A distributed resource allocation algorithm for many processes

Wim H. Hesselink (whh469b)
University of Groningen, The Netherlands
w.h.hesselink@rug.nl

June 27, 2013

Abstract

Resource allocation is the problem that a process may enter a critical section *CS* of its code only when its resource requirements are not in conflict with those of other processes in their critical sections. For each execution of *CS*, these requirements are given anew. In the resource requirements, levels can be distinguished, such as e.g. read access or write access. We allow unboundedly many processes that communicate by reliable asynchronous messages and have finite memory. A simple starvation-free solution is presented. Processes only wait for one another when they have conflicting resource requirements. The correctness of the solution is argued with invariants and temporal logic. It has been verified with the proof assistant PVS.

Key words: distributed algorithms; resource allocation; drinking philosophers; readers/writers problem; verification; starvation freedom; fairness

1 Introduction

Resource allocation is a problem that goes back to Dijkstra's dining philosophers [8] and the drinking philosophers of Chandi and Misra [4]. It is the problem that a process may enter a critical section of its code only when its resource requirements are not in conflict with those of other processes in their critical sections. In the case of the dining philosophers, the philosophers form a ring and the resource requirements are two forks shared with the neighbours in the ring. In the drinking philosophers problem the philosophers are the nodes of an arbitrary finite undirected graph and each edge is assigned a set of bottles. When a philosopher gets thirsty, it chooses as resource requirement a set of bottles on its incident edges.

In the general resource allocation problem, there is a network of active user processes and passive resources, e.g., memory pages or data bases. For brevity, we often abbreviate "user process" to "process". From time to time a process may need access to a set of resources, e.g., to read data from the resources, or to write and synchronize the data on a set of resources. In the first case, it needs the guarantee that the data are not concurrently modified. In the second case, it needs exclusive access to the resources of the set.

For each access of a process, this set of requirements, which consists of access rights to resources, is called the *job*. The actual access to these resources is called the *execution* of the job or the *critical section* (*CS*). Jobs of different processes are called *compatible* when concurrent execution is allowed, and *conflicting* when it is not. A process with a job is called *competing*, otherwise it is called *idle*. When it is idle, it may choose a job and become competing.

The problem is to design a communication protocol for the processes such that every process with a job can eventually execute it and become idle again. The safety property required is *partial mutual exclusion*: conflicting jobs must never be executed concurrently. The progress property required is that unnecessary waiting is avoided. This has two main aspects: starvation freedom and concurrency. Informally speaking, *starvation freedom* [9], also called lockout freedom [19], means that, when it is always the case that every competing process eventually does a step, every competing process will eventually execute its job and become idle again. *Concurrency* [4, 23] means that every competing process that does steps will eventually execute its job unless it comes in eternal conflict with a process that does no steps.

1.1 Setting and sketch of solution

We present a solution for a setting with finitely many resources and unboundedly many processes. The processes have extendable private memory and communicate by asynchronous messages. The processes and messages are assumed to be reliable: the processes never crash, the messages are guaranteed to arrive and be handled, but the delay is unknown. Messages are not lost, damaged, or duplicated. They can pass each other, however, unlike in [3, 19] where the messages in transit from one sender to one receiver are treated first-in-first-out. Every process can send messages to every other process, and receive messages from it.

We accommodate unboundedly many processes by presenting code for infinitely many processes, which are initially all idle, and by using interleaving semantics. An execution of the system is an infinite sequence of states such that every subsequent pair of states is related by a step of one of the processes or by equality in case of a skip step. Infinitely many consecutive skip steps will be ruled out by fairness assumptions. An execution may contain steps of infinitely many processes, but at every moment only finitely many have done steps. It follows that the set of not idle processes is always finite. Similarly, there are always only finitely many messages in transit.

In our solution, processes with a job are informed about potential competitors in the following way. The resources are distributed over finitely many *sites* which keep finite lists of processes registered for the resources. When a process chooses a job, it registers at some sites for the resources it needs, and receive from these sites finite sets of processes that it needs to communicate with before it can enter *CS*. The union of these sets is finite. It is called the current *neighbourhood* of the process. The first part of the protocol thus determines the neighbourhood of the process, it makes this set finite and hopefully small. This part is called the *registration algorithm*.

In the second part of the protocol, called the *central algorithm*, every process with a job contacts the processes in its neighbourhood and proceeds to *CS*, but waits if that is necessary to maintain partial mutual exclusion. We assume that every process has a process identifier, a natural number that characterizes the process uniquely. In the central algorithm, these numbers are used for tie breaking.

The protocol has two kinds of waiting conditions: waiting for messages in transit to arrive, and waiting for conflicting processes to proceed (in the second case, the process is waiting for a message that has not yet been sent). It is not our aim to minimize the waiting time. We offer a simple solution with as much nondeterminism as possible and as much progress as we can accommodate in view of the safety requirement.

The proofs of the safety and liveness properties of the protocol have been carried out with the interactive proof assistant PVS [21]. The descriptions of proofs closely follow our PVS proof scripts, which can be found on our web site [12]. It is our intention that the paper can be read independently, but the proofs require so many case distinctions that manual verification is problematic.

In some applications of the protocol, for every resource there is only a small set of processes that may ever need the resource. In this case one can decide to keep the neighbourhoods constant, and remove the registration algorithm and the sites. Then the protocol reduces to the central algorithm, and has similar functionality as the drinking philosophers [4].

1.2 Overview

Section 2 contains preparatory material: our programming notations and their semantics, and the model for the jobs and their compatibility. Section 3 presents the specification and the algorithm. Section 4 contains the proof that the algorithm satisfies partial mutual exclusion. In Section 5, we introduce a second safety property: absence of localized deadlock, and prove that the algorithm satisfies this. In Section 6, we introduce and formalize progress, in a form that combines starvation freedom and concurrency. We discuss message complexity and waiting times in Section 7. We briefly describe related research in Section 8, and conclude in Section 9.

2 Preparation of the Algorithm

Notations for concurrent programs and their semantics are introduced in Section 2.1. In Section 2.2, we give the syntax for message passing. Section 2.3 describes the semantics of the messages. In Section 2.4, we develop a model for the jobs and their compatibility.

2.1 Semantics, variables, and guarded commands

Every process has a private state space spanned by the private variables. If v is a private variable of a process, we write v for it in the code for the process, but elsewhere we write $v.p$ for the value of v of process p.

We model the algorithm as a transition system with as (global) state space the Cartesian product of the private state spaces augmented with the collection of messages in transit. If we need to emphasize the (global) state x of the system, the value of private variable v of process p in state x is denoted by $x.v.p$. In section 2.3, we explain how the messages in transit are modelled (by shared variables).

The transitions of the algorithm are presented as guarded commands of the form

$$[\!] \quad B \quad \to \quad S \; .$$

with B the enabling condition and S the command to be executed, compare [20]. Every transition is executed by a single process, often called p, which can only inspect and modify its own private variables, receive messages, and send messages to known destinations (a destination is known to p if it is a site, or a process with a number occurring in a message that p has received). The transition can be regarded as an atomic step because actions on private variables give no interference, the messages are asynchronous, and any delay in sending a message can be regarded as a delay in message delivery.

Each process has a program counter pc that is formally just a private variable of type $\mathbb{N}$. For the sake of conciseness, we use a labelled command $k: \; S$ as an abbreviation of the guarded command

$$[\!] \quad pc = k \quad \to \quad pc := k + 1 \; ; \; S \; .$$

In command S, we may use **goto** ℓ as syntactic sugar for $pc := \ell$.

A labelled command of the form

$$k: \quad \textbf{await } C \ ; \ T \ .$$

where C is a condition, is an abbreviation of the guarded command

$$[\!] \quad pc = k \ \wedge \ C \quad \rightarrow \quad pc := k + 1 \ ; \ T \ .$$

The values of pc are called *line numbers* or *locations*. For a process p and a line number ℓ, we write p **at** ℓ to express $pc.p = \ell$. If L is a set of line numbers, we write p **in** L to express $pc.p \in L$.

2.2 Asynchronous messages

Every message has a message key (the identifier used in the algorithm, see section 3.2), a sender, and a unique destination. It may have a value. As in CSP [13], we write $m.q.r\,!$ for the command for process q to send a message with key m to process r, and $m.q.r\,?$ for the command for process r to receive this message. Unlike CSP, the messages are asynchronous. We write $m.q.r\,!\,v$ for the command for process q to send a message with key m and value v to r, and $m.q.r\,?\,v$ for the command for process r to receive message m from q and assign its value to the private variable v.

In the algorithm, for every message key m, and for every source q and destination r, we shall prove that there is never more than one message with key m in transit from q to r. Therefore, e.g., in Promela, the language of the model checker Spin [14], one could model the messages by channels with buffer size 1.

For the correctness of the algorithm, the time needed for message transfer can be unbounded. For other issues, however, it is convenient to postulate an upper bound Δ for the time needed to execute an atomic command plus the time that the messages sent in this command are in transit. Similarly, when discussing progress, we assume that the execution time of the critical sections is bounded by Γ.

2.3 Modelling of the messages

We need to formalize the messages and their transitions by describing the state changes induced by sending and receiving messages. This is done as follows. For a message with key m of type `void`, we use $m.q.r$ as an integer shared variable that holds the number of messages with key m in transit from q to r, to be inspected and modified only by the processes q and r.

A command $m.q.r\,!$ for process q to send message m to r corresponds to an incrementation of $m.q.r$ by one, which can be denoted $m.q.r$`++` . A command $(m.q.r\,?\ ;\ S)$ for process r to receive message m from q followed by command S, corresponds to a guarded command in which $m.q.r$ is decremented by one when positive, followed by S:

$$[\!] \quad m.q.r > 0 \quad \rightarrow \quad m.q.r\texttt{--} \ ; \ S \ .$$

The value of $m.q.r$ can be any natural number, but in our algorithm we preserve the invariants $m.q.r \leq 1$.

For a message that holds a value v of some type T, the above way of modelling cannot be used. In principle, we should model such messages by means of bags (i.e. multisets) of messages from sender to destination. In the algorithm, however, there is never more than one message with key m in transit from q to r. For simplicity, therefore, we model such a “channel” with key m from q to r as a shared variable $m.q.r$ of type $T \cup \{\bot\}$, which equals v when there is a message with key m in transit from q to r with value v, and which equals $\bot$ otherwise.

A command $m.q.r\,!\,v$ for process q to send value v via m to r is modelled as $m.q.r := v$. The command $(m.q.r\ ?\ v\ ;\ S)$ for process r to receive a message with key m from q and assign the value to v, followed by command S, is modelled by

$$\begin{array}{ll} [\!] & m.q.r \neq \bot \quad \rightarrow \\ & v.r := m.q.r\ ;\ m.q.r := \bot\ ;\ S\ . \end{array}$$

Initially, $m.q.r = \bot$ for all q and r.

As we model the bag by a single variable, we need to make sure that the bag has never more than one element. In other words, this way of modelling gives us the proof obligation that $m.q.r$ is sent only under the precondition $m.q.r = \bot$.

2.4 The job model

We introduce a type *Job* to hold the jobs of the processes. For jobs u, v, we write $u{*}v$ to express that the jobs u and v are compatible (i.e., not conflicting). The value $\mathit{none} : \mathit{Job}$ represents the empty job (absence of resource requirements). We assume that compatibility satisfies the axioms that $u * \mathit{none} \equiv \mathit{true}$ and that $u * v \equiv v * u$ for all jobs u and v.

In the simple case that the processes always need exclusive access to the resources they need, one may regard every job as a set of resources, and define jobs to be compatible if and only if these sets are disjoint. We then have $u * v \equiv (u \cap v = \emptyset)$ and $\mathit{none} = \emptyset$.

If one wants to distinguish read requests from write requests, however, one needs a more complicated job model. In this case, one could model a job as a pair of sets of resources, say (r, w) where r is the set of the resources for read access and w the set of resources for write access. Jobs (r_1, w_1) and (r_2, w_2) are then compatible if and only if the intersections $w_1 \cap w_2$, $w_1 \cap r_2$, and $r_1 \cap w_2$ are empty. One can also propose compatibility relations with more than two permission levels, where “shallow” access (e.g. reading of metadata) is allowed concurrently with “innocent” writing.

We therefore use a flexible job model that allows an arbitrary number $K \geq 1$ of levels. Let $\mathit{upto}(K)$ be the set $\{i \in \mathbb{N} \mid i \leq K\}$. Let *Rsc* be the finite set of the resources. We define a job to be a function $\mathit{Rsc} \rightarrow \mathit{upto}(K)$, and define compatibility of jobs u and v by requiring that $u + v$ is at most K:

$$(0) \qquad u * v \quad \equiv \quad (\forall\, c \in \mathit{Rsc} : u(c) + v(c) \leq K)\ .$$

In this way, relation $*$ is indeed symmetric, and the job *none* given by $\mathit{none}(c) = 0$ for all c is compatible with all jobs.

The simple job model is the case with $K = 1$. We take $K = 2$ for the readers/writers problem. Read access at resource c requires $u(c) \geq 1$, write access requires $u(c) = 2$. In this way, concurrent reading is allowed, while writing needs exclusive access.

3 The Algorithm

Section 3.1 contains the specification of the algorithm. The implementation begins in Section 3.2 with an abstraction function from implementation to specification, and the split between the central algorithm and the registration algorithm.

The central algorithm is sketched in Section 3.3. Section 3.4 contains the code of the central algorithm and discusses some global aspects. In Section 3.5, we present its design as a layered algorithm. Section 3.6 contains commands to abort the entry protocol. The registration algorithm is presented in Section 3.7. As the registration algorithm only allows the registration level of a process to grow, we give in Section 3.8 the processes the option to lower it again.

The entire algorithm is presented in this section without verification. The verification is postponed to Section 4. Yet, we designed the algorithm concurrently with the verification, because that is for us the only way to obtain a reliable algorithm. We separate the two aspects here for the ease of reading.

3.1 Specification

In the specification, every user process p has two private variables $\mathit{job}.p : \mathit{Job}$ and $\mathit{st}.p : \mathit{Status}$. See section 2.4 for the type *Job*. The type *Status* is given by

$$\mathit{Status} = \{\mathit{idle}, \mathit{entry}, \mathit{CS}, \mathit{exit}, \mathit{abort}\} \ .$$

We use *Proc* for the (possibly infinite) set of process identifiers. As the processes are characterized by natural numbers, we take *Proc* to be an arbitrary subset of $\mathbb{N}$. The abstract state space Y consists of pairs of functions:

$$Y = [\#\ \mathit{job} : \mathit{Proc} \to \mathit{Job}\, ,\ \mathit{st} : \mathit{Proc} \to \mathit{Status}\ \#] \ .$$

Here, we use the record type constructor $[\#\ ,\ \#]$ of PVS. Given a global state y, its component *job* is a function from processes to jobs. Therefore, $\mathit{job}.p$ is the current job of process p. In other words, $\mathit{job}.p$ can be regarded as a private variable of process p. In the code of process p, we write *job* instead of $\mathit{job}.p$.

When we need to be explicit about the abstract state $y \in Y$, we write $y.\mathit{job}.p$ and $y.\mathit{st}.p$ for the job and the status of process p in state y. The initial state y_0 has $y_0.\mathit{job}.p = \mathit{none}$ and $y_0.\mathit{st}.p = \mathit{idle}$ for all process identifiers $p \in \mathit{Proc}$.

A user process has two aspects: a *client* and a *server*. As a client, it repeatedly may choose a job with the purpose to execute it. As a server, it participates in a protocol. For us, the clients are given and the servers are to be implemented. Each client only communicates with its own server. In the implementation, the servers need to communicate with each other in order to ensure that conflicting jobs are never executed concurrently, while unnecessary waiting is avoided.

For every process p, the client and the server share the private variables $\mathit{job}.p$ and $\mathit{st}.p$. The client of p can choose a job when the process is idle. It executes the critical section when allowed, and then enables the exit protocol. If the entry protocol takes "too long", the client can signal the server to *abort* the entry protocol. This is sketched in the transition diagram below, with the client performing the vertical transitions (c) and the server performing the horizontal transitions (s):

$$\begin{tikzcd}
& \mathit{entry} \arrow[rr, "s"'] \arrow[dl, "c"'] & & \mathit{CS} \arrow[dr, "c"] & \\
\mathit{abort} \arrow[rr, "s"] & & \mathit{idle} \arrow[ul, "c"'] & & \mathit{exit} \arrow[ll, "s"']
\end{tikzcd}$$

The possibilities are formalized in the following guarded commands, where we have included the modifications of *job*.

```
**client**($p$) :
[] $\mathit{st} = \mathit{idle} \ \to$ **choose** $\mathit{job} \neq \mathit{none}$ ; $\mathit{st} := \mathit{entry}$ .
[] $\mathit{st} = \mathit{CS} \ \to \ \mathit{st} := \mathit{exit}$ .
[] $\mathit{st} = \mathit{entry} \ \to \ \mathit{st} := \mathit{abort}$ .
**end client** .
```

The server of p inspects and modifies the variables *job* and *st* of p as follows:

```
server(p) :
[] $st = entry \;\to\; st := CS$ .
[] $st = exit \;\to\; job := none\ ;\ st := idle$ .
[] $st = abort \;\to\; job := none\ ;\ st := idle$ .
end server .
```

In the first alternative, the server grants access to the critical section. It may have to wait before doing so. The second alternative serves to enable other processes to proceed. In the last alternative, the server terminates the entry protocol of p.

The safety requirement of partial mutual exclusion is the condition that concurrently executed jobs be compatible, as formalized in the invariant

Rq0: $\quad st.q = CS \;\wedge\; st.r = CS \;\Rightarrow\; q = r \;\vee\; job.q * job.r$.

Here and henceforth, q and r stand for processes. For all invariants, we implicitly universally quantify over the free variables, usually q and r.

3.2 Towards the implementation

We turn to the implementation of the **server**. The user processes now get many private variables, the main ones being *job* and *pc*. The sites mentioned in Section 1.1 will be implemented in Section 3.7 as rudimentary processes, in the sense that they have one private variable (*list*), and that they answer to messages. Henceforth, however, when we write "processes", we mean user processes and not sites.

The algorithm uses six message keys `notify`, `withdraw`, `ack`, `gra`, `hello`, `welcome` for messages between processes, and four keys `asklist`, `lower`, `answer`, `done` for the communication between processes and sites.

The concrete state space X is a record type just as the abstract state space Y of Section 3.1, but now combining the private variables of the processes and the sites, and with ten families of shared variables for each of the ten message keys.

Anticipating the code developed below, we define the abstraction function $abs : X \to Y$ to be given by

$$abs(x) = (\#\ job := x.job,\ st := status(x)\ \#) .$$

Here, the pair (#, #) is the record constructor of PVS that corresponds to the type constructor used in section 3.1. The *status* of process p is determined by its current line number via $status(x)(p) = dec(x.pc.p)$ with

$$\begin{aligned} dec(k) = (\ & k = 21\ ?\ idle \\ :\ & k \in \{22 \ldots 26\}\ ?\ entry \\ :\ & k = 27\ ?\ CS \\ :\ & k = 28\ ?\ exit \\ :\ & k \in \{29 \ldots 33\}\ ?\ abort)\ . \end{aligned}$$

Here and henceforth, we use a C-like syntax for conditional expressions.

We concentrate on the first two alternatives of the specification of the **server**. As announced in Section 1.1, we distinguish within the protocol a registration algorithm that provides a competing process with a neighbourhood, and a central algorithm in which the process communicates with the processes in its neighbourhood to obtain permission to enter *CS*. The neighbourhoods are represented in the algorithm by two closely related private variables *nbh* and *nbh0* of the processes. The registration algorithm serves to maintain the invariant:

Rq1: $\quad q\ \textbf{at}\ 27 \;\wedge\; r\ \textbf{at}\ 27 \;\Rightarrow\; q = r \;\vee\; r \in nbh0.q \;\vee\; job.q * job.r$.

The central algorithm uses the sets *nbh0.p* to guarantee the invariant:

Rq2: q **at** 27 $\wedge$ r **at** 27 $\wedge$ $r \in nbh0.q$ $\wedge$ $q \in nbh0.r$ $\Rightarrow$ $job.q * job.r$.

Line 27 of the concrete algorithm is the location of the critical section *CS*. Using the symmetry of the operation $*$, we obtain that every concrete state that satisfies *Rq1* and *Rq2* is mapped by the abstraction function to an abstract state that satisfies predicate *Rq0* of Section 3.1.

3.3 Sketch of the central algorithm

Inspired by the shared-variable mutual exclusion algorithm of Lycklama-Hadzilacos [17], the central algorithm is designed in three layers: an outer protocol to communicate the job to all neighbours, a middle layer to guarantee starvation freedom and to guard against known conflicts, and an inner protocol to guarantee partial mutual exclusion.

The inner protocol is the competition for *CS*. It uses the process numbers mentioned in Section 1.1 for tie breaking, just as Lamport's Bakery algorithm [15]. If q and r are process identifiers with $q < r$, we speak of q as the lower process and r as the higher process. For every pair of processes, the inner protocol gives priority to the lower process, and it lets the higher process determine compatibility. As processes can enter the inner protocol concurrently only within the margins allowed by the middle layer, we expect that the priority bias of the inner protocol is not very noticeable unless the load is so heavy that the performance of any algorithm would be problematic.

Despite the three layers, the central algorithm is rather simple. The outer protocol uses three messages for every process in *nbh*. The middle layer needs no additional messages. The inner protocol uses one message for every higher process in *nbh*, and no messages for the lower processes in *nbh*.

3.4 Into the code of the central algorithm

The code of the central algorithm is given in Figure 1. For an unbroken flow of control, we include the lines 21, 22, 23, 27, which do not belong to the central algorithm. In accordance with the abstraction function of Section 3.2, the lines 21 and 27 correspond to the first two alternatives of the **client**, whereas the server is in *entry* when p is in 22. . . 26, and in *exit* when at line 28.

As process p is always able to receive messages, in Figure 1, the six alternatives of **receive** can be interleaved with the eight alternatives of `central`.

Every process has a private variable *job* of type *Job*, initially *none*. It has private variables *nbh*, *nbh0*, *prio*, *wack*, *after*, *away*, *need*, *prom*, which all hold finite sets of processes. All these sets are initially empty. It has the private variables *pack*, *fun*, and *curlist*, which serve in the registration algorithm and are treated in Section 3.7. The private variable *nbh0* is a history variable. It is set to *nbh* in line 25 and reset in line 28. It is not used in the algorithm, but serves in the proof of correctness.

Process p has a private extendable array *copy.p*, such that $copy.p(q) = job.q$ holds under suitable conditions. It is set when receiving `notify`.$q.p$ and reset in `after`. We use the convention that $copy.p(q) = none$ when q is not in the current range of the array. Initially, the range of *copy.p* is empty.

The lines 22 and 23 are treated in Section 3.7 with the registration algorithm. For now we just assume that *nbh.p* gets some value before *curlist.p* becomes empty at line 23 and that, somehow, predicate *Rq1* is guaranteed.

The central algorithm uses four message keys: `notify`, `withdraw`, `ack`, `gra`. The messages `notify` hold values of the type *Job*, the other messages hold no values, they are of type `void`. The alternatives of **receive** with labels `after` and `prom` correspond to delayed answers. The message key `asklist` is treated in Section 3.7.

```
central(p) :
21:     idle ; choose $job \neq none$ .
22:     await $pcr \leq 42$ ;
        $curlist := \{s \mid L(job)(s) > 0\}$ .
        for all $s \in curlist$ do asklist.$p.s \,!\, L(job)(s)$ od .
23:     await $curlist = \emptyset$ ; // pack and nbh have been formed.
        for all $q \in pack$ do hello.$p.q \,!$ od .
24:     await $pack \cup wack = \emptyset$ ;
        $prio := \{q \mid \neg job * copy(q) \ \wedge \ q \notin after\}$ .
25:     await $prio = \emptyset$ ; $nbh0 := nbh$ ;
        for all $q \in nbh$ do notify.$p.q \,!\, job$ od ;
        $need := \{q \in nbh \mid p < q \vee (q \in away \wedge \neg job * copy(q))\}$ .
26:     await $need = \emptyset$ .
27:     CS .
28:     for all $q \in nbh$ do withdraw.$p.q \,!$ od ;
        $wack := nbh$ ; $job := none$ ; $nbh := nbh0 := \emptyset$ ; goto 21 .
end central .

receive(p) from(q) :
    [] notify.$q.p \,?\, copy(q)$ ;
       if $q < p$ then add $q$ to $prom$ endif .
    [] withdraw.$q.p \,?$ ;
       add $q$ to $after$ ; remove $q$ from $prio$ ;
       if $q < p$ then remove $q$ from $away$ and $need$ endif .
    [] after:   $q \in after \ \wedge \ copy(q) \neq none \quad \rightarrow$
       ack.$p.q \,!$ ; remove $q$ from $after$ ; $copy(q) := none$ .
    [] ack.$q.p \,?$ ; remove $q$ from $wack$ .
    [] gra.$q.p \,?$ ; remove $q$ from $need$ .
    [] prom:   $q \in prom \ \wedge \ (pc \notin \{27, 28\} \ \vee \ job * copy(q)) \quad \rightarrow$
       gra.$p.q \,!$ ; add $q$ to $away$ ; remove $q$ from $prom$ ;
       if $pc \in \{26, 29\} \ \wedge \ \neg job * copy(q)$
       then add $q$ to $need$ endif .
end receive .
```

Figure 1: The central algorithm for process p (with p's private variables)

3.5 A layered solution

As announced, the central algorithm has three layers: an outer protocol to communicate the jobs, a middle layer to regulate access to the inner protocol, and an inner protocol to guarantee *Rq2*. The three layers have waiting conditions in the lines 24, 25, 26, respectively. The outer protocol uses the messages `notify`, `withdraw`, `ack`, and the private variables *job*, *nbh*, *wack*, and *copy*. It can be obtained from Fig. 1 by removing the lines 20 . . . 23, and all commands that use the messages `gra` and the private variables *prio*, *need*, *away*, *prom*. The middle layer consists of all commands that use *prio*. The inner protocol consists of the commands that use the messages `gra` and the private variables *need*, *away*, *prom*.

3.5.1 The outer protocol

In the outer protocol, every process p sends its *job* to all neighbours by means of `notify` messages in line 25, and it withdraws this in line 28. Reception of `notify` and `withdraw` is handled in the first three alternatives of **receive**. In the first line of `notify`, process p registers the *job* of q in $copy.p(q)$. The conditional statement of `notify` belongs to the inner protocol. When process p has received both `notify` and `withdraw` from q, it can execute the alternative `after` of **receive**, send `ack` back to q, and reset $copy.p(q) := none$. In this way, we allow the message `withdraw` to arrive before `notify`, even though it was sent later.

In the fourth alternative of **receive**, when process p receives an `ack` from q, it removes q from its set *wack*. This variable has been set by p to *nbh.p* in line 28, while sending `withdraw` to its neighbours. When process p arrives again at line 24, it waits for *wack* to be empty. In this way, it verifies that all its `withdraw` messages have been acknowledged, to preclude interference by delayed messages.

3.5.2 The inner protocol

To ensure partial mutual exclusion *Rq2*, every process forms in line 25 a set *need* of processes from which it needs "permission" to enter *CS*. Condition *Rq2* is implied by the invariants

Rq2a: $r \in nbh0.q \ \wedge \ q \in nbh0.r$
$\quad \Rightarrow \ r \in need.q \ \vee \ q \in need.r \ \vee \ job.q * job.r \ ,$
Jq0: $r \in need.q \ \Rightarrow \ q \ \mathbf{in} \ \{26, 29\} \ \wedge \ r \in nbh0.q \ .$

Indeed, the first two disjuncts of the consequent of *Rq2a* are false when q and r are at line 27 because of *Jq0*. The proofs of these invariants are postponed to Section 4.4.

At this point, we break the symmetry. Recall that we represent the processes by natural numbers, and that, if $q < r$, we say that process q is *lower* and that r is *higher*. Notifications from lower processes are regarded as requests for permission that must be granted when possible, because we give priority to lower processes. Therefore, when process p receives `notify` from $q < p$, it stores q in *prom.p*. When the alternative `prom` is enabled, process p grants permission by sending `gra` to q. In *away.p*, it records the lower processes to which it has granted permission. If it is at line 26 and in conflict with q, it puts q in *need.p*.

There is a difference in the interpretation of *need.p* for lower and higher processes. If $q < p$, then $q \in need.p$ means that process p is in conflict with q and has granted priority to q. Process p therefore needs to wait for q's `withdraw` message. If $p < q$, then $q \in need.p$ means that process p has requested permisssion from q and is still waiting for the `gra` message (no conflict implied).

3.5.3 The middle layer

Without waiting at line 25, the algorithm of Figure 1 would satisfy *Rq2*, but it would have two defects. At line 26, one low process could repeatedly pass all higher conflicting neighbours. Also, long waiting queues of conflicting processes could form. These defects are treated by the middle layer.

When process p enters at line 24, it assigns to *prio.p* the set of processes with known conflicting *job*s. This set is finite because it is contained in the finite set $\{q \mid copy.p(q) \neq none\}$. Process p then waits for the set *prio.p* to become empty. It removes q from *prio.p* when it receives `withdraw` from q. In this way, the middle layer only admits processes to the inner protocol that are not known to be conflicting with processes in the inner protocol. This improves the performance by making it unlikely that at line 26 long waiting queues of conflicting processes are formed. On the other hand, it ensures starvation freedom. In fact, when process p has executed line 25 and its `notify` messages have arrived, any conflicting neighbour of p that passes p at line 26, will have to wait for p at line 25, and hence cannot pass p again.

Remark. The first ideas for the present paper were tested in [11] in a context with a single resource. There, the `notify` messages are sent in the analogue of line 24 instead of line 25. This is also possible here. It has the effect that processes at line 25 are waiting for processes that arrived earlier at line 25. In other words, it induces a form of a first-come-first-served order. This, however, is not a good idea for resource allocation. Consider, e.g., the following senario.

Process p_0 arrives and starts using resource r_0 in *CS*. Now processes p_k for $k \geq 1$ arrive in their natural order at line 24 at intervals $> \Delta$ (see Section 2.2), needing exclusive access to the resources r_{k-1} and r_k, and with empty *wack*. If the notifications are sent in line 24, they all remain waiting at line 25, because $p_{k-1} \in prio.p_k$, until p_0 has passed *CS*. In the present version, with notifications sent at line 25, the processes with k odd start waiting at line 25, while the processes with k even go through to *CS*. □

3.6 Aborting the entry protocol

As specified in Section 3.1, during *entry*, we give the client the option to signal the server to abort the entry protocol. The client can do this by jumping over the remainder of the code of **central**:

$$\textbf{clientAbort}(p): \quad 22 \leq pc \leq 26 \quad \rightarrow \quad pc := 55 - pc\ .$$

By jumping in this way, the ranges of line numbers in the remainder of the code and in the invariants of Section 4 remain reasonably contiguous. The server reacts to the signal by

```
abort(p) :
    [] $pc = 29 \ \wedge\ need \cap \{q \mid p < q\} = \emptyset \quad \rightarrow$
          for all $q \in nbh$ do withdraw.$p.q$ ! od ; $wack := nbh$ ;
          $job := none$ ; $need := nbh := nbh0 := \emptyset$ ; $pc := 21$ .
    [] $pc = 30 \quad \rightarrow \quad job := none$ ; $nbh := prio := \emptyset$ ; $pc := 21$ .
    [] $pc = 31 \ \wedge\ pack = \emptyset \quad \rightarrow$
          $job := none$ ; $nbh := \emptyset$ ; $pc := 21$ .
    [] $pc = 32 \ \wedge\ curlist = \emptyset \quad \rightarrow$
          for all $q \in pack$ do hello.$p.q$ ! od ; $pc := 31$ .
    [] $pc = 33 \quad \rightarrow \quad job := none$ ; $pc := 21$ .
```

In all cases, *job* is reset to *none* and *pc* is reset to 21, in accordance to the specification. Other variables that have acquired values that are no longer useful are reset as well.

At the lines 29, 31 and 32, the server needs to wait for some additional condition. At line 29, it has to wait for emptiness of the higher part of *need*, necessary to catch the expected `gra` messages. This waiting is short because process p has priority over its higher neighbours. Indeed, the higher part of *need* is empty after $\Gamma + 2\Delta$ (see Section 2.2).

At line 31, the server needs to wait for `welcome` messages, which are treated below in Section 3.7. This is only waiting for messages to arrive. The waiting time is therefore shorter than 2Δ. At line 32, the server needs to complete its registration, as treated in Section 3.7. It takes less than 2Δ to reach line 31, from which it again takes less than 2Δ to become idle.

3.7 The registration algorithm

Recall from the Section 1.1, that the registration algorithm serves to provide the processes with sets *nbh* and that the resources are distributed over sites. Every process registers at the sites for the resources that it needs, and then obtains lists of other registered clients. Let *Site* be the finite set of the sites. We use a fixed function $loc : Rsc \to Site$ to formalize how the resources are distributed over the sites.

We use the job model with upper bound K of Section 2.4. In particular, every job is a function $Rsc \to upto(K)$. A process can only use resource c at level k if it is registered at site $loc(c)$ for level $\geq k$. It therefore has an array *fun* such that $fun.p(s)$ is the p's registration level at site s. When some process obtains a new *job*, it needs at site s the level

$$L(job)(s) = \mathrm{Max}\{job(c) \mid loc(c) = s\} \ .$$

For functions f, $g : Site \to \mathbb{N}$, we define $f \leq g$ to mean $(\forall\ s : f(s) \leq g(s))$.

In line 21 of Figure 1, the client in process p chooses a new job. At line 22, process p may have to wait, to avoid interference with the lowering thread that is treated below. It then sends $L(job)(s)$ to site s if it is positive, and thus asks the site for a lists of clients that might compete for its resources. The set *nbh* gets its contents while the process waits at line 23, through messages from the sites in answer to `asklist`.

We give the sites very small tasks. To avoid fragmentation of the code, we include treatment of messages **lower** and **done** that serve purposes not yet introduced. Site s communicates with the processes by receiving messages `asklist` and `lower`, and answering by `answer` and `done`, respectively; this according to the code:

```
site(s) from(q) :
    [] asklist.q.s ? k ; list(q) := max(list(q), k) ;
       answer.s.q ! {r | list(r) > K − k} .
    [] lower.q.s ? k ; list(q) := k ; done.s.q ! .
end site .
```

In this code, *list* is the private extendable array of site s, that holds the levels of registered processes. The value 0 means not-registered. The answering message `answer` has as value the set of the processes that are in potential conflict at the level k. If process q lowers at site s its level to k, it gets response `done` as an acknowledgment.

Process p receives the messages from site s according to:

```
listen(p) from(s) :
    [] answer.s.p ? v ; nbh := nbh ∪ (v \ {p}) ;
       if fun(s) < L(job)(s) then
           pack := pack ∪ (v \ {p}) ;
           fun(s) := L(job)(s) endif ;
       curlist := curlist \ {s} .
    [] done.s.p ? ; reglist := reglist \ {s} .
end listen .
```

If process p increases its registration level at site s, it collects the potential competitors in the private variables *pack*. When it has received all answers, its sends all members of *pack* a message `hello` in line 23, and waits for the responses `welcome` at line 24. The reason for this is that the processes $q \in pack.p$ can be anywhere in their protocol and need not have $p \in nbh.q$.

The new messages `hello` and `welcome` are between processes. They are treated in the following two alternatives that should be included in **receive** of Figure 1.

```
[] hello.$q.p$ ? ;
   welcome.$p.q$ ! $(pc \in \{26 \ldots 29\} \;\wedge\; q \notin nbh\, ?\, job : none)$ ;
   if $pc \in \{23 \ldots 32\}$ then add $q$ to $nbh$ endif .
[] welcome.$q.p$ ? $v$ ; remove $q$ from $pack$ ;
   if $v \neq none$ then $copy(q) := v$ endif .
```

Process p answers `hello` from q with `welcome`. If it is in $\{26 \ldots 29\}$ and $q \notin nbh.p$, the message `welcome` carries the *job* of p as a belated notification. Otherwise, it only holds *none* as an acknowledgement. If it is in $\{23 \ldots 32\}$, it adds q to *nbh* because, if possible, it needs to send `notify` to q in line 25, and in any case, it needs to send `withdraw` to q later. At this point, the set *nbh0* can become a proper subset of *nbh*.

In `welcome`, the assignment to $copy.p(q)$ ensures that, when process p raises its registration level, it cannot enter its inner protocol when in conflict with q, while q remains in its inner protocol. At this point, the guard of line 25 is necessary for safety. This is a third reason for the middle layer of Section 3.5.3.

3.8 Lowering

In the communication between process and sites the registration level as registered in the variables *fun.p* never decreases and occasionally increases. A process with a high registration level may receive many messages from other processes. A high registration level of a process thus forms a burden for its performance. We therefore offer the client the option to lower its registration level.

Lowering means the choice of a new value *news* for *fun*, which can be equal or lower than the current value. The processes can lower at the sites more or less concurrently with the loop 21–28. For this purpose, each of them gets a separate concurrent thread with a separate process counter *pcr*. We write q **at** ℓ to mean $pc.q = \ell$ if $\ell \in \{21 \ldots 33\}$, and to mean $pcr.q = \ell$ if $\ell \in \{41 \ldots 43\}$. Initially, every process is both **at** 21 and **at** 41.

```
lowering($p$) :
[] $pcr = 41 \quad\to\quad$ choose $news$ with $news \leq fun$ ; pcr := 42 .
[] $pcr = 42 \;\wedge\; (pc = 21 \;\vee\; (pc \geq 24 \;\wedge\; pc \neq 32 \;\wedge\; L(job) \leq news))$
     $\to\quad reglist := \{s \mid news(s) \neq fun(s)\}$ ; $fun := news$ ;
         for all $s \in reglist$ do lower.$p.s$ ! $news(s)$ od ; $pcr := 43$ .
[] $pcr = 43 \;\wedge\; reglist = \emptyset \quad\to\quad pcr := 41$ .
```

The decision to lower does not belong to the protocol and must therefore be taken by the client side of the process. When $pcr = 41$, the client may decide to start lowering by choosing a new value *news* for its variable *fun* and go to line 42. Lowering itself is executed by the server. At line 42, the server informs the sites for which the level is to be modified by sending them a message `lower` with the new value. If the main thread is in 22, 23, or 32, lowering needs to wait to avoid interference. The guard $L(job) \leq news$ is needed to protect the current job of p.

We did not announce lowering in the specification of Section 3.1 because it is functionally void. It is only an option that can possibly improve the performance and lessen the message burden of the system.

Initially, the private sets *pack.p* and *reglist.p* are empty and the functions *fun.p*, *news.p*, and *list.s* are constant zero.

3.9 Summary

In total, the transition system consists of the 29 step relations: there are 13 step relations at the line numbers 21…33 of Figure 1 and Section 3.6. There is the step relation **clientAbort** of Section 3.6. There are the 3 step relations at the line numbers 41, 42, 43 of Section 3.8. Finally, there are 12 message reception step relations, 10 for each of the ten message keys `notify`, `withdraw`, `ack`, `gra`, `hello`, `welcome`, `asklist`, `lower`, `answer`, `done`, and the two delayed message reception step relations `after` and `prom` of Figure 1.

For the discussion of progress, we partition the step relations or steps of the algorithm in a different way. The 3 steps at the lines 21, 41, and **clientAbort** are called the *free* steps, because they can be executed, but there is never a reason to do so. We define the *forward* steps to be the 12 steps at the line numbers 22. . . 33, and the *lowering* steps to be the 2 steps at the lines 42, 43. Execution of *CS* at line 27 is taken to be a forward step, because we may want to express that *CS* terminates. The 12 steps for (possibly delayed) message reception are called *triggered steps*.

Coming back to the abstraction function of Section 3.2, it is now straightforward to verify that every step of the concrete algorithm corresponds to a step of the abstract algorithm or to a skip step that does not change the abstract state. Moreover, every abstract step of a client corresponds to a unique concrete step of the same client. Note that a concrete client can do a lowering step at line 41, which corresponds to a skip step for the specification.

Two points remain: we have to verify that the concrete algorithm satisfies partial mutual exclusion and that it makes progress.

4 Verification of Partial Mutual Exclusion

In this section, we prove that *Rq1* and *Rq2* are invariants of the algorithm.

In Section 4.1, we describe the verification of safety by means of invariants. In Section 4.2, we describe some choices we made for the ease of proof management.

In Section 4.3 we develop the invariants of the outer protocol that are needed in the proof of *Rq2*. As indicated in Section 3.5.2, *Rq2* is implied by *Jq0* and *Rq2*a. These invariants of the inner protocol are proved in Section 4.4, together with a number of auxiliary ones.

In Section 4.5, we add the registration algorithm, verify that the new messages are modelled correctly and that this addition does not disturb the safety properties of the central algorithm. Section 4.6 shows that it indeed serves its purpose and guarantees the invariant *Rq1*.

4.1 Using invariants

In a distributed algorithm, at any moment, many processes are able to do a step that modifies the global state of the system. The only way to reason successfully and reliably about such a system is to analyse the properties that cannot be falsified by any step of the system. These are the invariants.

Formally, a predicate is called an *invariant* of an algorithm if it holds in all reachable states. A predicate J is called *inductive* if it holds initially and every step of the algorithm from a state that satisfies J results in a state that also satisfies J. Every inductive predicate is an invariant. Every predicate implied by an invariant is an invariant.

When a predicate is inductive, this is often easily verified. In many cases, the proof assistant PVS is able to do it without user intervention. It always requires a big case distinction, because the transition system has many different alternatives.

Most invariants, however, are not inductive. Preservation of such a predicate by some alternatives needs the validity of other invariants in the precondition. We use PVS to pin down the problematic alternatives, but human intelligence is needed to determine the useful other invariants.

In proofs of invariants, we therefore use the phrase "*preservation of J at $\ell_1 \dots \ell_m$ follows from $J_1 \dots J_n$*" to express that every step of the algorithm with precondition $J \land J_1 \dots J_n$ has the postcondition J, and that the additional predicates $J_1 \dots J_n$ are only needed for the alternatives $\ell_1 \dots \ell_m$.

We use the following names for the alternatives. The first 8 alternatives of **central** in Figure 1 are indicated by the line numbers. The alternatives of **receive** are indicated by the message names and the labels `after` and `prom`.

For all invariants postulated, the easy proof that they hold initially is left to the reader. We use the term invariant in a premature way. See the end of this section.

4.2 Proof engineering

Effective management of the combined design and verification of such an algorithm requires a number of measures. We give most invariants names of the form Xqd, where X stands for an upper case letter and d for a digit. This enables us to rename the invariants in the text and the PVS proof files by query-replace, and to keep them consistent. Indeed, any modification of proof files must be done very carefully to avoid that the proof is destroyed. Using short distinctive names also makes it easy to search for definitions and to see when all of them have been treated.

Line numbers may change during design. In order to use query-replace for this in all documents, we use line numbers of two digits. In this way, we preclude that the invariants get renamed by accident. This is also the reason to use disjoint ranges for the line numbers for forward steps and for lowering steps.

There is a trade off in the size of the invariants. Smaller invariants are easier to prove and easier to apply, but one needs more of them, and they are more difficult to remember. We therefore often combine a number of simple properties in a single invariant, see *Iq1* below. Bigger invariants are sometimes needed to express different aspects of a complicated state of affairs, compare *Iq2* below.

4.3 Invariants of the outer protocol

For now, we restrict ourselves to the transition system with the 14 transitions of Figure 1 and the 6 aborting transitions of Section 3.6. For simplicity of exposition, in the discussion of the invariants, we concentrate on the first set of transitions and ignore the aborting transitions. Yet, the invariants presented are also preserved by the aborting transitions. The nine steps of Sections 3.7 and 3.8 are added in Sections 4.5 and 4.6.

We have two invariants about neighbourhoods:

Iq0: $\quad q \notin nbh.q$,

Iq1: $\quad r \in nbh0.q \;\Rightarrow\; q \;\textbf{in}\; \{26 \dots 29\} \;\land\; r \in nbh.q$.

These predicates are easily seen to be inductive.

At line 24, the processes wait for acknowledgements as expressed by emptiness of *wack*. This corresponds to the invariant:

Iq2: $\quad \texttt{withdraw}.q.r + |\, q \in after.r \,| + \texttt{ack}.r.q = |\, r \in wack.q \,|$.

Recall that $\texttt{withdraw}.q.r$ is the number of `withdraw` messages from q to r and that $\texttt{ack}.r.q$ is the number of `ack` messages from r to q. For Boolean b, we define $|\, b \,| \in \mathbb{N}$ to be 1 if b holds, and 0 otherwise. Predicate *Iq2* is a concise expression of a complicated fact. Namely, $r \in wack.q$ holds if and only if if there is a `withdraw` message in transit from q to r, or an `ack` message in transit from r to q, or $q \in after.r$. Furthermore, the three possibilities are mutually exclusive. Finally, there is at most one `withdraw` message from q to r, and at most one `ack` message from r to q. One could therefore split *Iq2* into 9 different invariants.

Preservation of *Iq2* when `withdraw` is sent at line 28 follows from the inductive invariant:

Iq3: q **in** $\{25\ldots 30\} \;\Rightarrow\; wack.q = \emptyset$.

For practical purposes, it is useful to notice that *Iq2* and *Iq3* together imply

Iq2a: q **in** $\{25\ldots 30\} \;\Rightarrow\; \texttt{withdraw}.q.r = 0 \;\wedge\; q \notin after.r$.

As announced, one of the functions of the outer protocol is to guarantee that, under suitable conditions, process r has the job of q in its variable $copy.r(q)$. In fact, the conditions are that r is in *nbh0.q* and that there is no message `notify` in transit from q to r, as expressed in the invariant

Iq4: $r \in nbh0.q \;\wedge\; \texttt{notify}.q.r = \bot \;\Rightarrow\; copy.r(q) = job.q$.

Preservation of *Iq4* at line 21 follows from *Iq1*. Preservation at `after` follows from *Iq1* and *Iq2a*. Preservation at line 25 and `notify` follows from *Iq1* and the new invariants:

Iq5: $job.q = none \;\;\equiv\;\; q$ **at** 21 ,
Iq6: q **in** $\{26\ldots 29\} \;\wedge\; \texttt{notify}.q.r \neq \bot \;\Rightarrow\; \texttt{notify}.q.r = job.q$.

Predicate *Iq5* is inductive. Preservation of *Iq6* at line 25 follows from the new invariant

Iq7a: q **at** $25 \;\Rightarrow\; \texttt{notify}.q.r = \bot \;\wedge\; copy.r(q) = none$.

Predicate *Iq7a* is logically implied by *Iq2*, *Iq3*, and the new invariant:

Iq7: $(\texttt{notify}.q.r = \bot \;\wedge\; copy.r(q) = none)$
$\vee\; (q$ **in** $\{26\ldots 29\} \;\wedge\; r \in nbh.q)$
$\vee\; \texttt{withdraw}.q.r > 0 \;\vee\; q \in after.r$.

Preservation of *Iq7* at `after` follows from the new invariant:

Iq8: $\texttt{notify}.q.r = \bot \;\vee\; copy.r(q) = none$.

Preservation of *Iq8* at line 25 follows from *Iq7a*.

This is not circular reasoning: the above argument shows that, if all predicates *Iq** hold in the precondition of any step, they also hold in the postcondition. Therefore, the conjunction of them is inductive, and each of them is an invariant.

4.4 The proof of partial mutual exclusion

In Section 3.5.2, we saw that the partial mutual exclusion predicate *Rq2* is implied by *Jq0* and *Rq2a*. In this section, we prove that these two predicates are invariants.

Preservation of predicate *Jq0* at `prom` follows from *Iq4*, *Jq0*, and the new invariants

Rq1a: q **in** $\{26\ldots 29\} \;\wedge\; r$ **in** $\{26\ldots 29\}$
$\Rightarrow\; q = r \;\vee\; r \in nbh0.q \;\vee\; job.q * job.r$,
Jq1: $q \in prom.r \;\Rightarrow\; q < r$,
Jq2: $q < r \;\Rightarrow\; |\,\texttt{notify}.q.r \neq \bot\,| + |\,q \in prom.r\,| + \texttt{gra}.r.q = |\,r \in need.q\,|$.

Predicate *Rq1a* is a strengthening of *Rq1* of Section 3.2, which is guaranteed by the registration algorithm as shown in Section 4.6 below. Predicate *Jq1* is inductive. Preservation of *Jq2* at 25 and `prom` follows from *Iq5*, *Jq0* and *Jq1*. Note the similarity of *Jq2* with *Iq2*.

Predicate *Rq2a* of Section 3.5.2 is implied by *Iq0*, *Iq1*, *Iq2a*, and the new invariants:

Jq3: $q < r \;\wedge\; r \in \mathit{nbh0}.q \;\Rightarrow\; r \in \mathit{need}.q \;\vee\; q \in \mathit{away}.r$,
Jq4: $q \in \mathit{away}.r \;\wedge\; q \in \mathit{nbh0}.r \;\wedge\; \texttt{withdraw}.q.r = 0$
$\Rightarrow\; q \in \mathit{need}.r \;\vee\; \mathit{job}.q * \mathit{job}.r$.

Preservation of *Jq3* at `withdraw` follows from *Iq1*, *Iq2a*. At `gra`, it follows from the new invariant

Jq5: $\texttt{gra}.r.q > 0 \;\Rightarrow\; q \in \mathit{away}.r$.

Preservation of *Jq4* at 21 follows from *Iq1*. Preservation at `prom` follows from *Iq1*, *Iq4*, *Jq0*, *Jq1*, and *Jq2*. Preservation at line 25 and at `gra` and `withdraw` follows from *Iq4*, *Jq5* and the new invariants:

Jq6: $q \in \mathit{away}.r \;\Rightarrow\; q < r \;\wedge\; \texttt{notify}.q.r = \bot$,
Jq7: $q \in \mathit{away}.r \;\wedge\; \texttt{withdraw}.q.r = 0 \;\Rightarrow\; r \in \mathit{nbh0}.q$.

Preservation of *Jq5* at `withdraw` follows from *Iq2a*, *Jq0*, and *Jq2*. Preservation of *Jq6* follows at line 25 from *Iq1*, *Iq2*, *Iq3*, and *Jq7*, and at `prom` from *Jq1*, *Jq2*. Preservation of *Jq7* at 25 and 28 follows from *Iq1*, and at `prom` and `withdraw` from *Jq0*, *Jq1*, *Jq2*, and *Jq6*.

This concludes the proof of the invariants *Jq0* and *Rq2a* under assumption of *Rq1a*, and hence of *Rq2*.

4.5 Adding registration

We now add the registration algorithm to the central algorithm, i.e., we extend the transition system with the nine transitions of Sections 3.7 and 3.8. There are three things to verify. The modeling must be correct, the proof of the central algorithm must not be disturbed, and condition *Rq1a* of Section 4.4 must be guaranteed. The first two points are treated in this section. The third point is postponed to the next section.

The new messages `asklist`, `answer`, `welcome`, and `lower` are not void, and are therefore modelled in the same way as `notify`. This gives us the obligation to prove, for each of these four message keys m, that the value of m is $\bot$ whenever a message m is sent. For the message keys `asklist` and `answer`, this follows from the invariants (compare *Iq2*):

Kq0: $|\, \texttt{asklist}.q.s \neq \bot \,| + |\, \texttt{answer}.s.q \neq \bot \,| = |\, s \in \mathit{curlist}.q \,|$,
Kq1: $q \textbf{ in } \{23, 32\} \;\vee\; \mathit{curlist}.q = \emptyset$.

Predicate *Kq1* is inductive. Predicate *Kq0* is preserved at 22 because of *Kq1*. The invariants *Kq0* and *Kq1* together imply

Kq0a: $\texttt{answer}.s.q \neq \bot \;\Rightarrow\; q \textbf{ in } \{23, 32\}$.

For the message keys `welcome` and `lower`, $m.q.r = \bot$ holds whenever m is sent because of the invariants:

Kq2: $\texttt{hello}.q.r + |\, \texttt{welcome}.r.q \neq \bot \,| = |\, q \textbf{ in } \{24, 31\} \;\wedge\; r \in \mathit{pack}.q \,|$,
Kq3: $|\, \texttt{lower}.q.s \neq \bot \,| + \texttt{done}.s.q = |\, q \textbf{ at } 43 \;\wedge\; s \in \mathit{reglist}.q \,|$.

Predicate *Kq2* is preserved at `answer` because of *Kq0a*. Predicate *Kq3* is inductive.

We turn to the question whether the central algorithm is disturbed by the new registration commands. The only variables that the registration algorithm shares with the central algorithm are *pc*, *pcr*, *nbh*, *pack*, and *copy*. Sharing *pc* is harmless, because the flow of controle is not modified. Sharing *pack* and *pcr* is harmless because the central algorithm has no invariants for *pack* and *pcr*. Sharing *nbh* is almost harmless because all invariants except *Iq0* allow enlarging *nbh*. Predicate *Iq0* is preserved by `hello` because of $\texttt{hello}.q.q = 0$ which follows from *Kq2* together with the inductive invariant

Kq4: $q \notin \mathit{pack}.q$.

Modification of *copy* by `welcome` requires new invariants. Predicate *Iq4* is preserved at `welcome` because of *Iq1* and the new invariant

Kq5: $q \textbf{ in } \{26\ldots 29\} \Rightarrow \texttt{welcome}.q.r \in \{\bot, \mathit{none}, \mathit{job}.q\}$.

Predicate *Iq7* is preserved at `welcome` because of

Kq6: $\texttt{welcome}.q.r \in \{\bot, \mathit{none}\} \lor \texttt{withdraw}.q.r > 0$
$\lor\ q \in \mathit{after}.r \lor (q \textbf{ in } \{26\ldots 29\} \land r \in \mathit{nbh}.q)$.

Predicate *Iq8* is preserved at `welcome` because of

Kq7: $\texttt{welcome}.q.r \in \{\bot, \mathit{none}\} \lor (\texttt{notify}.q.r = \bot \land \mathit{copy}.r(q) = \mathit{none})$.

Predicate *Kq5* is preserved at 25 because of *Iq2a* and *Kq6*. Predicate *Kq6* is preserved at `after` and `hello` because of *Iq1* and *Kq7*. Predicate *Kq7* is preserved at 25 and `hello` because of *Iq2a*, *Iq7*, and *Kq6*.

This concludes the proof that the registration algorithm preserves the invariants claimed for the central algorithm.

4.6 Safety of registration

We approach predicate *Rq1a* in a bottom-up fashion. The lowering thread does not interfere with the main thread because of the inductive invariants:

Lq0: $q \textbf{ in } \{23, 32\} \Rightarrow q \textbf{ in } \{41, 42\}$,
Lq1: $\mathit{news}.q \le \mathit{fun}.q$.

The sets *prio* and *pack* are only nonempty in specific locations, as expressed by the invariants:

Lq2: $q \in \mathit{prio}.r \Rightarrow r \textbf{ in } \{25, 30\} \land q \notin \mathit{after}.r$,
Lq3: $q \textbf{ in } \{23, 24, 31, 32\} \lor \mathit{pack}.q = \emptyset$.

Indeed, *Lq2* is inductive. Predicate *Lq3* is preserved by `answer` because of *Kq0a*.

For the communication with the sites, we claim the invariants:

Lq4: $\texttt{asklist}.q.s \in \{\bot, L(\mathit{job}.q)(s)\}$,
Lq5: $\texttt{lower}.q.s \in \{\bot, \mathit{fun}.q(s)\}$.

Predicate *Lq4* is preserved at 21 and 28 because $\texttt{asklist}.q.s \neq \bot$ implies q **at** 23, as follows from *Kq0* and *Kq1*. Predicate *Lq5* is preserved at `answer` because of *Kq0a*, *Kq3*, and the mutual exclusion invariant *Lq0*.

There are subtle relations between $L(\mathit{job}.q)(s)$, $\mathit{fun}.q(s)$, and $\mathit{list}.s(q)$ expressed by the invariants:

Lq6: $q \textbf{ in } \{22, 33\} \lor s \in \mathit{curlist}.q \lor L(\mathit{job}.q)(s) \le \mathit{fun}.q(s)$,
Lq7: $q \textbf{ in } \{23\ldots 32\} \land \texttt{asklist}.s.q = \bot \Rightarrow L(\mathit{job}.q)(s) \le \mathit{list}.s(q)$,
Lq8: $\mathit{fun}.q(s) \le \mathit{list}.s(q)$.

Predicate *Lq6* is preserved at line 42 because of *Iq5*. Predicate *Lq7* is preserved by `asklist` because of *Lq4*. It is preserved by `lower` because of *Kq3*, *Kq1*, *Lq0*, *Lq5*, and *Lq6*. Predicate *Lq8* is preserved at 42 because of *Lq1*, at `answer` because of *Kq0*, *Kq1*, *Lq7*, and at `lower` because of *Lq5*.

After this preparation, we turn to the proof that the registration algorithm satisfies it purpose, i.e., guarantees predicate *Rq1a* of Section 4.4. At line 23, process q expects and receives an answer from site s. This answer is a set that contains process r iff $L(\mathit{job}.q)(s) + \mathit{list}.s(r) > K$, where $\mathit{list}.s$ is the value at the time of sending the answer. In this way, we arrive at the invariant:

Mq0: q **in** $\{23 \ldots 29\} \Rightarrow q = r \vee r \in \mathit{nbh}.q \vee \texttt{hello}.r.q > 0$
$\vee (r$ **in** $\{23, 32\} \wedge q \in \mathit{pack}.r) \vee L(\mathit{job}.q)(s) + \mathit{fun}.r(s) \leq K$
$\vee (s \in \mathit{curlist}.q \wedge (\texttt{answer}.s.q = \bot \vee r \in \texttt{answer}.s.q))$.

Predicate *Mq0* is preserved at line 22, 42, `welcome`, `asklist` because of *Kq0a*, *Lq1*, *Kq2*, *Lq4*, *Lq8*. It is preserved at `answer` because of *Kq0a* and the new invariant:

Mq1: q **in** $\{23 \ldots 29\} \wedge \texttt{answer}.s.r \neq \bot$
$\Rightarrow q = r \vee r \in \mathit{nbh}.q \vee q \in \texttt{answer}.s.r$
$\vee L(\mathit{job}.q)(s) + L(\mathit{job}.r)(s) \leq K$
$\vee (s \in \mathit{curlist}.q \wedge (\texttt{answer}.s.q = \bot \vee r \in \texttt{answer}.s.q))$.

Predicate *Mq1* is preserved at 21, 22, 28 because of *Kq0a*. It is preserved by `asklist` because of *Kq0*, *Kq1*, *Lq4*, *Lq7*.

The predicates *Mq0*, *Kq2*, *Kq1*, *Lq6* together imply the derived invariant:

Mq0a: q **in** $\{24 \ldots 29\} \wedge r$ **in** $\{24 \ldots 29\} \wedge \mathit{pack}.r = \emptyset$
$\Rightarrow q = r \vee r \in \mathit{nbh}.q \vee \mathit{job}.q * \mathit{job}.r$.

Predicate *Mq0a* approximates *Rq1a*, but the disjunct $r \in \mathit{nbh}.q$ of *Mq0a* is weaker than the disjunct $r \in \mathit{nbh0}.q$ of *Rq1a*. As a remedy, we postulate the invariant:

Mq2: q **in** $\{26 \ldots 29\} \wedge r \in \mathit{nbh}.q$
$\Rightarrow r \in \mathit{nbh0}.q \vee \texttt{welcome}.q.r = \mathit{job}.q \vee \mathit{copy}.r(q) = \mathit{job}.q$.

Predicate *Mq2* is preserved at `notify`, `answer`, `after`, `welcome`, `hello` because of *Iq1*, *Iq2a*, *Iq5*, *Iq8*, *Kq0a*, *Kq2*, *Kq5*.

The conjunction of *Mq0a* and *Mq2* is not strong enough to imply *Rq1a*. We need yet another invariant:

Mq3: q **in** $\{26 \ldots 29\} \wedge r$ **in** $\{25 \ldots 29\}$
$\Rightarrow q = r \vee r \in \mathit{nbh0}.q \vee q \in \mathit{prio}.r \vee \mathit{job}.q * \mathit{job}.r$.

Predicate *Mq3* is preserved at lines 24 and 25 because of *Mq0a* and *Iq2a*, *Iq5*, *Kq2*, *Lq3*, *Mq2*. It is preserved at `welcome` because of *Iq2a*.

We can now conclude the proof of partial mutual exclusion *Rq0*. Indeed, predicate *Rq1a* of Section 4.4 is implied by *Mq3* and *Lq2*, because *Lq2* eliminates the alternative $q \in \mathit{prio}.r$ of *Mq3*. This concludes the proof of the invariants *Rq1* and *Rq2* of Section 3.2. It follows that the abstraction function of Section 3.2 maps all reachable states of the concrete algorithm to abstract states that satisfy condition *Rq0* of Section 3.1.

5 Absence of Localized Deadlock

Absolute deadlock in a transition system means that no transition can modify the state anymore. In our transition system, there are always infinitely many idle processes and these can always go to *entry*. Therefore, absolute deadlock cannot occur, but this fact is not very meaningful.

We introduce localized deadlock to pin down the kind of blocking that can or that cannot occur in the algorithm. In Section 5.1, we introduce some invariants needed for this purpose. In Section 5.2, we zoom in on specific properties related to disabledness. In Section 5.3, we define and prove the absence of localized deadlock.

5.1 Invariants against blocking

In order to prove absence of blocking, we need additional invariants about *need*, *copy*, *prio*, because of the guards of the alternatives 25, 26, `after`, and `prom`.

We need two invariants for *need* in the inner protocol:

Nq0: $q < r \ \wedge\ q \in \mathit{need}.r \ \Rightarrow\ q \in \mathit{away}.r$,
Nq1: $q < r \ \wedge\ q \in \mathit{need}.r \ \wedge\ \mathit{job}.q * \mathit{job}.r \ \Rightarrow\ \texttt{withdraw}.q.r > 0$.

Predicate *Nq0* is inductive. Predicate *Nq1* is preserved at 21 because of *Iq5*, at 25 because of *Iq1*, *Iq4*, *Jq6* and *Jq7*, and at 28 because of *Iq1*, *Jq0*, *Jq7*, and *Nq0*. It is preserved at `prom` because of *Iq4*, *Jq0*, and *Jq2*.

We also need a new invariant of the outer protocol (compare *Iq7*):

Nq2: $\texttt{notify}.q.r = \bot \ \wedge\ \mathit{copy}.r(q) = \mathit{none} \ \wedge\ \texttt{welcome}.q.r \in \{\bot, \mathit{none}\}$
$\quad \Rightarrow\ q \notin \mathit{after}.r \ \wedge\ \texttt{withdraw}.q.r = 0$.

Preservation of *Nq2* at 28, `hello`, `after`, and `notify` follows from *Iq1*, *Iq2*, *Iq4*, *Iq5*, *Kq2*, *Mq2*, and the new postulate

Nq3: $\texttt{notify}.q.r \neq \mathit{none}$.

Predicate *Nq3* is preserved at 25 because of *Iq5*.

Next to *Lq2*, we need a second invariant about *prio*:

Nq4: $q \in \mathit{prio}.r \ \Rightarrow\ \neg\, \mathit{copy}.r(q) * \mathit{job}.r$.

Predicate *Nq4* is preserved at 21, 28, and `after` because of *Lq2*. It is preserved at `notify` and `welcome` because of *Iq8* and *Kq7*. This concludes the construction of the invariants for progress.

5.2 Disabledness and conflicts

We use the invariants obtained thus far to derive four so-called *waiting invariants* that focus on disabledness of processes in relation to the occurrence of conflicts. Forward steps can be disabled at the lines 23, 24, 25, 26 (and 29, 31, 32) by nonemptiness of *curlist*, *wack*, *prio*, *need*, respectively. Message reception is disabled when there is no message.

Disabledness of processes is often caused by processes that are in conflict, i.e., have conflicting jobs. We write $q \bowtie r$ to denote that q and r are in conflict. So we have

$$q \bowtie r \quad \equiv \quad \neg\, \mathit{job}.q * \mathit{job}.r \ .$$

Let $\mathit{dAfter}(q, r)$ and $\mathit{dProm}(q, r)$ be the conditions, respectively, that the alternatives `after` and `prom` for sending `ack` or `gra` from q to r are disabled:

$$\begin{aligned} \mathit{dAfter}(q,r) &\equiv r \notin \mathit{after}.q \ \vee\ \mathit{copy}.q(r) = \mathit{none} \ , \\ \mathit{dProm}(q,r) &\equiv r \notin \mathit{prom}.q \\ &\qquad \vee\ (q \ \textbf{in}\ \{27, 28\} \ \wedge\ \neg\, \mathit{job}.q * \mathit{copy}.q(r)) \ . \end{aligned}$$

For emptiness of *wack.q*, the invariants *Iq2* and *Nq2* imply the waiting invariant:

Waq0: $\texttt{withdraw}.q.r = \texttt{ack}.r.q = 0 \ \wedge\ \texttt{notify}.q.r = \bot$
$\quad \wedge\ \texttt{welcome}.q.r \in \{\bot, \mathit{none}\} \ \wedge\ \mathit{dAfter}(r, q) \ \Rightarrow\ r \notin \mathit{wack}.q$.

For emptiness of *prio.q*, the invariants *Kq7*, *Lq2*, *Mq2*, and *Nq4*, together with *Iq4*, *Iq5*, *Iq7*, and *Iq8* imply the waiting invariant

Waq1: $r \in \mathit{prio}.q \;\wedge\; \mathtt{withdraw}.r.q = 0 \;\wedge\; \mathtt{welcome}.q.r \in \{\bot, \mathit{none}\}$
$\Rightarrow\; r$ **in** $\{26\ldots29\} \;\wedge\; q \bowtie r$.

For emptiness of *need.q*, we are forced to make a case distinction. Using the invariants *Nq0* and *Nq1* together with *Iq1*, *Jq0*, and *Jq7*, we obtain the waiting invariant

Waq2: $r < q \;\wedge\; r \in \mathit{need}.q \;\wedge\; \mathtt{withdraw}.r.q = 0 \;\Rightarrow\; r$ **in** $\{26\ldots29\} \;\wedge\; q \bowtie r$.

It follows from *Iq4*, *Iq5*, *Jq0*, and *Jq2*, that we have

Waq3: $q < r \;\wedge\; r \in \mathit{need}.q \;\wedge\; \mathtt{gra}.r.q = 0 \;\wedge\; \mathtt{notify}.q.r = \bot$
$\wedge\; \mathit{dProm}(r,q) \;\Rightarrow\; r$ **in** $\{27, 28\} \;\wedge\; q \bowtie r$.

5.3 Localized deadlock

In this section we prove absence of localized deadlock. Informally speaking, this means that, when there are blocked server processes, some of them are in conflict with uncollaborating processes. The result is not used in the proof of Theorem 2 in the next section, and it follows from Theorem 2. Yet, an independent proof of the result is a good preparation of the more complicated proof of Theorem 2.

The concept is formally defined by means of the partitioning of the steps introduced in Section 3.9. We define a process p to be *silent* when every forward, lowering, or triggered step of p is disabled, there is no message from p or towards p in transit, and no process can do a triggered step that sends a message to p. We define p to be *locked* when it is silent and not at line 21.

Let W be an arbitrary set of processes (willing to do steps). The set W is said to be *silent* if all its processes are silent. It is said to be *locked* if it is silent and contains locked processes. We define *absence of W-deadlock* to mean that, if W is locked, then it contains some process that is in conflict with a process not in W. We define *absence of localized deadlock* to be absence of W-deadlock for every set W.

This is our next result:

Theorem 1 *Assume that a set W of processes is locked. Then there are processes $q \in W$ and $r \notin W$ with $q \bowtie r$.*

Proof. The algorithm clearly satisfies the invariant that every process is always in $\{21\ldots33\}$. We treat the line numbers $\neq 21$ one by one. The processes in $\{27, 28, 30, 33\}$ can do a forward step. Therefore, all processes in W are in $\{21\ldots26, 29, 31, 32\}$.

If process $q \in W$ is locked at line 22, then $\mathit{pcr}.q = 43$. As q cannot do the forward step of line 43, the set *reglist.q* is nonempty, say $s \in \mathit{reglist}.q$. By *Kq3*, it follows that q can receive a message `done` from site s or site s can receive `lower` from q. This contradicts the assumption that q is locked.

Similarly, because W is locked, we have $\mathtt{asklist}.q.s = \mathtt{answer}.s.q = \bot$ for all $q \in W$ and all sites s. By *Kq0*, this implies $\mathit{curlist}.q = \emptyset$ for all $q \in W$. It follows that W has no processes locked at the lines 23 and 32.

Similarly, for all pairs q, r with $q \in W$ or $r \in W$, the values of $\mathtt{notify}.q.r$ and $\mathtt{welcome}.q.r$ are $\bot$ and the values of $\mathtt{hello}.q.r$, $\mathtt{withdraw}.q.r$, $\mathtt{ack}.q.r$, and $\mathtt{gra}.q.r$ are all 0. Also, $\mathit{dAfter}(q,r)$ and $\mathit{dProm}(q,r)$ hold. This simplifies the waiting invariants *Waq** of Section 5.2 considerably. In fact, for $q \in W$ and r arbitrary, we obtain:

Wax0: $r \notin \mathit{wack}.q$,
Wax1: $r \in \mathit{prio}.q \;\Rightarrow\; r$ **in** $\{26\ldots29\} \;\wedge\; q \bowtie r$,
Wax2: $r < q \;\wedge\; r \in \mathit{need}.q \;\Rightarrow\; r$ **in** $\{26\ldots29\} \;\wedge\; q \bowtie r$,
Wax3: $q < r \;\wedge\; r \in \mathit{need}.q \;\Rightarrow\; r$ **in** $\{27, 28\} \;\wedge\; q \bowtie r$.

It follows from *Kq2* and *Wax0* that W has no processes disabled at 24 and 31. This proves that all processes of W are in $\{21, 25, 26, 29\}$.

If process $q \in W$ is locked at line 29, there is a process $r \in \mathit{need}.q$ with $q < r$. By *Wax3*, process r is in $\{27, 28\}$ and $q \bowtie r$. As r is in $\{27, 28\}$, it is not in W, thus proving the assertion.

We may therefore assume that W contains no processes locked at 29. Assume that W contains processes locked at 26. Let q be the lowest process in W locked at 26. The set $\mathit{need}.q$ is nonempty, say $r \in \mathit{need}.q$. It follows from *Iq0* and *Jq0* that $r \neq q$. Then *Wax2* and *Wax3* imply $q \bowtie r$. Moreover, if $q < r$, then r is in $\{27, 28\}$ by *Wax3*, so that $r \notin W$. On the other hand, if $r < q$, then r is in $\{26 \ldots 29\}$ by *Wax2*. If $r \in W$, then r would be at 26, contradicting the minimality of q. This proves that $r \notin W$ in either case.

Assume that W also contains no processes locked at 26. Then it has some process $q \in W$ locked at 25 and the set $\mathit{prio}.q$ is nonempty, say $r \in \mathit{prio}.q$. Now *Wax1* implies that r is in $\{26 \ldots 29\}$ and $q \bowtie r$. Because r is in $\{26 \ldots 29\}$, it is not in W. □

6 Progress

The algorithm satisfies strong progress properties. In this section, we introduce and formalize the progress properties.

We introduce weak fairness in Section 6.1. Section 6.2 contains the formalization in (linear-time) temporal logic. Section 6.3 formalizes weak fairness. In Section 6.4, we introduce absence of localized starvation to unify starvation freedom and concurrency as announced in the Introduction, and we announce the main progress result Theorem 2. In essence, the proof of this result looks like the proof of Theorem 1, but it takes several sections to treat the various line numbers. In fact, in Section 6.5, the proof of the Theorem is reduced to an investigation of line numbers in the range 22. . . 33. Section 6.6 treats all line numbers except for 26 and 29, which are treated in Section 6.7. The Theorem is proved in Section 6.8. Progress for the lowering thread of Section 3.8 is treated in Section 6.9.

6.1 Weak fairness

First, however, weak fairness needs to be introduced. Roughly speaking, a system is called weakly fair if, whenever some process from some time onward always can do a step, it will do the step. Yet if a process is *idle*, it must not be forced to be interested in *CS*. Similarly, if a process is waiting a long time in *entry*, we do not want it to be forced to abort. In terms of the partitioning of the steps of Section 3.9, we therefore do not enforce weak fairness for the free transitions, but only for the triggered transitions and (to some extent) the forward and the lowering transitions.

Formally, we do not argue about the fairness of systems, but characterize the executions they can perform. Recall that an *execution* is an infinite sequence of states that starts in an initial state and for which every pair of subsequent states satisfies the next-state relation. The next-state relation is defined as the union of a number of step relations.

An execution is called *weakly fair* for a step relation iff, when the step relation is from some state onward always enabled, it will eventually be taken. For example, if some process p is at line 22, we expect that p will eventually execute line 22. If some message m from q to r is in transit, we expect that r eventually receives message m. By imposing weak fairness for some step relations, we restrict the attention to the executions that are weakly fair for these step relations.

6.2 Formalization in temporal logic

Recall that the state space X of the algorithm is the Cartesian product of the private state spaces of the processes and the sites, augmented with shared variables for messages in transit. Executions are infinite sequences of states, i.e., elements of the set X^ω. For a state sequence $xs \in X^\omega$, we write $xs(n)$ for the nth element of xs. Occasionally, we refer to $xs(n)$ as the state at time n. For a programming variable v, we write $xs(n).\mathtt{v}$ for the value of v in state $xs(n)$.

We identify a predicate U on the state space X with the subset of X where U holds. For instance, the predicate p **at** 21 is identified with the set of the states in which process p is at line 21.

For a predicate or set of states U, we define $[\![U]\!] \subseteq X^\omega$ as the set of infinite sequences xs with $xs(0) \in U$. For a relation $A \subseteq X^2$, we define $[\![A]\!]_2 \subseteq X^\omega$ as the set of sequences xs with $(xs(0), xs(1)) \in A$.

For $xs \in X^\omega$ and $k \in \mathbb{N}$, we define the suffix $(xs|k)$ by $(xs|k)(n) = xs(k+n)$. For a subset $P \subseteq X^\omega$ we define $\Box P$ (*always* P) and $\Diamond P$ (*eventually* P) as the subsets of X^ω given by

$$\begin{array}{lcl} xs \in \Box P & \equiv & (\forall\, k \in \mathbb{N} : (xs|k) \in P) \ , \\ xs \in \Diamond P & \equiv & (\exists\, k \in \mathbb{N} : (xs|k) \in P) \ . \end{array}$$

We now apply this to the algorithm. We write $init \subseteq X$ for the set of initial states and $step \subseteq X^2$ for the next state relation. Following [1], we use the convention that relation $step$ is reflexive (contains the identity relation). An *execution* is an infinite sequence of states that starts in an initial state and in which each subsequent pair of states is connected by a step. The set of executions of the algorithm is therefore

$$Ex = [\![init]\!] \cap \Box [\![step]\!]_2 \ .$$

If J is an invariant of the system, it holds in all states of every execution. We therefore have $Ex \subseteq \Box J$. An execution in which process p is always eventually at line 21, is an element of $\Box\Diamond [\![p \textbf{ at } 21]\!]$.

Remark. Note the difference between $\Box\Diamond [\![U]\!]$ and $\Diamond\Box [\![U]\!]$. In general, $\Box\Diamond [\![U]\!]$ is a bigger set (a weaker condition) than $\Diamond\Box [\![U]\!]$. The first set contains all sequences that are infinitely often in U, the second set contains the sequences that are from some time onward eternally in U. $\Box$

6.3 Weak fairness formalized

For a relation $R \subseteq X^2$, we define the *disabled* set $D(R) = \{x \mid \forall\, y : (x, y) \notin R\}$. Now *weak fairness* [16] for R is defined as the set of executions in which R is infinitely often disabled or taken:

$$wf(R) \quad = \quad Ex \cap \Box\Diamond ([\![D(R)]\!] \cup [\![R]\!]_2) \ .$$

For our algorithm, the next state relation $step \subseteq X^2$ is the union of the identity relation on X (because $step$ should be reflexive) with the relations $step(p)$ that consists of the state pairs (x, y) where y is a state obtained when process p does a step starting in x. In accordance with Section 3.9, the steps of process p are partioned as:

$$step(p) \quad = \quad free(p) \cup fwd(p) \cup low(p) \cup trig(p) \ ,$$

where $free(p)$, $fwd(p)$, $low(p)$, $trig(p)$ is the union of the free step relations, of the the forward step relations, of the lowering step relations, and of the triggered step relations, respectively. The set of triggered steps of p is again a union:

$$trig(p) = \bigcup_{q,m} rec(m,q,p) \cup \bigcup_{s,n} sit(n,p,s)\ ,$$

where $rec(m,q,p)$ consists of the steps where p receives message m from q. Note that we take the union here over all processes q and the eight message alternatives m, including the delayed answers `after` and `prom`, and $sit(n,p,s)$ consists of the four commands for message keys n between process p and site s.

The set $wf(fwd(p))$ consists of the executions for which every forward step of process p is infinitely often disabled or taken. Indeed, if we write $step(p,\ell)$ for the step relation of process p at line ℓ, we have $fwd(p) = \bigcup_{\ell=22}^{33} step(p,\ell)$ and it can be proved that

$$wf(fwd(p)) = \bigcap_{\ell=22}^{33} wf(step(p,\ell))\ .$$

Similarly, the set $wf(low(p))$ consists of the executions for which every lowering step of process p is infinitely often disabled or taken. Note, however, that $wf(fwd(p)) \cap wf(low(p))$ is a proper subset of $wf(fwd(p) \cup low(p))$, because the latter set contains (e.g.) an execution in which process p does infinitely many forward steps and ignores a continuously enabled lowering step at line 43.

The set $wf(rec(m,q,p))$ consists of the executions for which every message m in transit from q to p is eventually received.

An execution is defined to be *weakly fair for* process p if it is weakly fair for the forward steps of p, and for the lowering steps of p, and for all messages with p as destination or source, as captured in the definition

$$\begin{aligned} Wf(p) = {} & wf(fwd(p)) \cap wf(low(p)) \cap \textstyle\bigcap_{s,n} wf(sit(n,p,s)) \\ & \cap \textstyle\bigcap_{q,m}(wf(rec(m,q,p)) \cap wf(rec(m,p,q)))\ , \end{aligned}$$

where s ranges over the sites, n over the messages to and from sites, q over the processes, and m over the eight message alternatives between processes. Note that it is possible to impose and meet countably many weak (or strong) fairness conditions [10, Section 4].

6.4 Absence of localized starvation

As discussed in the Introduction, there are two progress properties to consider: starvation freedom, which means that, when every competing process will always eventually do a step, every competing process will eventually become idle again. *Concurrency* [4, 23] means that every competing process will eventually become idle again unless it comes in eternal conflict with an "unwilling" process that does not proceeed. In either case, the client's option to abort *entry* cannot be excluded, but it must not be needed for reaching the *idle* state. We formalize willingness as weak fairness.

For starvation freedom we assume weak fairness for all forward, lowering, and triggered steps of all processes. For concurrency for process p, we only need weak fairness for the forward and lowering steps of process p itself and for all triggered steps in which process p is involved. As we want to verify both properties with a single proof, we unify and generalize them in the concept of absence of localized starvation, which is similar to the absence of localized deadlock of Section 5.3.

Let W be any set of processes. We define *absence of W-starvation* to mean that weak fairness for all forward and lowering steps of the processes in W and for all triggered steps that involve processes in W, implies that every process in W eventually comes back to line 21 unless it comes in eternal conflict with some process outside W. The special case that W is the set of all processes is starvation freedom. The special case that W is a singleton set is concurrency [4, 23].

We speak of *absence of localized starvation* if absence of W-starvation holds for every set W. The aim is thus to prove that the algorithm satisfies absence of localized starvation. In order to do so, we formalize the definition in terms of temporal logic.

An execution in which process p comes always eventually back to line 21 is an element of $\Box\Diamond[\![p \textbf{ at } 21]\!]$. An execution where process q is eventually in eternal conflict with r is an element of $\Diamond\Box[\![q \bowtie r]\!]$. Let $cf(W)$ be the set of the executions in which some process $q \in W$ comes in eternal conflict with some process $r \notin W$:

$$cf(W) = \bigcup_{q \in W, r \notin W} \Diamond\Box[\![q \bowtie r]\!] \ .$$

Absence of W-starvation thus means that all "sufficiently fair" executions are elements of the set

$$\Box\Diamond[\![p \textbf{ at } 21]\!] \cup cf(W).$$

An execution is defined to be weakly fair for a set of processes W if it is weakly fair for each of them:

$$Wf(W) = \bigcap_{p \in W} Wf(p) \ .$$

Absence of localized starvation thus is the following result:

Theorem 2 *$Wf(W) \subseteq \Box\Diamond[\![p \textbf{ at } 21]\!] \cup cf(W)$ holds for every set W of processes and every process $p \in W$.*

6.5 Trajectories

The proof of Theorem 2 is given in Section 6.8. We first claim that a process that from some time onward does not become idle anymore, remains eternally at some line number $\neq 21$. More precisely, we have

Lemma 3 *Let p be a process. Let xs be an execution that is not in $\Box\Diamond[\![p \textbf{ at } 21]\!]$. Then there is a line number $\ell \in \{22 \ldots 33\}$ such that $xs \in \Diamond\Box[\![p \textbf{ at } \ell]\!]$.*

Proof. We use a so-called variant function to prove this in a systematic way. First note that $pc.p$ has values in the range $21 \ldots 33$. As we want to reach line 21, and all jumps are forward jumps except for the jumps to line 21 and the jump from line 32 to line 31, we define the state function $vfpc(p) : \mathbb{N}$ by

$$\begin{aligned} &vfpc(p) = vv(pc.p) \quad \text{where} \\ &vv(k) = (k = 21 \mathbin{?} 0 \ : \ k = 31 \mathbin{?} 1 \ : \ 35 - k) \ . \end{aligned}$$

The only transitions in which $vfpc(p)$ increases are those in which process p goes from line 21 to 22. These steps have the precondition $vfpc(p) = 0$.

As the execution xs is not an element of $\Box\Diamond[\![p \textbf{ at } 21]\!]$, there is a time n such that from time n onward all states of the execution satisfy $vfpc(p) > 0$. From n onward, $vfpc(p)$ can only decrease. Therefore, this function has a limit value. So, there is a value $k > 0$ and a time n' such that $vfpc(p) = k$ holds from time n' onward. There is a line number $\ell \in \{22 \ldots 33\}$ with $vv(\ell) = k$. We have $pc.p = \ell$ from time n' onward. $\Box$

6.6 Verification of progress

We prepare the proof of Theorem 2 by concentrating on what can be inferred from weak fairness for a single process. In view of Lemma 3, we first enumerate line numbers where a weakly fair process cannot stay eternally.

Lemma 4 *For any process p, the intersection $Wf(p) \cap \Diamond\Box[\![p \textbf{ at } \ell]\!]$ is empty for the line numbers $\ell =$ 43, 22, 23, 24, 27, 28, 30, 31, 32, 33.*

Proof. Let *xs* be an element of the intersection, i.e., an execution, weakly fair for p, in which eventually process p remains at line ℓ. As the forward step of line ℓ is disabled, we have $\ell \notin \{27, 28, 30, 33\}$.

Next assume that $\ell \in \{23, 32\}$. If process p remains at line 23 or 32, its variable *curlist*.p does not grow, and it shrinks when p receives a message `answer` from a site. As it is a finite set, this implies that, if the set *curlist*.p remains nonempty, there is a site s that remains in *curlist*.p forever. As the messages `asklist`.$p.s$ and the answers `answer`.$s.p$ are eventually received, predicate *Kq0* implies that r is eventually removed from *curlist*.p. This proves that *curlist*.p is eventually always empty. The forward step of line ℓ is therefore eventually always enabled. By weak fairness, this step will be taken, so that p do not remain at line ℓ. This proves that $\ell \notin \{23, 32\}$. Let us call the argument used in this paragraph the *shrinking argument* for the set *curlist*.p and the invariant *Kq0*.

Indeed, the same shrinking argument, but now for the set *reglist*.p and the invariant *Kq3*, gives that, if p remains at line 43, the finite set *reglist*.p is eventually always empty. Therefore the forward step of p is eventually always enabled, and by weak fairness p leaves line 43. This proves that $\ell \neq 43$.

If p remains at line 22, by weak fairness, its forward step of line 22 is always eventually disabled. Therefore, p is always eventually at line 43. By the previous paragraph, however, it eventually leaves line 43, but then by the guard of line 42, it cannot enter line 43 again. This proves that $\ell \neq 22$.

If p remains at line 24 or 31, the finite set *pack*.p eventually becomes empty because of the shrinking argument with the invariant *Kq2*. It follows that process p cannot remain at line 31. This proves that $\ell \neq 31$.

It remains to consider the case that p remains at line 24. By the previous paragraph, the set *pack*.p is eventually always empty. In order to show that p leaves line 24, it suffices to show that p is eventually always enabled. For this purpose, it suffices to show that *wack*.p is eventually always empty. At line 24, the set *wack*.p never grows, and it can shrink by reception of messages `ack`. If it does not shrink to the empty set, there is a process r with $r \in$ *wack*.p eventually always. We now use predicate *Waq0* of Section 5.2. As process p remains at line 24, it will not send any messages `withdraw` and `notify` to r. From some time onward, we therefore have `notify`.$p.r = \bot$ and `withdraw`.$p.r = 0$ and `welcome`.$p.r \in \{\bot, none\}$. Therefore, by *Waq0*, $\neg dAfter(r, p) \ \vee$ `ack`.$r.p > 0$ holds henceforward. By weak fairness for its triggered step `after`, process r will send `ack` to p. This `ack` will be received and process p will remove r from *wack*.p. This is a contradiction, proving that $\ell \neq 24$. □

A process can be forced to remain at line 25 by coming in some eternal conflict:

Lemma 5 *Let xs* $\in$ *Wf*(p) $\cap \Diamond\Box[\![\, p$ ***at*** $25\,]\!]$. *Then there is a process q with xs* $\in \Diamond\Box[\![\, q$ ***in*** $\{26 \ldots 29\} \wedge p \bowtie q\,]\!]$.

Proof. The sequence *xs* is an execution, weakly fair for p, in which, from some time n_0 onward, process p is and remains at line 25. As the execution is weakly fair for p, by the shrinking argument, there is a process q such that, from time n_0 onward, $q \in$ *prio*.p always holds. If `withdraw`.$q.p > 0$ holds at some time $n \geq n_0$, by weak fairness this message will eventually arrive and falsify $q \in$ *prio*.p. Therefore, `withdraw`.$q.p = 0$ holds from time n_0 onward. By the above argument, there is a time $n_1 \geq n_0$ such that `welcome`.$p.q \in \{\bot, none\}$ holds from time n_1 onward. By *Waq1*, q is in $\{26 \ldots 29\}$ and $p \bowtie q$ holds from time n_1 onward. □

6.7 Progress at lines 26 and 29

It remains to consider an execution, weakly fair for process p in which eventually p remains at line 26 or 29. If we would treat the line numbers 26 and 29 separately, line

29 is somewhat simpler than line 26. We prefer, however, to treat them together.

Process p waits at line 26 or 29 for emptiness of *need* or of a subset of it. This condition belongs to the inner protocol. The inner protocol in isolation, however, is not starvation free because it would allow a lower process repeatedly to claim priority over p by sending notifications. We need the outer protocol to preclude this. Technically, the problem is that *need.p* can grow at lines 26 and 29 because of the alternative `prom`.

We therefore investigate conditions under which the predicate $q \in \mathit{need}.p$ or its negation is stable, i.e., once true remains true. As p remains at line 26 or 29, the set *nbh0.p* remains constant. We define the predicate

$$\mathit{bf}(q,p) : \neg(q \in \mathit{nbh0}.p \wedge q \textbf{ in} \{25 \ldots 30\}) \vee \mathit{job}.p * \mathit{job}.q \vee p \in \mathit{prio}.q \; .$$

Roughly speaking, $\mathit{bf}(q,p)$ expresses that q is not in the inner protocol or is not in conflict with p. While process p is and remains at line 26 or 29 and $\mathit{copy}.q(p) \neq \mathit{none}$, the predicate $\mathit{bf}(q,p)$ is stable. The main point is when process q executes line 24. If the second disjunct of $\mathit{bf}(q,p)$ does not hold, process q puts p into *prio.q*. The proof uses *Iq2a*, *Iq4*, *Iq7*, *Iq8*, and *Rq1a*.

While process p is and remains at line 26 or 29 and either $\mathit{bf}(q,p)$ holds or $p < q$, the predicate $q \notin \mathit{need}.p$ is stable. This is proved at the alternative `prom` with *Iq4*, *Jq0*, *Jq1*, *Jq2*, and *Lq2*.

While process p is and remains at line 26 or 29 and $\mathit{bf}(q,p)$ is false and $q \leq p$, the predicate $q \in \mathit{need}.p$ is stable. This is proved at the alternatives `gra` and `withdraw` with *Iq2a*, *Jq5*.

We can now combine these predicates in the variant function

$$\begin{array}{rcl} \mathit{vf}(q,p) & = & (\mathit{bf}(q,p)\,? \quad |\, q \in \mathit{need}.p\,| \\ & & :\; q \in \mathit{need}.p\,? \quad 3 \\ & & :\; p < q\,? \quad 2 \,:\; 4)\;. \end{array}$$

Clearly, $\mathit{vf}(q,p)$ is odd (1 or 3) if and only if $q \in \mathit{need}.p$ holds. Similarly, $\mathit{vf}(q,p) \leq 1$ holds if and only if $\mathit{bf}(q,p)$.

The above stability results about $\mathit{bf}(q,p)$ and $q \in \mathit{need}.p$ and its negation imply that, while p is and remains at line 26 or 29 and $\mathit{copy}.q(p) \neq \mathit{none}$, the function $\mathit{vf}(q,p)$ never increases. By weak fairness, eventually $\texttt{notify}.p.q = \bot$ holds. As process p remains at line 26 or 29, $\texttt{notify}.p.q = \bot$ remains valid. The invariant *Iq4* together with *Iq5* and *Iq1* therefore implies that $\mathit{copy}.q.(p) \neq \mathit{none}$ holds and remains valid. Therefore, from that time onward, $\mathit{vf}(q,p)$ never increases.

It follows that, for every $q \in \mathit{nbh0}.p$, eventually, $\mathit{vf}(q,p)$ gets a constant value. Therefore, the truth value of $q \in \mathit{need}.p$ is also eventually constant. Finally, as *need.p* is always a subset of the finite set *nbh0.p*, which is constant while p is at line 26 or 29, we can now conclude that the set *need.p* is eventually constant.

If the set *need.p* is eventually empty, process p would be eventually always enabled. Weak fairness of p would then imply that process p would leave line 26 or 29. Therefore, there is some process q eventually always in *need.p*. We have $q \neq p$ because of *Jq0* and *Iq0*. This proves

(1) $$\ell \in \{26, 29\} \;\Rightarrow\; \mathit{Wf}(p) \cap \Diamond\Box[\![\, p \textbf{ at } \ell \,]\!] \subseteq \textstyle\bigcup_{q \neq p} \Diamond\Box[\![\, q \in \mathit{need}.p \,]\!] \; .$$

Assume that $q \in \mathit{need}.p$ holds from time n_0 onward. We now make a case distinction. First assume $q < p$. If $\texttt{withdraw}.q.p > 0$ holds at some time $n \geq n_0$, weak fairness implies that the message `withdraw` will be received at some time $> n_0$, which would falsify $q \in \mathit{need}.p$. Therefore, $\texttt{withdraw}.q.p = 0$ holds from time n_0 onward. By predicate *Waq2*, we thus have $q \textbf{ in } \{26 \ldots 29\}$ and $p \bowtie q$ from time n_0 onward. This proves

(2) $$q < p \;\Rightarrow\; \mathit{Wf}(p) \cap \Diamond\Box[\![\, q \in \mathit{need}.p \,]\!] \subseteq \Diamond\Box[\![\, q \textbf{ in } \{26 \ldots 29\} \wedge p \bowtie q \,]\!] \; .$$

Next, assume $p < q$. If $\texttt{gra}.q.p > 0$ holds at some time $n \geq n_0$, this $\texttt{gra}$ message will be received because of weak fairness, falsifying $q \in \mathit{need}.p$. Therefore, $\texttt{gra}.q.p = 0$ holds from time n_0 onward. Also by weak fairness, we have $\texttt{notify}.p.q = \bot$ at some time $n_1 \geq n_0$. Because process p is and remains at line 26 or 29, it cannot send such messages again. Therefore, $\texttt{notify}.p.q = \bot$ holds from time n_1 onward. Because process q sends no $\texttt{gra}$ message to p after n_0, weak fairness implies that $\mathit{dProm}(q,p)$ holds infinitely often after n_1. By *Waq3* this implies that, infinitely often, we have

$$\mathit{bg}(q,p): \quad q \textbf{ in } \{27, 28\} \ \wedge \ p \in \mathit{nbh}.q \ \wedge \ p \bowtie q \ .$$

We need the condition $\mathit{bg}(q,p)$ eternally, however, not only infinitely often. For this purpose, we reuse that $\mathit{vf}(q,p)$ is eventually constant, say that $\mathit{vf}(q,p) = k$ holds from time $n_2 \geq n_1$ onward. As condition $\mathit{bg}(q,p)$ contradicts $\mathit{bf}(q,p)$, we have $k > 1$. Therefore, $\mathit{bf}(q,p)$ is false from time n_2 onward. This implies that process q does not execute line 28 from time n_2 onward. Therefore, condition $\mathit{bg}(q,p)$ holds eventually always. Consequently, we obtain

$$(3) \qquad p < q \ \Rightarrow \ \mathit{Wf}(p) \cap \Diamond\Box[\![\, q \in \mathit{need}.p \,]\!] \subseteq \Diamond\Box[\![\, q \textbf{ in } \{27, 28\} \wedge p \bowtie q \,]\!] \ .$$

We now combine these results as follows:

Lemma 6 *Let W be a set of processes. Let $p \in W$ and $\ell \in \{26, 29\}$. Let $\mathit{xs} \in \mathit{Wf}(W) \cap \Diamond\Box[\![\, p$ **at** $\ell \,]\!]$. Then $\mathit{xs} \in \mathit{cf}(W)$.*

Proof. Because W contains some process that remains eternally at ℓ, we can consider the lowest process in W, say q, with this property. By formula (1), there is a process $r \neq q$ that remains eternally in $\mathit{need}.q$. If $q < r$, formula (3) implies that r remains eternally in $\{27, 28\}$ and in conflict with q. By Lemma 4, it follows that $r \notin W$ and hence that $\mathit{xs} \in \mathit{cf}(W)$. On the other hand, if $r < q$, formula (2) implies that r remains eternally in $\{26 \ldots 29\}$ and in conflict with q. If $r \in W$, minimality of q implies that eventually process r is not eternally in $\{26, 29\}$, and therefore by Lemma 3, it is eventually always in $\{27, 28\}$, contradicting Lemma 4. Therefore $r \notin W$, and $\mathit{xs} \in \mathit{cf}(W)$. $\Box$

6.8 The proof of Theorem 2

Consider an execution $\mathit{xs} \in \mathit{Wf}(W) \setminus \mathit{cf}(W)$. Let $p \in W$. The Lemmas 3, 4, and 6 together imply that

$$(4) \qquad \mathit{xs} \in \Box\Diamond[\![\, p \textbf{ at } 21 \,]\!] \cup \Diamond\Box[\![\, p \textbf{ at } 25 \,]\!] \ .$$

Assume that $\mathit{xs} \in \Diamond\Box[\![\, p \textbf{ at } 25 \,]\!]$. Lemma 5 then gives us a process q with $\mathit{xs} \in \Diamond\Box[\![\, q \textbf{ in } \{26 \ldots 29\} \wedge p \bowtie q \,]\!]$. By Lemma 3, there is $\ell \in \{26 \ldots 29\}$ with $\mathit{xs} \in \Diamond\Box[\![\, q \textbf{ at } \ell \,]\!]$. By formula (4), we have $q \notin W$ and hence $\mathit{xs} \in \mathit{cf}(W)$. This contradicts the assumption, thus proving that $\mathit{xs} \in \Box\Diamond[\![\, p \textbf{ at } 21 \,]\!]$. This concludes the proof of Theorem 2.

6.9 Progress for lowering

Lemma 4 gives progress for the lowering step at line 43, under the assumption of weak fairness. Progress at line 42, however, requires more than weak fairness. While process p is at line 42, the value of $\mathit{pc}.p$ can repeatedly enter and leave the set $\{22, 23, 32\}$. Therefore, the step of line 42 is not eventually always enabled. Indeed, we do not want that resource acquisition suffers for lowering.

We need strong fairness for progress at line 42. Strong fairness at line 42 means that, if process p is infinitely often enabled at line 42, it will eventually take the step of line 42. Formally, for a relation $R \subseteq X^2$, *strong fairness* [16] for R is defined as the set of executions in which R is eventually always disabled or infinitely often taken:

$$sf(R) \quad = \quad Ex \cap (\Diamond\Box[\![\, D(R) \,]\!] \cup \Box\Diamond[\![\, R \,]\!]_2) \ .$$

If, in some execution, process p is always eventually at line 21, and yet remains at line 42, the step of line 42 is infinitely often enabled, so that indeed strong fairness guarantees that the step will be taken. In combination with Lemma 4, we thus obtain progress for the lowering thread in the sense that it always returns to its idle state at line 41:

$$\Box\Diamond[\![\, p \textbf{ at } 21 \,]\!] \cap Wf(p) \cap sf(step(p, 42)) \subseteq \Box\Diamond[\![\, p \textbf{ at } 41 \,]\!] \ ,$$

where $step(p, 42)$ is the relation corresponding to the step of p at line 42.

7 Message Complexity and Waiting Times

In the central algorithm, a process exchanges 3 or 4 messages with every neighbour. In the querying phase, at lines 22 and 23, it exchanges 2 messages with every site it is interested in, plus 2 messages for every potential competitor.

In a message passing algorithm, we can distinguish two kinds of waiting. There is waiting for answers that can be sent immediately. Such waiting requires at most 2Δ, because Δ is an upper bound of the time needed for the execution of an alternative plus the time the messages sent are in transit. This happens at line 23, for emptiness of *curlist*, and at line 24, for emptiness of *pack* and *wack*. In the case of concurrent lowering, the waiting at line 22 is, via line 43, also of this kind.

The other kind of waiting is when a process needs to wait for the progress of other processes. These are the important waiting conditions. The central algorithm has two locations where this happens. At line 25, the process waits for emptiness of *prio* to make accumulation of conflicting processes unlikely. At line 26, it waits for emptiness of *need* to ensure partial mutual exclusion.

The waiting time T_1 for emptiness of *need* at line 26 depends on the conflict graph of the processes that are concurrently in the inner protocol. The middle layer tries to keep this graph small by guaranteeing that conflicting processes do not enter the inner protocol concurrently unless they pass line 25 within a period Δ.

The waiting time T_2 for emptiness of *prio* in line 25 depends on the efficiency of the inner protocol, because for a process p the elements of *prio.p* are in the inner protocol and are removed from *prio.p* when they withdraw. We thus have $T_2 \leq T_1 + \Gamma + \Delta$ where Γ is an upper bound for the time spent in *CS*. Indeed, every element of *prio* arrives in *CS* after time T_1, and at line 28 after $T_1 + \Gamma$, while the message `withdraw` takes time Δ. It follows that the total time for the main loop body is at most $6\Delta + T_2 + T_1 + \Gamma \leq 2T_1 + 2\Gamma + 7\Delta$.

The value of T_1 heavily depends on the load of the system and other system parameters: for resource c, the number of jobs activated per Γ that need resource c; the number of resources per job; the number of resources per site; the number of processes per site; the number of conflicts per job. Of course, all these numbers should be averages, and they are not independent. Experiments are needed to evaluate the performance of the algorithm, and to compare it with other algorithms.

8 Related research

The readers/writers problem [2] goes back to Courtois, Heymans, and Parnas [6], in the context of shared memory systems with semaphores. We are not aware of solutions for systems with message passing.

In the drinking philosophers' problems of [4, 19, 25], the philosophers are the nodes of a fixed finite undirected graph with bottles attached to the edges. The job of process p is always a subset of the set of bottles on its incident edges, chosen nondeterministically when the process becomes "thirsty". When it becomes thirsty, a process p needs to communicate with all members of its neighbourhood $nbh.p$ in the graph. The message complexity is therefore proportional to the size of $nbh.p$. This is a disadvantage for cases with large complete graphs. To enforce starvation freedom, these solutions assign directions to the edges of the graph such that the resulting directed graph is acyclic. They more or less ignore the information contained in the jobs, which is heavily used by our algorithm.

Much work has been done to minimize the response time [3, 5, 22, 23, 25]. For instance, the paper [25] offers the possibility of a waiting time that is constant and not proportional to the (in our case unbounded) number of processes. It does so by means of the algorithm of [18], which uses a linear ordering of the resources adapted to a fixed netwerk topology. The papers [23, 25] offer modular approaches to the general resource allocation problem.

Another important performance aspect is robustness against failures. The paper [5] introduces the measure of failure locality, see also [24]. The paper [7] concentrates on self-stabilization, while imposing specific conditions on the resource requirements.

As far as we can see the only papers that treat the dynamic resource allocation problem that allows conflicts between arbitrary pairs of processes are [3, 23]. The algorithm of [3] ignores the resources and takes the conflicts as given. Whenever two processes have conflicting jobs, at least one of the two is activated with knowledge of the conflict. The emphasis of [3, 23] is on minimizing the response time. The algorithms are more complicated and need more messages than our solution. The paper [23] uses resource managers.

9 Conclusions

The problem of distributed resource allocation is a matter of partial mutual exclusion, with the partiality determined by a dynamic conflict graph. Our solution allows potentially infinitely many processes, and it allows conflicts between every pair of processes. The primary disentanglement is the split into the central algorithm and the registration algorithm.

In the central algorithm, the conflict graph is dynamic but limited by the current registrations, and the jobs can be treated as uninterpreted objects with a compatibility relation. The central algorithm itself consists of three layers. In the outer protocol, the processes communicate their jobs. In the inner protocol they compete for the critical section. The middle layer protects the inner protocol from flooding with conflicting processes.

The neighbourhoods used in the central algorithm are formed in a registration phase in which the processes communicate with a finite number of sites. We use a flexible job model that allows e.g. the distinction between read permissions and write permissions. We reach a fully dynamic conflict graph by enabling the processes to modify their registrations.

Our solution does not automatically satisfy the "economy" condition of [4] that permanently tranquil philosophers should not send or receive an infinite number of messages. Indeed, in our algorithm, a permanently idle process that occurs infinitely often in the neighbourhood of other processes will receive and send infinitely many messages. It can avoid this, however, by lowering its registrations to zero. It is true, however, that a process that has never been competing, never receives messages.

Our solution is more concurrent than the layered solution of [25]. It satisfies the requirement that, "if a drinker requests a set B of bottles, it should eventually enter its critical region, as long as no other drinker uses or wants any of the bottles in B forever" ([25, p. 243]).

Our algorithm does not minimize the response time. Yet, it may perform reasonably well in this respect, because the middle layer of the central algorithm prohibits entrance for new processes that have known conflicts with processes currently in the inner protocol. In the inner protocol, the lower processes have the advantage that they can force priority over higher conflicting processes. When conflicts in the inner protocol are rare, however, this bias towards the lower processes will not be noticeable.

The algorithm as presented allows several simplifications. (1) The aborting commands of Section 3.6 can be removed. (2) One can decide to give every resource its own site, or to use a single site for all resources. (3) If one takes the simplest job model, i.e. $K = 1$ in Section 2.4, the arrays *fun*, *news*, and *list* reduce to finite sets. (4) One can fix the network topology, i.e., replace the variables *nbh.p* by constants, and remove the registration algorithm. (5) If the set *Proc* of all processes is finite and rather small, one can even take *nbh.p* = *Proc* for all p.

The algorithm could not have been designed without a proof assistant like PVS. This holds in particular for the proofs of safety of registration (the invariants *Mq**), the proof of progress of the central algorithm (Section 6.7), and the use of array *copy* in the registration algorithm to avoid additional waiting.

In the mechanical proof, we summarize the argument for safety by forming the conjunction of the universal quantifications of the predicates of the families *Iq**, *Jq**, *Kq**, *Lq**, *Mq**, *Nq** (the so-called constituent invariants). As verified mechanically, this conjunction is inductive. Such mechanical verification is relevant, because with more than 40 invariants, the probability of overlooking an unjustified assumption or a clerical error is significant. As each of the constituent invariants is a consequence of the conjunction, each of them is itself invariant, as are all logical consequences of them. In particular, the mutual exclusion predicate *Rq0* is invariant. One may wonder whether the constituent invariants are independent. We do believe so, but we have no suitable way to verify this.

Future research. It would be interesting to see how much the algorithm can be simplified by using reliable synchronous messages. It will be quite a challenge to make the algorithm robust by allowing the asynchronous messages to be lost or duplicated. In our algorithm, a process that has ever been competing must eternally be willing to receive messages. It should be possible to offer the processes the option to (perhaps temporarily) quit from the algorithm, with the guarantee that there will be no messages in transit to a process that has quitted. Here, the difficulty is the last communication between two quitting processes.

Acknowledgement. The observation that the algorithm also solves the readers/writers problem was made by Arnold Meijster.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.

[2] G.R. Andrews. *Foundations of multithreaded, parallel, and distributed Programming.* Addison Wesley, Reading, etc., 2000.

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

[4] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):623–646, 1984.

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

[6] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14:667–668, 1971.

[7] A.K. Datta, M. Gradinariu, and M. Raynal. Stabilizing mobile philosophers. *Inf. Process. Lett.*, 95:299–306, 2005.

[8] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[9] E.W. Dijkstra. A strong P/V-implementation of conditional critical regions. Tech. Rept., Tech. Univ. Eindhoven, EWD 651,
see `www.cs.utexas.edu/users/EWD`, 1977.

[10] W.H. Hesselink. Splitting forward simulations to cope with liveness. *Acta Inf.*, 42:583–602, 2006.

[11] W.H. Hesselink. Partial mutual exclusion for infinitely many processes. Submitted. See http://arxiv.org/abs/1111.5775, 2011.

[12] W.H. Hesselink. Distributed resource allocation.
`http://wimhesselink.nl/mechver/distrRscAlloc`, 2012.

[13] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[14] G.J. Holzmann. *The SPIN Model Checker, primer and reference manual.* Addison-Wesley, 2004.

[15] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17:453–455, 1974.

[16] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.

[17] E.A. Lycklama and V. Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Trans. Program. Lang. Syst.*, 13:558–576, 1991.

[18] N.A. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal of Computer and System Sciences*, 23:254–278, 1981.

[19] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufman, San Francisco, 1996.

[20] J. Misra. *A discipline of multiprogramming: programming theory for distributed applications.* Springer, New York, 2001.

[21] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001. `http://pvs.csl.sri.com`

[22] I. Page, T. Jacob, and E. Chern. Fast algorithms for distributed resource allocation. *IEEE Trans. Parallel and Distributed Systems*, 4:188–197, 1993.

[23] I. Rhee. A modular algorithm for resource allocation. *Distr. Comput.*, 11:157–168, 1998.

[24] P.A.G. Sivilotti, S.M. Pike, and N. Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. Ohio State University, 2000.

[25] J.L. Welch and N.A. Lynch. A modular drinking philosophers algorithm. *Distr. Comput.*, 6:233–244, 1993.