THE DRIVING PHILOSOPHERS[*]

S. Baehni
Distributed Programming Laboratory
EPFL, Switzerland

R. Baldoni
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”, Italy

R. Guerraoui
Distributed Programming Laboratory
EPFL, Switzerland

B. Pochon
Distributed Programming Laboratory
EPFL, Switzerland

Abstract We introduce a new synchronization problem in mobile ad-hoc systems: the Driving Philosophers. In this problem, an unbounded number of driving philosophers (processes) access a round-about (set of shared resources organized along a logical ring). The crux of the problem is to ensure, beside traditional mutual exclusion and starvation freedom at each particular resource, gridlock freedom (i.e., a cyclic waiting chain among processes). The problem captures explicitly the very notion of process mobility and the underlying model does not involve any assumption on the total number of (participating) processes or the use of shared memory, i.e., the model conveys the ad-hoc environment. We present a generic algorithm that solves the problem in a synchronous model. Instances of this algorithm can be fair but not concurrent, or concurrent but not fair. We derive the impossibility of achieving fairness and concurrency at the same time as well as the impossibility of solving the problem in an asynchronous model. We also conjecture the impossibility of solving the problem in an ad-hoc network model with limited-range communication.

[*]The work presented in this paper was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322 and by the Federal Office for Education and Science (OFES) for the PALCOM IST project (Framework VI), under grant number 03.0495-1. Roberto Baldoni has been partially supported by the ministry of the italian universities and research (MIUR) in the context of the project IS-MANET.

Introduction

Whilst 98% of the computers in the world are embedded devices, most research on synchronization is done with the 2% left in mind [7]. One possible reason might be the lack of precisely defined problems for the former case.

In 1971, Dijkstra introduced an intricate synchronization paradigm, the Dining Philosophers problem [6]. The problem crystallizes the difficulty of accessing shared resources, by posing orthogonal constraints, in terms of mutual exclusion, starvation-freedom, and deadlock-freedom. In Dijkstra's problem, the number of processes (i.e., philosophers) is known, as well as the arrangement of processes. Hence the pairs of processes in which conflicts may appear are known. Variants of this problem, in particular the Drinking Philosophers [5], traditionally make the same assumptions.

The motivation behind the Driving Philosophers is to define a problem that crystallizes the difficulty of accessing shared resources amongst mobile processes that communicate through ad-hoc networks. The Driving Philosophers problem was inspired by the practical issue of synchronizing cars in a round-about. Like in the Dining Philosophers, asynchronous processes compete on a set of resources. Unlike in the Dining Philosophers however, the processes do so (a) without a priori knowing the number of participating processes, how many resources they might require, nor how many are available, (b) following a specific order amongst the resources that the processes request (i.e., the resources model the portions of the road in the round-about; the processes are in this sense mobile), and (c) in a system model with no shared memory or any communication medium which would make it possible to reach all processes in the system (ad-hoc network).

In this paper we first precisely define the Driving Philosophers problem. We then give a generic canvas to solve the problem. By instantiating the generic canvas with a set of predicates, we present different modular solutions to the Driving Philosophers in a synchronous model. Synchrony assumptions can be met in practice assuming a typical wireless network, and processes equipped with local GPS receivers. The genericity of our approach allows for investigating several algorithmic flavors. In particular, we introduce the notions of *concurrent* and *fair* algorithms. Roughly speaking, a concurrent algorithm is one that does not deny concurrent accesses to distinct resources, whereas a fair algorithm grants requests following the arrival time. In a precise sense, we show that concurrency and fairness are two antagonistic notions.

We also show that even if no failure is allowed, the Driving Philosophers problem is impossible without assumptions on communication delays and process relative computation speeds (asynchronous model), or specific assumptions on space or arrival rate of participating processes. We also conjecture the impossibility of solving the Driving Philosophers in a synchronous model in which communication is local, i.e., a model in which processes may communicate only using a restricted communication range. We give a proof of this conjecture in a restricted case, and leave the generalization open.

The rest of the paper is organized as follows: In Section 1, we first introduce some basic terminology, then the Driving Philosophers specification. In Section 2, we give our generic canvas solving the Driving Philosophers in a synchronous model. We instantiate our canvas with three different sets of predicates, and introduce our notion of concurrency. In Section 3, we introduce our notion of fairness, and give a new algorithm that complies with this notion. We prove then that concurrency and fairness are antagonistic. In Section 4, we prove the impossibility of solving the Driving Philosophers in the asynchronous model, and we conjecture the impossibility of solving the problem in a model with only limited-range communication. We prove this conjecture in a restricted case. In Section 5, we discuss possible variants of our problem, and present some related works. For space limitations, we postpone all proofs to a companion technical report [2].

1 The Driving Philosophers

Definitions

Processes. We consider a set of processes (philosophers) $\Omega = \{p_0, p_1, \ldots\}$. No process is a priori required to take part in the Driving Philosophers problem. More precisely, we consider that the processes take part to the problem in an uncoordinated manner (i.e, a process may be leaving the problem while another process simultaneously joins the problem). We denote by *participating* processes the set of processes which take part in the problem *at a specific point in time*. Note that the set of participating processes typically changes over time, e.g., when new processes take part in the problem. Every process has a unique identity. Processes communicate by message-passing using the primitives send and receive. The primitive send allows a process to send a message to the current participating processes, whereas the primitive receive allows a process to receive a message sent to it, that it has not yet received. Communication is reliable in the following sense: (*validity*) if a correct process sends a message to a correct process, the message is eventually received, (*no duplication*) each message is received at most once, and (*integrity*) the network does not create nor corrupt messages.

Resources. We consider a set of k resources $\Theta = \{r_0, r_1, \ldots, r_{k-1}\}$. Resources are organized in our case along a ring: $r_{i \oplus 1}$ follows r_i, where $a \oplus b$ (resp. $a \ominus b$) is defined as $(a + b)$ *mod* k (resp. $(a - b)$ *mod* k). Processes ignore the number of resources. Access to any resource may only take place within a *critical section* of code [6]. Before and after executing the critical section of code, any process executes two other fragments of code, respectively the *entry* and *exit* sections. Our problem is to design entry and exit sections, in order to adequately schedule the accesses to resources. A process is mobile in the sense it may request and access different resources at different times. We consider that the entry (resp. exit) section for resource r_s is invoked by process p_i using the primitive *entry*(i, s) (resp. *exit*(i, s)). When a process invokes a procedure for an entry or exit section, this process blocks until the procedure returns. We say that a resource r_s is *requested* by p_i upon the invocation of *entry*(i, s), *granted* to p_i upon returning from *entry*(i, s), and *released* by p_i upon the invocation of *exit*(i, s). We say that a process p_i *owns* a resource r_j at time t if there exists an invocation *entry*(i, s) which returns before time t, such that no invocation *exit*(i, s) occurs between the invocation of *entry*(i, s) and time t. Note that a process may own a resource for a finite but arbitrarily long period of time before releasing it (i.e., it is a "philosopher" in the sense that it may "think" for arbitrarily long).[1] We say that a process p_i is *new*, if p_i does not own any resource prior to invoking *entry*(i, s), for some resource r_s. At any point in time, at most one new process p_i may be requesting a resource r_s. The interaction between a process and its *entry* and *exit* sections are illustrated in Figure 1.

Problem

The Driving Philosophers problem is defined for a set of processes and a set of resources. Informally, any process which takes part in the problem has to access an *ordered* sequence of resources, starting from any resource, such that any resource is accessed by at most a single process at any time. Formally, an algorithm solves the Driving Philosophers problem if, for each of its execution, the following properties hold:[2]

(P1) (Mutual exclusion) No two processes own the same resource at the same time.
(P2) (No starvation) Any requested resource is eventually granted.

Processes are assumed to well behave in the sense that they respect the following conditions.

(B1) A process may request a resource r_s only if it *(i)* owns $r_{s\ominus 1}$ or *(ii)* does not own any resource.
(B2) After releasing every resource it owns, no process ever requests a resource.
(B3) If any process obtains every resource it requests, it eventually releases any resource it owns.

Property B1 defines the *ordering* relation among resources. Property B2 denotes the fact that a process may only take part in the problem at most once. Property B3 ensures that every process eventually releases every resource it owns.

We note that a traditional mutual exclusion algorithm, used to access each resource separately, will ensure properties P1, but may fail to ensure P2. The problem that may arise is *gridlock*, i.e., a situation in which (1) every resource is owned by a process, (2) every process would like to acquire the next resource, and (3) no process releases its current resource (i.e., no process desires to leave the round-about). We explain the gridlock problem in more details in the next paragraph.

Driving versus Dining Philosophers

Our problem differs in several aspects from the Dining Philosophers. Due to the mobility assumption, every process in the Driving case competes for different resources at different times. This fundamentally differs from the Dining case, in which each process repeatedly competes for a single critical section. The processes request resources following a specific order in the Driving case.

[1] Note that this is different from the speed of the cars, on which we make no assumption.
[2] Following [1, 14], our problem specification is broken into safety and liveness properties, as well as well-behaviorness of processes.

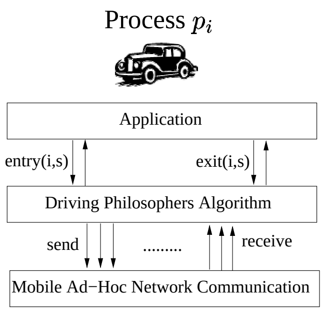
Process p_i
Application
entry(i,s)
exit(i,s)
Driving Philosophers Algorithm
send
....
receive
Mobile Ad−Hoc Network Communication

Figure 1. Component layout at process p_i

The major impact of considering mobile processes is the possibility of gridlock. Interestingly this case cannot occur in the Dining case because accessing a critical section first necessitates to acquire both adjacent tokens, which prevents two adjacent processes to access their critical section simultaneously. On the other hand, in the Dining case, processes may deadlock, if every process has acquired the left token and is waiting on the right one to be released. There is no such risk of a deadlock in the Driving case, because two simultaneous accesses to two adjacent resources are not directly conflicting.

In other words, the main difference between the Driving Philosophers problem and the Dining Philosophers lies in the fact that conflicts are not always between the same processes in the Driving case (processes are mobile). One may see the Dining Philosophers as resource-driven (resources are "applied" on a set of processes), whereas the Driving Philosophers is process-driven (processes are "applied" on a set of resources).

2 A Generic Algorithm

A generic algorithm solving the Driving Philosophers problem is presented in this section. We design this algorithm with the analogy between the Driving Philosophers and a round-about in mind, as shown in Figure 2. In this sense we assume that any process p_i which takes part in the problem invokes the entry and exit section procedures in such a way that p_i releases resource r_s, i.e., invokes $\mathit{exit}(i, s)$ before requesting $r_{s\oplus 2}$ (if p_i ever requests $r_{s\oplus 2}$). In this way, any process holds at most two resources at a time. This is an assumption on process well behavior, which could be described together with properties B1, B2 and B3. As such, the algorithm presented in Figure 3 solves a constrained variant of the Driving Philosophers problem.

System Model. We consider a synchronous model,[3] where there exists a known bound on (i) the time it takes for a process to execute a step, and (ii) on the message propagation delay. Computation proceeds in a round-based manner, i.e., processes interact in a synchronous, round-based computational way [14].[4] Roughly speaking, in each synchronous round, every process goes through three phases: in the first

[3]Mobile devices can typically be equipped with a GPS receiver that provides them with the synchrony assumption.
[4]Note that philosophers are still "asynchronous thinkers".

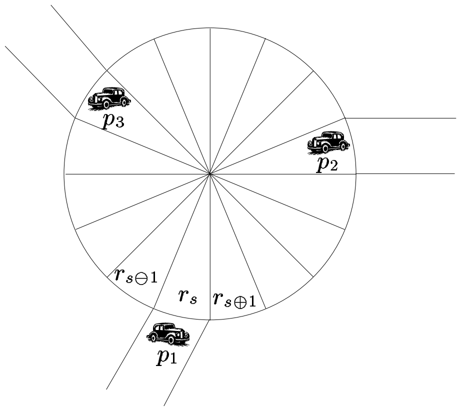

Figure 2. Analogy of the Driving Philosophers problem with a round-about

(send) phase, the process sends a message to the participating processes; in the second (receive) phase, the process receives all messages sent to it; in the third (computation) phase, the process computes the message to send in the next round. Compared with [14], our model differs in the sense that the set of participating processes (in a given round) is not necessarily the whole set of processes, not even necessarily the set of processes that ever take part to the problem.

Configurations and Runs. A *configuration* is an instantaneous cut of the state of the system at the end of a round. Roughly speaking it represents the state of resources and processes participating in the problem at the end of a round. More precisely, a configuration of the system at the end of round r is a tuple $C = \langle \textit{Waiting}, \textit{Driving} \rangle$. *Waiting* $: \Theta \rightarrow 2^{\Omega}$ is a function which gives information about processes in their trying state at the end of round r: for any resource $r_s \in \Theta$, $\textit{Waiting}(s)$ is the set of processes in the entry section for r_s,[5] and is $\emptyset$ if no process has requested this resource. *Driving* $: \Theta \rightarrow \Omega \cup \{\bot\}$ is a function which gives information about resources that are occupied at the end of round r: for any resource $r_s \in \Theta$, $\textit{Driving}(s)$ is the process that owns r_s in C, or $\bot$ if no process owns r_s. A *run* R is a (possibly infinite) sequence of configurations, ordered according to global time, starting from some initial configuration C. We say that a configuration $C = \langle \textit{Waiting}, \textit{Driving} \rangle$ is *gridlocked* if $\forall r_s \in \Theta : \textit{Driving}(s) \neq \bot$.

The Canvas. We first give a generic canvas for the Driving Philosophers problem in Figure 3. Key to this canvas is a predicate that defines when processes are allowed to effectively access a resource. We instantiate this canvas to various algorithms: each algorithm A corresponds to a predicate $\textit{pred}(A)$. The description of the canvas is divided between the mechanisms ensuring mutual exclusion and starvation freedom.

As far as mutual exclusion is concerned, any process maintains two local sets *pendingRequests.init* and *pendingRequests.transit* of pending requests of processes,

[5]This may represents at most two processes: one in the round-about, transiting through this resource, and one new process.

```
Critical section procedures for $p_i$:
1:  $init := true$
2:  $pendingRequests := (\emptyset, \emptyset)$        {$(pendingRequests.init, pendingRequests.transit)$}
3:  $starvers := (\emptyset, \emptyset)$        {$(starvers.init, starvers.transit)$}
4:  $resources := \emptyset$
5:  $acquiring := entryRound := \bot$
6:  $res_0 := res_1 := \bot$

7:  procedure $entry(i, s)$        {Entry section for process $p_i$ and resource $r_s$}
8:    $entryRound := r$ ; $acquiring := s$        {$r$ is the round number at which $entry(i, s)$ is invoked}
9:    if $init$ then
10:       $pendingRequests.init := pendingRequests.init \cup \{(i, s, entryRound)\}$
11:    else
12:       $pendingRequests.transit := pendingRequests.transit \cup \{(i, s, entryRound)\}$
13:    wait until beginning of next round        {To ensure we send our init or transit message}
14:    wait until $acquiring = \bot$
15:    return

16: procedure $exit(i, s)$        {Exit section for process $p_i$ and resource $r_s$}
17:    if $res_0 = s$ then $res_0 := \bot$ else $res_1 := \bot$
18:    return

19: upon beginning of a round $r$ do
20:    $resources := \emptyset$
21:    for all $j \in \{0, 1\} : res_j \neq \bot$ do send $(\textsc{Resource}, i, res_j, r)$ to all participating processes
22:    for all $(j, s', r') \in pendingRequests.init$ do send $(\textsc{Init}, j, s', r')$ to all participating processes
23:    for all $(j, s', r') \in pendingRequests.transit$ do send $(\textsc{Transit}, j, s', r')$ to all participating processes

24: upon receiving $msg = (\textsc{Resource}, j, s', r')$ do
25:    $resources := resources \cup \{(j, s', r')\}$
26: upon receiving $msg = (\textsc{Init}, j, s', r')$ do
27:    $pendingRequests.init := pendingRequests.init \cup \{(j, s', r')\}$
28: upon receiving $msg = (\textsc{Transit}, j, s', r')$ do
29:    $pendingRequests.transit := pendingRequests.transit \cup \{(j, s', r')\}$

30: upon end of a round $r$ do
31:    for all $(j, s', r) \in resources$ do        {Old init and transit messages are removed}
32:       $pendingRequests.init := pendingRequests.init \setminus \{(j, s', *)\}$
33:       $pendingRequests.transit := pendingRequests.transit \setminus \{(j, s', *)\}$
34:       $starvers.init := starvers.init \setminus \{(j, s', *)\}$
35:       $starvers.transit := starvers.transit \setminus \{(j, s', *)\}$
36:    if $\exists (j, s', r') \in pendingRequests.init$ and $r' < Starving(entryRound)$ then
37:       $starvers.init := starvers.init \cup \{(j, s', r')\}$
38:    if $\exists (j, s', r') \in pendingRequests.transit$ and $r' < Starving(entryRound)$ then
39:       $starvers.transit := starvers.transit \cup \{(j, s', r')\}$
40:    if $acquiring \neq \bot$ then
41:       if $pred(A)$ then
42:          if $res_0 = \bot$ then $res_0 := acquiring$ else $res_1 := acquiring$
43:          $acquiring := \bot$; $init := false$
```

Figure 3. Canvas for our Driving Philosophers algorithms, instantiated with predicate *pred*(A)

respectively new or in transit.[6] The union of the two sets is denoted by *pendingRequests*. Both sets are updated at the end of each round, with the messages received during the round. Each message consists of a tuple, where the first field is the type of the message (i.e., RESOURCE, INIT, or TRANSIT), the second field is the identifier of the process sending the message, the third field is the identifier of the resource involved, and the fourth field is the round number in which the message is sent. We assume that the sets

[6]A process is *in transit* as soon as it owns a resource.

automatically eliminate duplicate entries. A process p_i that wishes to access a resource sends a message INIT or TRANSIT (depending on whether p_i is new or in transit) with its process identifier and its entry round. When a process p_i owns a resource r_s, p_i announces to the other participating processes that it holds r_s, by sending a message RESOURCE in every subsequent round in which p_i holds r_s. Processes record the set of busy resources in the set *resources*.

As far as starvation freedom is concerned, any process p_i maintains two local sets, *starvers.init* and *starvers.transit*, with the identity of "starving" processes, respectively new or in transit. *starvers* denotes the union of the two sets. To decide whether a process is starving, p_i uses a function *Starving* : $\mathbb{N} \to \mathbb{N}$, which p_i applies to its own entry round (i.e., the round at which p_i invokes the entry section), stored in variable *entryRound*, and then compares the result with the entry round of other processes. At the end of any round, p_i adds to *starvers* the processes which are waiting since earlier than *Starving*$(\mathit{entryRound})$. Process p_i removes a process from its set *starvers* as soon as p_i receives a message RESOURCE from this process. We let function *Starving* be *Starving*$(r) = r - \Delta$, where Δ is a constant, for instance $\Delta = 10, 15, 20, \ldots$. Different choices are possible for function *Starving*.

Before accessing any resource, predicate *pred*(A) must hold true for a process to enter. *pred*(A) is defined in a generic way as:

$$\begin{aligned} \mathit{pred}(A) \quad \triangleq \quad & \mathit{predMutex} \wedge \\ & [(\mathit{predInit}(A) \wedge \neg \mathit{predStarvers} \wedge \neg \mathit{predGridlock}_i) \vee \neg \mathit{init}\ ] \wedge \\ & [\ \mathit{predTransit}(A) \vee \mathit{init}\ ], \end{aligned}$$

where *predInit*(A) and *predTransit*(A) are defined separately for each instance of the canvas, and are respectively evaluated by new processes and processes in transit, as part of *pred*(A). *predMutex* and *predStarvers* are defined as:

$$\begin{aligned} \mathit{predMutex} \quad & \triangleq \quad (*, s, r) \notin \mathit{resources} \\ \mathit{predStarvers} \quad & \triangleq \quad \mathit{starvers} \neq \emptyset \wedge (i, *) \neq \mathit{min}_{r',j}\{(j, r') | (j, s', r') \in \mathit{starvers}\}, \end{aligned}$$

where function *min* (resp. *max*) takes as subscript the variable for which the minimum (resp. the maximum) is considered (in the order of appearance of the variables if more than one). *predMutex* ensures mutual exclusion at the resource and is generic to both new processes and processes in transit. *predStarvers* ensures starvation freedom, by preventing new processes to access a free resource, when there is a starving process (unless the starving process is the process evaluating *predStarvers* itself). *predGridlock*$_i$ avoids a gridlock, by preventing a new process to access a free resource, when this process could create a cyclic chain of waiting processes. The index i in *predGridlock*$_i$ allows a process p_j distinct from p_i to evaluate *predGridlock* with p_i's identity. In contrast, the predicates *predMutex* and *predStarvers* are always evaluated by and concerning a single process p_j.

Roughly speaking, *predGridlock*$_i$ is described as "there may remain no free resource in next round **and** i is the highest process id in INIT messages for free resources with the shortest waiting time **and** p_i does not only receive its own INIT message."

More precisely, let s_{max} be the highest resource identifier process p_i is aware of, from the messages received in previous rounds.[7] $\mathit{predGridlock}_i$ is defined as:

$$\begin{aligned}\mathit{predGridlock}_i \quad \triangleq \quad & \forall s \in [0, s_{max}] : (*, s, *) \in \mathit{pendingRequests} \cup \mathit{resources} \wedge \\ & (i, *) = max_{r',j}\{(j, r')|(j, s', r') \in \mathit{pendingRequests.init} \wedge \\ & (*, s', *) \notin \mathit{resources}\} \wedge \mathit{pendingRequests} \cup \mathit{resources} \neq \{(i, *, *)\}.\end{aligned}$$

Predicting a gridlock is not easy, because the number of ressources. The idea used in the predicate is to make sure the new configuration always contains at least a free resource. We state a preliminary lemma, in sight of proving the mutual exclusion property of the Driving Philosophers problem, separately from any specific instance of the canvas.

LEMMA 1 *If process p_i owns resource r_s in round r, no process but p_i may own r_s in round $r + 1$.*

A Simple Sequential Algorithm. Clearly there are solutions to the Driving Philosophers problem in the synchronous model. A simple algorithm consists in allowing a single process at a time in the round-about. A process p_i, that wishes to access resource r_s, sends a request message to all other participating processes, as soon as p_i takes part in the problem. Process p_i enters the critical section if and only if (a) in the previous round, there was no message from any process in the critical section, and (b) p_i is the process in *pendingRequests.init* which has been waiting for the longest period of time. The algorithm, denoted Serial, is obtained by instantiating the canvas in Figure 3 with the following predicates:

$$\begin{aligned}\mathit{predInit}(\mathsf{Serial}) \quad \triangleq \quad & (i, *) = \mathit{min}_{r',j}\{(j, r')|(j, s', r') \in \mathit{pendingRequests.init}\} \wedge \\ & (*, *, r) \notin \mathit{resources} \\ \mathit{predTransit}(\mathsf{Serial}) \quad \triangleq \quad & \mathit{true}.\end{aligned}$$

In the next paragraph, we refine our problem. Indeed we forbid such solutions by requiring an additional property to the Driving Philosophers problem.

Concurrency. To avoid sequential solutions such as the one described above, we add a concurrency property to our Driving Philosophers problem. We reformulate the definition of concurrency from [5] in our settings:[8]

(P3) From any configuration C,[9] any invocation of an entry section *entry*(i, s) by p_i for r_s is granted within the minimum number of steps for any entry section

[7]In Figure 3, maintaining s_{max} up-to-date when new messages are received is not shown.

[8]Indeed the very same definition of concurrency ("The solution does not deny the possibility of simultaneous drinking from different bottles by different philosophers") does not apply in our case. In our case for instance, a process cannot enter the round-about if its presence might cause a gridlock, although it may not be in direct conflict with any other process.

[9]Note that in a given configuration C, no process may be starving. Starvation appears when we consider a sequence of configurations, i.e., a run.

invocation to return in any run, unless (1) there is a concurrent entry section invocation for the same resource (contention on r_s), or (2) the configuration resulting if all concurrent yet non-conflicting entry section invocations are granted (including p_i's one) may be gridlocked.

Looking ahead, we introduce a relation $>_c$ to compare different algorithms with respect to their degree of concurrency.

DEFINITION 2 *Let A_1 and A_2 be any two distinct Driving Philosophers algorithms. We say that A_1 is more concurrent than A_2, denoted $A_1 >_c A_2$, if (1) for any configuration C in which any new process p_i has invoked entry(i, s) for resource r_s and pred(A_2) is true at p_i (i.e., for any process p_i which is in its trying section and is going to enter its critical section), then pred(A_1) is true at p_i, and (2) there is a configuration C_0 such that pred(A_1) is true and pred(A_2) is false, at p_i.*

Algorithm Concur1. Roughly speaking, the idea of our first concurrent algorithm, is that processes initially compete to access their first resource; once a process owns a resource, it has priority on the next resource over a new process. The algorithm is defined with the following predicates:

$$\begin{aligned} \textit{predInit}(\mathsf{Concur1}) &\triangleq (*, s, *) \notin \textit{pendingRequests.transit} \\ \textit{predTransit}(\mathsf{Concur1}) &\triangleq \textit{true}. \end{aligned}$$

THEOREM 3 *Concur1 solves the Driving Philosophers problem, and is concurrent.*

Algorithm Concur2. Roughly speaking, in our second concurrent algorithm, a new process p_i has priority over a process that already owns a resource, unless p_i detects a potential gridlock or a starving process (distinct of p_i). The algorithm is defined by the following predicates:

$$\begin{aligned} \textit{predInit}(\mathsf{Concur2}) &\triangleq \textit{true} \\ \textit{predTransit}(\mathsf{Concur2}) &\triangleq (*, s \oplus 1, *) \notin \textit{pendingRequests.init} \lor \\ &\quad (\exists (j, s \oplus 1, *) \in \textit{pendingRequests.init} \land \textit{predGridlock}_j) \lor \\ &\quad (\exists (j, s', *) \in \textit{starvers.init} \land s' \neq s \oplus 1). \end{aligned}$$

THEOREM 4 *Concur2 solves the Driving Philosophers problem, and is concurrent.*

3 Local Fairness

It is appealing to define a notion of fairness that takes into account the *position* of a process with respect to the resource(s) it owns. In this section we introduce a new notion of fairness, denoted x-fairness, defined only within a proximity scope, and propose a locally fair algorithm. We relate concurrency with fairness, and prove that our locally fair algorithm cannot be concurrent for most locality values. We first introduce

Δ-starvation,[10] to crystallize the notion of starvation in local fairness. We also introduce, for any resource r_s and any $h \in \mathbb{N}$, a set of resources denoted by *cluster*(s, h), corresponding to the resources neighboring r_s within a radius of h resources. Finally, we define the resources neighboring the location of a process p_i as *neighborhood*(i). Formally, we have:

DEFINITION 5 *For any resource r_s, a process p_i is* Δ-starving *if it invokes entry(i, s) in round r, and does not return from the invocation before round $r + \Delta$.*

DEFINITION 6 *For any resource r_s and any $h \in \mathbb{N}$,*
cluster$(s, h) = \{r_{s \ominus \lfloor \frac{h}{2} \rfloor}, \dots, r_s, \dots, r_{s \oplus \lfloor \frac{h-1}{2} \rfloor}\}$.

DEFINITION 7 *For any process p_i, neighborhood$(i) \supseteq \{r_{s\ominus 1}, r_s, r_{s\oplus 1}\}$ if p_i owns r_s or invokes entry(i, s).*

DEFINITION 8 *A Driving Philosophers algorithm is* x-fair *if no new process p_i, before returning from entry(i, s), waits more than any other process that invokes any entry section after p_i to access its first resource within cluster(s, x), unless (1) the configuration resulting if all concurrent yet non-conflicting entry section invocations are granted (including p_i's one) may be gridlocked, or (2) there is (at least) a Δ-starving process.*

THEOREM 9 *There is no concurrent, x-fair algorithm to the Driving Philosophers problem, for any $2 \leq x \leq k$.*

Algorithm x-Fair. This algorithm is x-fair according to Definition 8. In case of possibility of a gridlock or Δ-starvation, processes in transit have a static priority over new processes. The algorithm is defined by the following predicates:

$$\begin{aligned}
\textit{predInit}(\text{x-Fair}) \quad \triangleq \quad & (\textit{entryRound}, i) \leq \textit{min}_{r',j}\{(r', j) | \\
& \quad \big[(j, s', r') \in \textit{pendingRequests.init} \wedge s' \in \textit{cluster}(s, x)\big] \vee \\
& \quad \Big[(j, s', r') \in \textit{pendingRequests.transit} \wedge s' = s \ominus \lfloor \frac{x}{2} \rfloor\Big]\}
\end{aligned}$$

$$\begin{aligned}
& \textit{predTransit}(\text{x-Fair}) \quad \triangleq \\
& \quad \Big[(\textit{entryRound}, i) < \textit{min}_{r',j}\{(r', j) | (j, s \oplus \lfloor \frac{x}{2} \rfloor, r') \in \textit{pendingRequests.init}\}\Big] \vee \\
& \quad \Big[\exists (j, s \oplus \lfloor \frac{x}{2} \rfloor, r') \in \textit{pendingRequests.init} \wedge (r', j) < (\textit{entryRound}, i) \wedge \textit{predGridlock}_j\Big] \vee \\
& \quad \Big[\exists (j, s', *) \in \textit{starvers.init} \wedge s' \neq s \oplus \lfloor \frac{x}{2} \rfloor\Big].
\end{aligned}$$

Roughly speaking, a new process may access its first resource only if it has been waiting for a longer time than a process in transit, trying to access the same resource. This general rule cannot be satisfied in all cases. More precisely, when there is a risk

[10] In Figure 3, Δ-starvation is hidden behind function *Starving*.

of gridlock or when a new process is starving, any other new process must refrain from accessing the resource, and must give way to processes in transit.

THEOREM 10 *x-Fair solves the Driving Philosophers problem, and is x-fair for any $1 \leq x \leq k$.*

COROLLARY 11 *x-Fair is not concurrent, for any $2 \leq x \leq k$.*

THEOREM 12 *1-Fair is concurrent.*

THEOREM 13 *Concur2 $>_c$ 1-Fair $>_c$ Concur1 $>_c$ Serial.*

4 Impossibility Results

Asynchrony

We consider here an asynchronous model, where the time taken by any process p_i to execute a step is finite but unknown, the time taken by p_i to use any resource r_s is finite but unknown, and processes do not fail. Communication is reliable, in the sense that any message sent is eventually delivered, no spurious messages are created, and no messages are duplicated. Communication is asynchronous in the sense that the message propagation time is finite but unknown, and may be arbitrarily large. Intuitively, mutex is not solvable in this model because we do not know from which processes we may receive messages, and how long we may wait before considering that there is no process to communicate with. The mutex impossibility automatically implies the impossibility of the Driving Philosophers problem in such a model, as the (non-concurrent variant of the) Driving Philosophers reduces to mutex.

THEOREM 14 *There is no solution to the mutex problem in an asynchronous model amongst an arbitrarily large set of processes.*

COROLLARY 15 *There is no solution to the Driving Philosophers problem in an asynchronous model amongst an arbitrarily large set of processes.*

Locality

In this section, we investigate the solvability of the Driving Philosophers problem with *local* communication, revisiting the assumption that all participating processes may directly communicate with each other, but considering that processes may communicate only with *nearby* processes. This local communication assumption is motivated by the limited communication range of typical ad-hoc mobile devices. We conjecture the impossibility of a solution to the Driving Philosophers problem with local communication, and prove it for a restricted case. Informally, we say that communication is h-local for any process p_i, or that p_i h-communicates, if p_i may communicate only with processes whose neighborhood are in the cluster of any resource within p_i's neighborhood. More precisely, let $Scope_i$ be the set of processes to which p_i may send a message, or from which p_i may receive a message. For any process p_i, resource r_s and $h \in \mathbb{N}$, we say that communication is h-local, if $\forall p_j \in Scope_i$, $\exists r_s \in neighborhood(i)$, such that $neighborhood(j) \cap cluster(s, h) \neq \emptyset$.

CONJECTURE 16 *In any Driving Philosophers algorithm, there exists a run of A, for which there exist a process p_i and a resource r_s such that, for any $h \in \mathbb{N}$, between the invocation entry(i, s) and its return, there exists $H > h$ such that p_i H-communicates.*

We prove a weaker proposition, Proposition 18, which corresponds to Conjecture 16 restricted to algorithms belonging to a class we introduce and we denote by $\mathsf{ConservativeAlgorithms}$.

DEFINITION 17 *An algorithm solving the Driving Philosophers belongs to* $\mathsf{ConservativeAlgorithms}$ *if any new process p_i may return from its first invocation to entry(i, s) only if no process owns any resource in cluster(s, h).*

PROPOSITION 18 *In any Driving Philosophers algorithm* $A \in \mathsf{ConservativeAlgorithms}$*, there exists a run of A, for which there exist a process p_i and a resource r_s such that, for any $h \in \mathbb{N}$, between the invocation entry(i, s) and its return, there exists $H > h$ such that p_i H-communicates.*

5 Concluding Remarks

Since Dijkstra's seminal paper [6] which first stated the mutual exclusion (mutex) problem and solved it in a system where processes communicate using shared memory, many mutex solutions have been given. In the message passing model, mutex was first solved by Lamport [12]. Other papers have refined his result, improving the performance of mutex algorithms (e.g. [13]). Several variants of mutex have later appeared in the literature, for instance group mutual exclusion [11], and l-exclusion [8]. In the Dining Philosophers, a fixed set of processes is organized as a ring. The Drinking Philosophers generalizes the ring of the Dining Philosophers to an arbitrary graph of processes, whereas [3] generalizes all philosophers problem as *neighborhood-constrained* problems. [3] however assumes a static configuration of processes and resources. Interestingly, the same generalizations can be made to the Driving Philosophers. This generalization is however orthogonal to the issues raised in this paper, and is subject to future work.

To our knowledge, all attempts to address mutex kind of problems in mobile adhoc networks [4, 16] consider weak variants of the problem where mutual exclusion is ensured only when the network is "stable" for a certain period of time.

In fact, another seminal problem in distributed computing, namely consensus, has recently been considered in a model with an unbounded number of processes [15], more precisely, where the participation of any process to the algorithm is not required. The underlying model however assumes a shared memory. Interestingly, consensus is in fact not solvable in our system model (no shared memory), even if we consider strong synchrony assumptions. This conveys an interesting difference between the consensus and mutual exclusion problems, in the kinds of models we consider.

In the channel allocation problem [9], a known set of fixed processes (nodes) communicate through point-to-point asynchronous message passing. Each node knows the list of free resources (frequency bands) in its area and the list of processes' requests for these frequencies. Any node has to grant requests of any process, but not simultaneously with an adjacent node, and for the same frequency. The problem does however not consider starvation issues, as these frequency allocations occur for calls that can be dropped. In the multi-robot grid (MRG) problem [10], a fixed set of robots has to move on a grid to reach specific targets. The number of robots is known and no new robot may enter the grid. Furthermore to reach its target, a robot does not need to follow a specific path.

Acknowledgments

We would like to thank Hagit Attiya for pointing out to us the terminology of gridlock freedom.

References

[1] H. Attiya and J. Welch. *Distributed Computing*. McGraw-Hill, 1998.

[2] S. Baehni, R. Baldoni, R. Guerraoui, and B. Pochon. The Driving Philosophers. Technical Report IC/2004/15, EPFL, Lausanne, 2004.

[3] V. Barbosa and E. Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *ACM Transactions on Programming Languages and Systems*, 11(4):562–584, 1989.

[4] M. Benchaïba, A. Bouabdallah, N. Badache, and M. Ahmed-Nacer. Distributed mutual exclusion algorithms in mobile ad-hoc networks. *ACM Operating Systems Review*, 38(1):74–89, January 2004.

[5] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[6] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[7] D. Estrin, R. Govindan, and J. Heidemann. Embedding the internet. *Communication of the ACM*, 43(5):39–41, 2000.

[8] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *IEEE Symposium on Foundations of Computer Science*, pages 234–254, 1979.

[9] N. Garg, M. Papatriantafi lou, and P. Tsigas. Distributed long-lived list colouring: How to dynamically allocate frequencies in cellular networks. *ACM Wireless Network*, 8(1):49–60, 2002.

[10] R. Grossi, A. Pietracaprina, and G. Pucci. Optimal deterministic protocols for mobile robots on a grid. *Information and Computation*, 173:132–142, 2002.

[11] Y. Joung. Asynchronous group mutual exclusion. In *Proceedings of the* 17^{th} *ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 51–60, 1998.

[12] L. Lamport. The mutual exclusion problem. *Journal of the ACM*, 33(2):313–348, 1985.

[13] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[14] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[15] M. Merritt and G. Taubenfeld. Resilient consensus for infi nitely many processes. In *Proceedings of the* 17^{th} *International Symposium on Distributed Computing (DISC'03)*, pages 1–15, October 2003.

[16] J. Walter, J. Welch, and N. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 9(6):585–600, 2001.