $O(\delta+T_\delta+T_{dining})$ and the message complexity is $O(\delta)+M_\delta+M_{dining}$.*

Proof: The response time of the composite resource allocation algorithm is calculated by considering the time taken for each state transition. According to the algorithm, a process transits from **allocating** to **releasing** in $O(1)$ time. Therefore, by substituting $T = O(1)$ in Theorem 3, a process transits from **detecting** to **allocating** in $O(\delta)$ time. Each barrier in the **releasing** state introduces an extra $O(1)$ waiting time. Summing all these quantities and the response times of the $(\delta+1)$-*coloring* algorithm and the dining philosophers algorithm, the response time of the composite resource allocation algorithm is $O(\delta+T_\delta+T_{dining})$. The proof of the message complexity is similar. $\square$

From Theorem 5 and Property ($C2$), we have the following corollary.

Corollary 1 *The response time of the composite resource allocation algorithm is $O(\delta^2+\log^* M)$ and the message complexity is $O(\delta^2+\delta\log^* M)$ when the $(\delta+1)$-coloring algorithm in Figure 7 and the dining philosophers algorithm in [8] are used.*

Note that the resource allocation problem can also be solved by invoking $(\delta+1)$-*coloring* at **allocating** and the dining philosophers algorithm at **releasing**. However, the total response time then becomes $\delta *$ (response time of $(\delta+1)$-*coloring*) + response time of dining philosophers algorithm. This is because our conflict detection algorithm introduces a multiplicative factor of $O(\delta)$ on the time complexity of the algorithm placed at state **allocating**. The above response time is higher than what we obtain in Figure 6 by using a null computation at **allocating** and invoking both node coloring and dining philosophers algorithms at **releasing**.

6 Conclusion

The idea of computing views on a dynamic graph provides us with a new method for solving dynamic graph problems using solutions to static graph algorithms. In the case of distributed resource allocation, we designed an algorithm by combining existing solutions to $(\delta+1)$-*coloring* and dining philosophers. The result is an algorithm with a response time of $O(\delta^2+\log^* M)$ and a message complexity of $O(\delta^2+\delta\log^* M)$. Both of these complexity measures depend minimally on global parameters and improve the existing complexity measures by a factor of $\log M$.

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 view computation and finally settled on what appeared to be the simplest.

In the proposed model for the resource allocation problem, a job of a process corresponds to exactly one entry into the critical section. Thus, if a process wishes to access the same set of resources more than once, it has to submit a different job each time. This means that the process has to participate in the possibly expensive node coloring each time. It is possible to make an optimization to the algorithm that avoids the possibly redundant node colorings: a process that needs to access the same set of resources as in its last request can transit from **releasing** to **detecting** directly. In the extreme case, in which the requested set of resources remains fixed forever for each process, the resource allocation problem reduces to the dining philosophers problem. It should be noted however that avoiding a node coloring can have an adverse effect on the response time if the level of conflict has reduced considerably since the last node coloring. This is because the response time is directly proportional to the color of a node and a higher-than-needed node color may imply an unreasonably large response time.

References

[1] Arnold, Ken. and Gosling, James *The Java Programming Languages* Addison Wesley Pub. Co., Reading, Mass., 1996.

[2] Attiya, H., and Rachman, O. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing.* 27, 2 (April 1998), 319–340.

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

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

[5] Chandy. K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comp. Syst.*, Vol. 3, No. 1, February, 1985, pp. 63-75.

[6] Chandy. K. M., and Misra, J.. The drinking philosophers problem. *ACM Trans. on Prog. Lang. and Syst. 6*, 4 (Oct. 1984), 632–646.

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

[8] Choy. M., and Singh, A. K. Efficient fault tolerant algorithms for distributed resource allocation. *ACM Trans. Progr. Lang. and Syst. 17*, 3, (May 1995), 535–559.

[9] Dijkstra, E. W. Hierarchical ordering of sequential processes, *Acta Informatica 1*, 1971, 115–138.

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

[11] Johnson, K. L., Kaashoek, M, F., and Wallach, D. A. CRL: High-performance All-software Distributed Shared Memory. In *15th Symposium on Operating Systems Principles* 1995.

[12] Manna, Z., and Pnueli, A. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, New York, 1992.

[13] Rhee, Injong A modular algorithm for resource allocation. *Distributed Computing*, 11 (3), 1998, 157–168.

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

[15] Weidman, E. and Page, I. and Pervin, W. Explicit dynamic exclusion algorithm. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, 1991, 142–149.

Appendix A

Property 2 For any job P, $\Box\ ((\neg(P \text{ at } \mathbf{idle}) \wedge \neg\langle\exists(\text{delete } e) :: \mathit{sent}(e, P, -)\rangle) \Rightarrow (r \in N_p \Rightarrow p \in N_r))$.

Proof: Assume on the contrary that at a global state s where $(\neg(P \text{ at } \mathbf{idle}) \wedge \neg\langle\exists(\text{delete } e) :: \mathit{sent}(e, P, -)\rangle) \wedge r \in N_p \wedge p \notin N_r))$.
Since $r \in N_p$, according to the user outline in Figure 3, one of the following two cases hold.

Case 1: P obtained r in N_p while at **idle** and subsequently inserted p into N_r. Let t be the global state at which this insert operation is executed at r and let u be the global state at which it is completed at p.

Case 2: A job of process r obtained p in N_r and inserted r into N_p. Let t be the global state at which this insert operation is issued at r and let u be the global state at which it is executed at p.

Let R be the job of process r that is active at t. In both cases, by assumption

$(r \in N_p \text{ in } [u, s]) \wedge (p \in N_r \text{ at } t) \wedge (p \notin N_r \text{ at } s)$.

The last two conjuncts imply that p is removed from N_r in the interval (t, s). Since delete operations by previous jobs of process p complete prior to global state t and since P itself does not issue any delete operations until s, p is not removed from N_r on account of a delete operation from p. It remains to show that p is not removed from N_r by the resetting of N_r to empty by R.

Now, R resets N_r only after issuing and completing all its delete operations. In Case 1, any delete operations by R to process p are issued after global state t (based on our assumption about the non-interleaving of insert operations during a delete phase at **releasing**). In Case 2, R is at **idle** at global state t, and therefore any delete operations by R to process p are again issued after global state t. Therefore, in either case, $\langle\exists\ \mathit{delete}\ d ::\ \mathit{send}(d, R, P) \wedge\ \mathit{complete}(d, R)\rangle$ in (t, s), where $\mathit{complete}(d, R)$ denotes the state at which d is completed. According to the algorithm, $r \notin N_p$ holds at the state $y = rcv(d, R, P)$. Consequently, by FIFO assumption, $(r \notin N_p)$ at $y \in (u, s]$. This contradicts $(r \in N_p)$ in $[u, s]$.

This completes the proof. $\Box$

Property 3 For any two jobs P and R,
$\Box\ (((P \text{ at } \textbf{allocating}) \wedge (R \text{ at } \textbf{allocating}) \wedge r \in N_p \wedge p \in N_r) \Rightarrow (\phi(P, R) \vee \phi(R, P)))$,
where formula ϕ is defined as follows:

$$\phi(X, Y) \equiv \langle \exists (\text{insert } e) :: (y \in N_x\ \mathcal{S}\ send(e, X, Y)) \wedge (x \in N_y\ \mathcal{S}\ rcv(e, X, Y)) \ \wedge$$
$$\boxminus\ (sent(e, x, y) \Rightarrow (\neg rcv(delete, x, y) \wedge \neg rcv(delete, y, x)))\rangle$$

Proof: Assume $((P \text{ at } \textbf{allocating}) \wedge (R \text{ at } \textbf{allocating}) \wedge r \in N_p \wedge p \in N_r)$ holds at a global state s. Then either P obtained $r \in N_p$ at **idle** and issued an insert operation to add p to N_r, or R obtained $p \in N_r$ at **idle** and issued an insert operation to add r to N_p. Consider these two cases separately.

Case 1: In this case, P issued an insert operation to add p to N_r. Let t be the global state at which the operation is issued and u be the global state at which it is executed. Thus, $p \in N_r$ holds immediately after u. We consider two sub-cases ($p \in N_r$ in $(u, s]$) or not.

Case 1.1: ($p \in N_r$ in $(u, s]$). Therefore, the operation to insert p into N_r was executed by R. (If this was executed by a previous job of process r then N_r would have been reset to empty in the interval $(u, s]$.) We also have that $r \in N_p$ holds at t. Because R does not issue any delete operations prior to s and because previous jobs of process r purge their $\langle delete \rangle$ messages from the network prior to t, $r \in N_p$ holds in $(t, s]$. This also implies that no delete operations from p are executed at r in the interval $(t, s]$, and no delete operations from r are executed at p in the interval $(t, s]$. The proof obligation follows.

Case 1.2: $\neg(p \in N_r \text{ in } (u, s])$. Consider the global state v at which it is set to false for the last time. Since $p \in N_r$ holds at a later global state s, R must have obtained $p \in N_r$ at **idle** and issued an insert operation to add r to N_p. Let t' be the global state at which the operation is issued by R and u' be the global state at which it is executed by P. Thus, $r \in N_p$ holds after u', and continuously in the interval $(u', s]$. The rest of the proof is as in Case 1.1.

Case 2: This proof is symmetric to Case 1. $\Box$

Property 4 *For any job P and message m,*
$((P \textit{ at } \textbf{detecting} \ \vee P \textit{ at } \textbf{allocating}) \wedge r \in N_p) \textit{ at } send(m, P, r) \Rightarrow (p \in N_r \textit{ at } rcv(m, P, r))$

Proof: Assume the antecedent of the implication in the property holds for a job P and a message m. We show that $p \in N_r$ at state $t = rcv(m, P, r)$. By Property 2, we have $p \in N_r$ at state $s = send(m, P, r)$. Process p can be removed from N_r in the interval $(s, t]$ either by a delete operation from process p or by the resetting of N_r to empty by a job R of process r. The former is not possible because P has not issued any delete operations, and because previous jobs of process p would have purged their $\langle delete \rangle$ messages from the network, Next, we show that the latter is also not possible.

If R resets N_r to empty at a global state w in the interval $(s, t]$ then consider the global state u right before R begins its delete phase at **releasing**. On account of the non-interleaving of insert

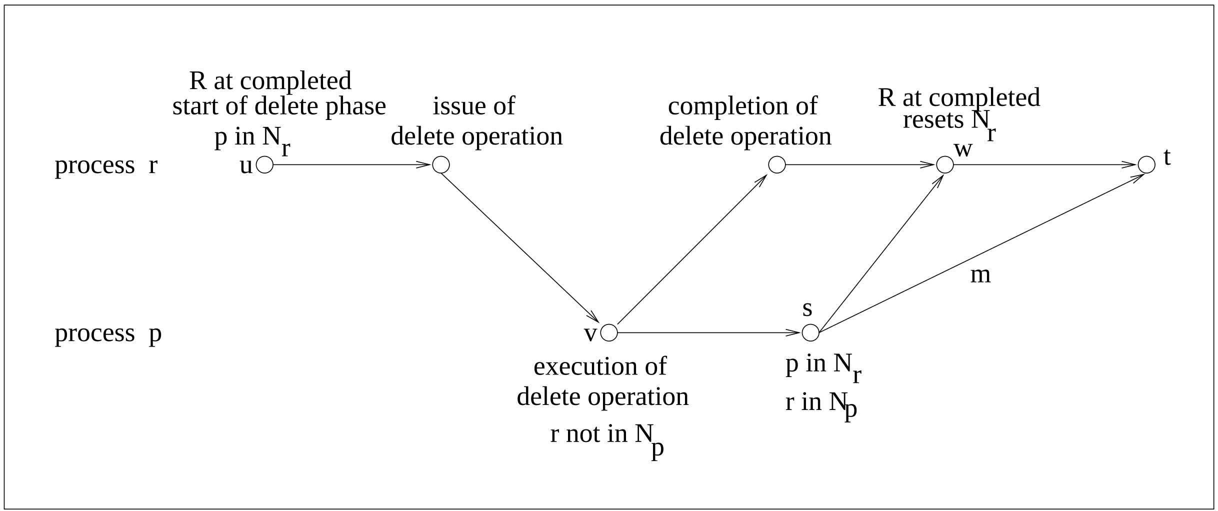

Figure 8: History constructed in Property 4

operations, we have $p \in N_r$ at u. Therefore, R issues and completes a delete operation to remove r from N_p in the interval $[u, t]$. In order for R to reset N_r prior to t, this delete operation must be performed at p at a global state v prior to s. The sequence of global states is shown in Figure 8. Since ($r \notin N_p$ at v) and ($r \in N_p$ at s), some job of process p must have obtained $r \in N_p$ in the interval $(v, s]$. Since P is at **detecting** or at **allocating** at s, process p must have issued and completed an insert operation to add itself to N_r in the interval $(v, s]$. But, this would require the interleaving of an insert operation at r between u and w, which is not possible. Therefore, R cannot reset N_r in the interval $(s, t]$.

This completes the proof of the property. $\square$

Appendix B

Proposition 1 *For any job P that has set $collecting_p$ to true, if P issued an insert operation e to add p to N_q then P receives exactly one $\langle release \rangle$ message from process q. Also, the $\langle release \rangle$ message is sent by the job of q that executes e.*

Proof: The proof is by induction. We prove that the proposition holds for a job P by assuming that it holds for all predecessors of P. Let s be a state where $collecting_p$ is true and t be the state where $rcv(insert(e), P, q)$ holds.

We first show that P receives at least one $\langle release \rangle$ message from process q. According to the algorithm, $(q \in N_p \ \wedge \ blocking_{pq})$ at $send(e, P, q)$ holds. Since any intervening delete operations issued by q do not change N_p, $q \in N_p$ holds at the completion of the insert operation. Since P is already past the first waiting statement at **detecting**, either $blocking_{pq}$ is set to false or q is removed from N_p in at a global state $g \in (t, s]$. Therefore, either $rcv(release, q, P)$ at g (handler G5), or q issues an insert operation e' such that P at **detecting** $\wedge$ $rcv(e', q, P)$ at g (handler G6), or q issues a delete operation d' such that P at **detecting** $\wedge$ $rcv(d', q, P)$ at g. The proof obligation follows in the first case. In the second case, e' must be issued by q after t, or else since P waits at **idle** for the completion of its insert operations, the insert operation issued by q will be executed at p while P is at **idle**. For a similar reason, in the third case, d' is issued by q after t. In both cases, since P does not receive a $\langle release \rangle$ message, a job Q of process q must have included p in $pending_q$ at t upon finding ($collecting_q \vee$ $state_q =$ **allocating**) to be true at t (handler G6). Since q issues e' or d' after t, Q must have transited to **releasing**, and according to handler G7, sent a $\langle release \rangle$ message to p. By the FIFO assumption, $rcv(release, Q, P)$ precedes g. Since $g \in (t, s]$, this concludes a proof of the fact that P receives at least one $\langle release \rangle$ message from process q.

We next show that P does not receive any $\langle release \rangle$ messages that are sent before t. Consider any such $\langle release \rangle$ message m, for which $send(m, q, P)$ at u and u precedes t. Message m is sent either when an insert operation issued by a job P' preceding P is executed at q (handler G6) or when a job Q' transits to **releasing** and finds $p \in pending_q$ at u. In the former case, by the induction hypothesis, P' receives $\langle release \rangle$ message m or a $\langle release \rangle$ message sent later by q. Consequently, due to the FIFO assumption, message m is not received by P. In the latter case, since $p \in pending_q$ at u, according to handler G6, an insert operation issued by a job P' preceding P must have been executed at q while ($collecting_q \vee state_q =$ **allocating**) held. By the induction hypothesis, P' receives m and therefore, P does not receive m.

We conclude that P receives at least one $\langle release \rangle$ message from process q and any such $\langle release \rangle$ message is sent after t. We now show that exactly one $\langle release \rangle$ message is received by P as stated in the proposition. Consider the first $\langle release \rangle$ message received by P. It is sent by q either in handler G6 or in handler G7. Consider these two cases separately.

In the former case, q sends the first $\langle release \rangle$ message to P at state t and p is not added to $pending_q$ due to the insert operation by P. Since set $pending_q$ can be non-empty only while the predicate ($collecting_q \vee state_q =$ **allocating**) holds, $pending_q$ is empty at t. On account of handlers G6 and G7, it follows that no further $\langle release \rangle$ messages are sent to P by process q.

In the latter case, process q adds p to $pending_q$ due to the insert operation by P and sends a $\langle release \rangle$ message to P upon transiting to **releasing**. On account of handlers G6 and G7, this is the only $\langle release \rangle$ message that P will receive from q. □

Proposition 2 *For any active job P at a global state satisfying ($collecting_p \vee state_p =$ **allocating**), V_p is a subset of $temp_p$, the copy of N_p when P set $collecting_p$ to true.*

Proof: Consider a global state at which job P is active and ($collecting_p \vee state_p =$ **allocating**) holds. Let $q \in V_p$ at this global state. We need to show that $q \in temp_p$. According to handler G2, P must have received a $\langle req \rangle$ message m from a job Q of process q while $collecting_p$ is $true$. Let s be global state at which Q sends m and t be the global state at which P receives m. Note that P sets $collecting_p$ to $true$ before t. According to the algorithm, $p \in N_q \wedge$ $collecting_q = true$ at s. Consider the sequence of events as a result of which p belongs to N_q at s. Either process p issues an insert operation to add p to N_q, or p does not issue such an operation and Q obtains $p \in N_q$ while at **idle**. Consider these two cases separately.

Case 1: process p issues an insert operation to add p to N_q. Let u be the global state at which the insert operation is executed at q and let v be the global state at which this insert operation is completed at p. By assumption, $p \in N_q$ in $(u, s]$. Job Q must be active at u or otherwise N_q would have been reset to empty (implying $p \notin N_q$ in $(u, s]$). Since Q is at **detecting** at s, it does not issue any delete operations that are executed at p in the interval $(v, t]$. Also, a job preceding Q must have completed all its delete operations at p before v. Therefore, $q \in N_p$ holds at v and no delete operations are executed at p in the interval $(v, t]$. Now, the job issuing the insert operation adding p to N_q is either P or a job P' preceding P. In the former case, since P does not issue any delete operations until t, we have $q \in N_p$ in $(v, t]$ and in particular, $q \in temp_p$. In the latter case, since P is active at t, P' must have issued and completed a delete operation at q in the interval $(v, t]$. This delete operation must have been executed at q in the interval $(u, s]$, contradicting the assumption that $p \in N_q$ at s.

Case 2: p does not issue an insert operation to add p to N_q and Q obtains $p \in N_q$ while at **idle**. According to the user outline, Q issues an insert operation to add q to N_p. Let u be the global state at which the insert operation is issued at q and let v be the global state at which the insert operation is executed at p. By assumption, $blocking_{qp}$ is $true$ immediately after u and $p \in N_q$ in $(u, s]$ holds. Job P sets $collecting_p$ to $true$ either after or before v.

In the former case, q is inserted to N_p at v. We show that $q \in N_p$ in $(v, t]$, and hence $q \in temp_p$. Since Q issued an insert operation at u and message m is received at t, process q does not delete q from N_p in the interval $(v, t]$. If a job P' preceding P resets N_p to empty in the interval $(v, t]$ then since an insert operation is executed at v and since the execution of insert operations does not interleave with the issuance and completion of delete operations, P' also issues and completes a delete operation to remove p from N_q in the interval $(v, t]$ before resetting N_p to empty. Consequently, p is removed from N_q in the interval $(u, s]$, contradicting the assumption that $p \in N_q$ in $(u, s]$ holds. Therefore, q is not removed from N_p in the interval $(v, t]$ and $q \in N_p$ in $(v, t]$ holds.

In the latter case, $collecting_p$ is true at v and according to handler G6, q is inserted into $pending_p$. Consequently, P does not send any $\langle release \rangle$ messages to Q until it transits to **releasing** (handlers G6 and G7). In particular, P does not send $\langle release \rangle$ message to Q before s. Since $collecting_q$ is $true$ at s, by Proposition 1, any $\langle release \rangle$ messages that Q receives from p is sent by P. This means that Q does not receive any $\langle release \rangle$ messages from p before state s. Variable $blocking_{qp}$ which is set to true at u can be reset to false either on account of an insert operation by p (handler G6) or on account of receiving a $\langle release \rangle$ message from p (handler G5). By assumption, no insert operations issued by p are executed at q in the interval $(u, s]$. As a result, $blocking_{qp}$ remains $true$ in the interval $(u, s]$ and Q cannot go past the first waiting statement at **detecting** and send the $\langle req \rangle$ message m. This is a contradiction.

The proof of the proposition follows. □

Property (B5) *For any $\langle req \rangle$ message m sent from process p to r that is received by job R, if $FirstAck(m, n)$ then n is sent by R.*

Proof: The proof is by induction. We prove that the property holds for a $\langle req \rangle$ message m sent from a job P to a job Q by assuming that it holds for all $\langle req \rangle$ messages that are sent prior to m. We need to show that the first acknowledgment to m, call it m_1, is either a $\langle req \rangle$ message sent by Q or a $\langle no \rangle$ message sent by Q in response to m.

We first show that if m_1 is a $\langle no \rangle$ message then it is not sent in response to some other $\langle req \rangle$ message besides m. The proof is by contradiction. Assume that message m_1 is sent in response to a $\langle req \rangle$ message m_2 different from m. Since each job sends at most one $\langle req \rangle$ message to a process, message m_2 must have been sent by a job P' preceding P. Job P' must have received an acknowledgment to m_2 and by the induction hypothesis, the first acknowledgment is either a $\langle req \rangle$ message m_3 sent by process q or a $\langle no \rangle$ message sent in response to m_2. The latter is not possible as the $\langle no \rangle$ message m_1 sent in response to m_2 is received by P. Therefore, the first acknowledgment of m_2 must be m_3. By definition of acknowledgment to $\langle req \rangle$ messages, when P' receives m_3, $collecting_p$ is *true* and a $\langle no \rangle$ message is not sent in response to m_3 (handler G2). By the induction hypothesis, message m_3 must have been 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 generate a $\langle no \rangle$ message m_1, a contradiction.

Next, we show that if m_1 is a $\langle req \rangle$ message then it is sent by Q. For this we show that message m_1 is not sent by a job Q' preceding Q or by a job Q'' succeeding Q. We first show that message m_1 is not sent by a job Q' preceding Q. Let s be the state $send(req(m_1), Q', P)$, t be the state $rcv(m_1, Q', P)$, and u be the state $rcv(m, P, Q)$. By the algorithm, $q \in N_p$ holds at s and by Property 2, $p \in N_q$ holds at s. By the user outline, either process p issues a delete operation that is completed at q in the interval $(s, u]$ or Q' issues and completes a delete operation at p in the interval $(s, u]$. In the former case, the delete operation must be issued by a predecessor of P and by the FIFO assumption, cannot complete at p before t, contradicting the assumption that P is active at t. In the latter case, by the FIFO assumption, the delete operation cannot complete at q at or before u, contradicting the assumption that Q is active at u. Therefore, message m_1 is not sent by a job preceding Q.

Finally, we show that message m_1 is not sent by a job Q'' succeeding Q. In this case, while P is at **detecting** and is waiting for an acknowledgment, job Q is terminated and a succeeding job Q'' transits to **detecting**. Since Q'' sends message m_1, it must have obtained $p \in N_q$ and initialized $blocking_{qp}$ to *true* when it transited to **detecting**. Since P does not send any $\langle release \rangle$ message while $collecting_p$ is *true* (handlers G6 and G7), Q'' cannot go past the first waiting statement of **detecting** and send message m_1, a contradiction.

In summary, if m_1 is a $\langle req \rangle$ message then it is sent by Q. This concludes the proof of the property. □

Property (B6) *Each $\langle req \rangle$ message is acknowledged within 2ν time units and no $\langle req \rangle$ message is acknowledged more than once.*

Proof: The proof is by induction. We prove that the property holds for a $\langle req \rangle$ message m sent from a job P to a job Q by assuming that it holds for all $\langle req \rangle$ messages that are sent prior to m.

We first show that an acknowledgment for m is received within 2ν time units. If variable $collecting_q$ is $false$ when Q receives m, then an acknowledgment in the form of a $\langle no \rangle$ message is returned to P (handler G2) within 2ν time units. Therefore, it suffices to show that either P receives a $\langle req \rangle$ message within 2ν time units or variable $collecting_q$ is $false$ when Q receives m. The proof is by contradiction. Assume that P does not receive a $\langle req \rangle$ message within 2ν time units and $collecting_q$ is *true* when Q receives m. According to handler G2, Q includes p in V_q. By Proposition 2, Q sends a $\langle req \rangle$ message m_1 to p when it sets $collecting_q$ to *true*. Message m serves as an acknowledgment to m_1. Consider the value of $collecting_p$ when m_1 is received. If m_1 is received while $collecting_p$ is false then it is received before m is sent. According to handler G2, a $\langle no \rangle$ message is sent to q. By the FIFO assumption, this $\langle no \rangle$ message is received at q while *collecting*$_q$ is $true$. Therefore, Q receives at least two acknowledgments for message m_1 (message m and the $\langle no \rangle$ message) and m_1 does not satisfy the induction hypothesis, a contradiction. Now, consider the other possibility in which m_1 is received while *collecting*$_p$ is $true$. If m_1 is received by a predecessor of P then, by Property (B5), this predecessor sends the first acknowledgment to Q. Since m also serves as an acknowledgment to m_1, message m_1 does not satisfy the induction hypothesis, a contradiction. If message m_1 is received by P then by assumption it is not received within 2ν time units of transmission of m. This implies that m_1 is sent more than ν units after m is sent and consequently, m is received by Q before m_1 is sent. This contradicts the assumption that *collecting*$_q$ is *true* when Q receives m.

Next, we show that $\langle req \rangle$ message m is not acknowledged more than once. By Property (B5), the first acknowledgment to m is either a $\langle req \rangle$ message m_2 sent by Q or a $\langle no \rangle$ message sent in response to m. Consider the two cases separately. In the first case, message m is received after Q sets *collecting*$_q$ to *true*, or otherwise the ensuing $\langle no \rangle$ message would reach P before message m_2 and serve as the first acknowledgment. Therefore, message m_2 has at least one acknowledgment. Since m_2 is an acknowledgment to m, a $\langle no \rangle$ message is not sent in response to m_2. Therefore, by Property (B5), the first acknowledgment of m_2 is a $\langle req \rangle$ message sent by P. In other words, the first acknowledgment of m_2 is message m. Consequently, message m does not generate a $\langle no \rangle$ message. Since *collecting*$_q$ is *true* until Q receives m and a job sends at most one $\langle req \rangle$ message to another job, Q does not send any other acknowledgments to m besides m_2. Any successor of Q that obtains $p \in N_q$ will be blocked at the first waiting statement of *trying* until P transits to **releasing**. Therefore, P does not receive any other acknowledgment to m from successors of Q either.

Now consider the second case in which the first acknowledgment of m is a $\langle no \rangle$ message sent in response to m. Let r be the global state at which Q receives m. According to handler G2, *collecting*$_q$ is $false$ at r. If Q is at **allocating** or at **releasing** at r then it does not send any further acknowledgments to m. Otherwise, Q is at **idle** or at **detecting** and has not yet set *collecting*$_q$ to *true*. Since $q \in N_p$ holds prior to the transmission of m, by Property 4, $p \in N_q$ holds at r. Since *blocking*$_{qp}$ is set to *true* at r (handler G2), Q cannot go past the first waiting statement at **detecting** unless P transits to **releasing**. Since P sends at most one $\langle req \rangle$ message to Q and Q does not send any $\langle req \rangle$ messages until crossing the first waiting statement at **detecting**, P does not receive any other $\langle req \rangle$ or $\langle no \rangle$ messages from q until P transits to **releasing**. In other words, P does not receive any other acknowledgments to m. This completes the proof of the property. □