The Driving Philosophers

S. Baehni[1] R. Baldoni[2] R. Guerraoui[1] B. Pochon[1]

[1] Distributed Programming Laboratory, EPFL, Switzerland
[2] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Italy

March 31, 2004

Contact author: Bastian Pochon
Email address: Bastian.Pochon@EPFL.ch
Postal address: Laboratoire de Programmation Distribuée
Faculté d'Informatique et de Communication
EPFL
CH-1015 Lausanne
Switzerland

Category: **Regular presentation**

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 amongst 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 Driving Philosophers

S. Baehni[1] R. Baldoni[2] R. Guerraoui[1] B. Pochon[1]
[1] Distributed Programming Laboratory, EPFL, Switzerland
[2] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Italy

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 amongst 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.

1 Introduction

Whilst 98% of the computers in the world are embedded devices, most research on synchronization is done with the 2% left in mind [5]. 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 [4]. 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 [3], 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, they do so (a) without a prior: 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 access 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 2, we first introduce some basic terminology, then the Driving Philosophers specification. In Section 3, 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 the notion of concurrency. In Section 4, we introduce the notion of fairness, and give a new algorithm that complies with this notion. Thus we prove that concurrency and fairness are antagonistic. In Section 5, 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 6, we discuss possible variants of our problem, and present some related works. Because of space limitation, we postpone any proof to the optional Appendix.

2 The Driving Philosophers

2.1 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 to 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 [4]. 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 invocation of *entry*(i, s), *granted* to p_i upon return from *entry*(i, s), and *released* by p_i upon 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) returns 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 might "think" for arbitrarily long).[1] We say that a process p_i is *new*, with respect to a resource r_s, if p_i does not own any resource prior to invoking *entry*(i, s). The interaction between a process and its *entry* and *exit* sections are illustrated in Fig. 1.

2.2 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.

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.

2.3 Differences with the 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 unique critical section. More specifically, 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, 12], our problem specification is broken into safety and liveness properties, as well as well-behaviorness of processes.

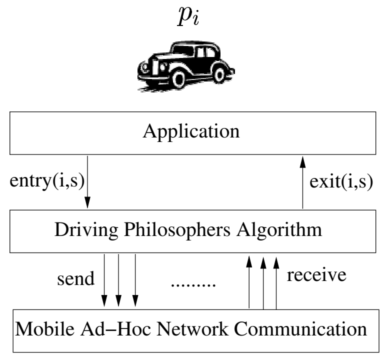

Figure 1: Component layout

The major impact of considering mobile processes is to introduce 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.

To sum up, 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 (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).

3 A Generic Algorithm

A generic algorithm solving the Driving Philosophers problem is presented in this section. To design this algorithm, we use the analogy of the Driving Philosophers with a round-about, as shown in Fig. 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 any resource r_s, i.e., invokes $exit(i, s)$ (1) as soon as p_i acquires $r_{s\oplus 1}$, and (2) before requesting $r_{s\oplus 2}$. In this way, any process holds at most two resources at a time. This is an assumption on processes well behavior, which could be described together with properties B1, B2 and B3. As such, the algorithm presented in Fig. 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 [12].[4] Roughly speaking, in each synchronous round, every process goes through three phases: in the first (send) phase, the process sends a message to 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 [12], our model differs in the sense that the set of participating processes (in a given round) is

[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."

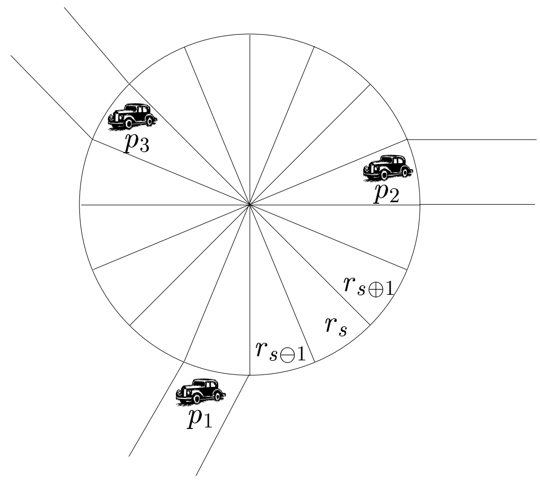

Figure 2: Analogy of the Driving Philosophers problem with a round-about

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 $C = \langle \mathit{Waiting}, \mathit{Driving} \rangle$. $\mathit{Waiting} : \Theta \rightarrow 2^{\Omega}$ is a function which gives information about processes in their trying state at the end of round r: for any resource $r_s \in \Theta$, $\mathit{Waiting}(s)$ is the set of processes in the entry section for r_s, $\emptyset$ if no process has requested for this resource. $\mathit{Driving} : \Theta \rightarrow \Omega \cup \{\perp\}$ is a function which gives information about resources that are occupied at the end of round r: for any resource $r_s \in \Theta$, $\mathit{Driving}(s)$ is the process that owns r_s in C, or $\perp$ 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 $C = \langle \mathit{Waiting}, \mathit{Driving} \rangle$ is *gridlocked* if $\forall r_s \in \Theta : \mathit{Driving}(s) \neq \perp$.

The Canvas. We first give a generic canvas for the Driving Philosophers problem in Fig. 3. Key to this canvas is a predicate that defines when processes are allowed to effectively access the resource. We instantiate this canvas to various algorithms: each algorithm A corresponds to a predicate *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, respectively new or which already own a resource. *pendingRequests* denotes the union of the two sets. 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 (e.g., RESOURCE, INIT, 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 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 already owns a resource respectively) 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

Critical section procedures for p_i:

```
init := true
pendingRequests := (∅, ∅)                {(pendingRequests.init, pendingRequests.transit)}
starvers := (∅, ∅)                       {(starvers.init, starvers.transit)}
resources := ∅
acquiring := entryRound := ⊥
res_0 := res_1 := ⊥

procedure entry(i, s)                    {Entry section for process p_i and resource r_s}
  entryRound := r ; acquiring := s       {r is the round number at which entry(i, s) is invoked}
  if init then
    pendingRequests.init := pendingRequests.init ∪ {(i, s, entryRound)}
  else
    pendingRequests.transit := pendingRequests.transit ∪ {(i, s, entryRound)}
  wait until beginning of next round     {To ensure we send the init or the transit message}
  wait until acquiring = ⊥
  return

procedure exit(i, s)                     {Exit section for process p_i and resource r_s}
  if res_0 = s then res_0 := ⊥ else res_1 := ⊥
  return

upon beginning of a round r do
  resources := ∅
  for all j ∈ {0, 1} : res_j ≠ ⊥ do send (RESOURCE, i, res_j, r) to all participating processes
  for all (j, s′, r′) ∈ pendingRequests.init do send (INIT, j, s′, r′) to all participating processes
  for all (j, s′, r′) ∈ pendingRequests.transit do send (TRANSIT, j, s′, r′) to all participating processes

upon receiving msg = (RESOURCE, j, s′, r′) do
  resources := resources ∪ {(j, s′, r′)}
upon receiving msg = (INIT, j, s′, r′) do
  pendingRequests.init := pendingRequests.init ∪ {(j, s′, r′)}
upon receiving msg = (TRANSIT, j, s′, r′) do
  pendingRequests.transit := pendingRequests.transit ∪ {(j, s′, r′)}

upon end of a round r do
  for all (j, s′, r) ∈ resources do      {Old init and transit messages are removed}
    pendingRequests.init := pendingRequests.init \ {(j, s′, ∗)}
    pendingRequests.transit := pendingRequests.transit \ {(j, s′, ∗)}
    starvers.init := starvers.init \ {(j, s′, ∗)}
    starvers.transit := starvers.transit \ {(j, s′, ∗)}
  if ∃(j, s′, r′) ∈ pendingRequests.init and r′ < Starving(entryRound) then
    starvers.init := starvers.init ∪ {(j, s′, r′)}
  if ∃(j, s′, r′) ∈ pendingRequests.transit and r′ < Starving(entryRound) then
    starvers.transit := starvers.transit ∪ {(j, s′, r′)}
  if acquiring ≠ ⊥ then
    if pred(A) then
      if res_0 = ⊥ then res_0 := acquiring else res_1 := acquiring
      acquiring := ⊥; init := false
```

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

in every subsequent round in which p_i holds r_s. Processes remember the set of busy resources using 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 which already own a resource. *starvers* denotes the union of the two sets. To decide whether a process is starving, p_i uses a function $Starving : \mathbb{N} \rightarrow \mathbb{N}$, 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 to 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 $Starving(entryRound)$. Process p_i removes a process from its set *starvers* as soon as p_i receives a message RESOURCE from this process. We choose function *Starving* as $Starving(r) = r - \Delta$, where Δ is a constant, for instance $\Delta = 10, 15, 20, \ldots$. Note that 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:

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

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

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

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. $predGridlock_i$ avoids a gridlock, by preventing a new process to access a free resource, when this process could create a cyclic chain of waiting processes. Roughly speaking, $predGridlock_i$ is described as "there may remain no free resource in next round **and** i is the highest pid 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. $predGridlock_i$ is defined as:

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

Predicting a gridlock is not easy, because the number of processes and of resources is not known. The idea used in the predicate is to make sure the new configuration still 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 decides to take part to the problem. Any 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 Fig. 3 with the following predicates:

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

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 [3] in our settings:[5]

(P3) From any configuration C,[6] 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 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:

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

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

[5] 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.

[6] In a given configuration C, a process may not be starving, nor Δ-starving.

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:

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

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

4 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 notion of 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 the notion of Δ-starvation, which crystallizes the notion of starvation used in local fairness. We also introduce, for any resource r_s and any $h \in \mathbb{N}$, a set of resources denoted by $\mathit{cluster}(s, h)$, 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 $\mathit{neighborhood}(i)$. Formally, we have:

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

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

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

Definition 8. *A Driving Philosophers algorithm is x-*fair *if no new process p_i, before returning from $\mathit{entry}(i, s)$, waits more than any other process that invokes any entry section after p_i to access its first resource within $\mathit{cluster}(s, x)$, 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:

$$predInit(\textsf{x-Fair}) \triangleq (entryRound, i) \leq min_{r',j}\{(r',j)| \\ (j,s',r') \in pendingRequests.init \wedge s' \in cluster(s,x) \wedge \\ (j,s',r') \in pendingRequests.transit \wedge s' = s \ominus \lfloor\frac{x}{2}\rfloor\}$$

$$predTransit(\textsf{x-Fair}) \triangleq \Big[(entryRound, i) < min_{r',j}\{(r',j)|(j, s \oplus \lfloor\frac{x}{2}\rfloor, r') \in pendingRequests.init\}\Big] \vee \\ \Big[\exists (j, s \oplus \lfloor\frac{x}{2}\rfloor, r') \in pendingRequests.init \wedge (r',j) < (entryRound, i) \wedge \\ predGridlock_j\Big] \vee \Big[\exists (j, s', *) \in starvers.init \wedge s' \neq s \oplus \lfloor\frac{x}{2}\rfloor\Big].$$

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 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.*

5 Impossibility Results

5.1 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.*

5.2 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, 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 ConservativeAlgorithms.

Definition 17. *An algorithm solving the Driving Philosophers belongs to* ConservativeAlgorithms *if any process p_i which does not own any resource, 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 $A \in$* 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.*

6 Concluding Remarks

Since Dijkstra's seminal paper [4] 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 [10]. Other papers have refined his result, improving the performance of mutex algorithms (e.g. [11]). Several variants of mutex have later appeared in the literature, for instance group mutual exclusion [9], and l-exclusion [6]. 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. Interestingly, the same generalization 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 ad-hoc networks [2, 14] 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 [13]. 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 [7], 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 [8], 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.

7 Acknowledgments

Thanks to Hagit Attiya for pointing out the notion of gridlock freedom.

References

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

[2] 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.

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

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

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

[6] 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.

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

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

[9] 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.

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

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

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

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

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

A Optional Appendix

This section contains the proofs of the lemmas, propositions and theorems stated in the main part of the paper.

Proof of Lemma 1. (Sketch) According to the algorithm, p_i sends a message (RESOURCE, i, s, r) in round r to announce that it holds r_s. At the end of round r, any participating process receives p_i's message. As a consequence, *predMutex* evaluates to false at any process participating in round $r+1$. Thus no other process but p_i may access r_s in round $r+1$. □

Proof of Theorem 3. (Sketch) Mutual exclusion follows from Lemma 1, and the observation that, when any resource r_s is empty, and two processes simultaneously request r_s, the priority is given to the process in transit, as specified in *predInit*(Concur1) and *predTransit*(Concur1). Starvation-freedom is ensured by preventing any new process to access a resource, as long a process is detected as starving by function *Starving*.

To prove that Concur1 is concurrent, first consider the minimal number of instruction *mininst* for a process to access a resource. This obviously corresponds to executing the entry section along with receiving messages once. Now consider any configuration run R. No process is starving in C (independently of the output of function *Starving* and of risk of gridlock). Consider that a new process requests resource r_s in C, by invoking *entry*(i, s). If (1) the resource r_s is empty, and (2) no process in transit also requests r_s, then according to *predInit*(Concur1) the entry section at p_i returns within *mininst* instructions. □

Proof of Theorem 4. (Sketch) The proof is very much similar to that one for Theorem 3. □

Proof of Theorem 9. (Sketch) Consider a configuration where only a single resource r_s is owned by process p_i. At time t_1, a second process p_j desires to acquire r_s, and thus has to wait on p_i (mutual exclusion). At time $t_2 > t_1$, a third process p_l arrives within *cluster*(s, x), and desires to acquire any resource r_t distinct from r_s. If p_l enters before p_j, it violates x-fairness, and if p_l does not enter, it violates concurrency. Thus in both cases, we obtain a contradiction. □

Proof of Theorem 10. (Sketch) Mutual exclusion follows from Lemma 1, and the observation that, when any resource r_s is empty, and two processes simultaneously request r_s, the priority is given to the process that has been waiting for the longest time, or, if both processes have been waiting for the same time, to the process with the least identifier. Starvation-freedom is ensured by preventing any new process to access a resource, as long a process is detected as starving by function *Starving*. As far as x-fairness is concerned, note that the predicates *predInit*(x-Fair) and *predTransit*(x-Fair) ensure priority within *cluster*(s, x), for any resource r_s. □

Proof of Corollary 11. (Sketch) By Theorem 10, x-Fair solves the Driving Philosophers problem and is x-fair. Hence by Theorem 9, x-Fair cannot be concurrent for $2 \leq x \leq k$. □

Proof of Theorem 12. (Sketch) Observe that 1-fairness corresponds to being fair within a cluster composed of a single resource. In this case, giving the way to the other process for achieving fairness, does not contradict Definition 2 for concurrency, since there exists a direct conflict amongst the processes concerned. □

Proof of Theorem 13. (Sketch) We prove the theorem by showing the three inequalities separately:

$\mathsf{Concur2} >_c \mathsf{1\text{-}Fair}$: Consider any configuration C in which $pred(\mathsf{1\text{-}Fair})$ holds true at a new process p_i that has invoked $entry(i,s)$ for some resource r_s. There are two cases to consider in which $pred(\mathsf{1\text{-}Fair})$ holds true at p_i: (i) $r_{s\ominus 1}$ is empty, or process p_j that owns $r_{s\ominus 1}$ has not invoked $entry(j,s)$, or (ii) process p_j that owns $r_{s\ominus 1}$ has invoked $entry(j,s)$ but $entryRound_j > entryRound_i$. In both cases (i) and (ii), $pred(\mathsf{Concur2})$ holds true at p_i as well: obviously in case (i), and because p_i has (static) priority over p_j in case (ii).

Consider a configuration C_0 such that $Waiting_{C_0}(s) = \{p_i, p_j\}$, $Driving_{C_0}(s \ominus 1) = \{p_i\}$, $i < j$, $\forall s' \neq s : Waiting_{C_0}(s') = \emptyset$, and $\forall s' \neq s \ominus 1 : Driving_{C_0}(s') = \perp$. Process p_j is thus a new process, for which $pred(\mathsf{Concur2})$ holds true, while $pred(\mathsf{1\text{-}Fair})$ does not.

$\mathsf{1\text{-}Fair} >_c \mathsf{Concur1}$: Consider any configuration C in which $pred(\mathsf{Concur1})$ holds true at a new process p_i that has invoked $entry(i,s)$ for some resource r_s. The only case to consider in which $pred(\mathsf{Concur1})$ holds true at p_i is when $r_{s\ominus 1}$ is empty, or process p_j that owns $r_{s\ominus 1}$ has not invoked $entry(j,s)$. In this case, $pred(\mathsf{1\text{-}Fair})$ holds true at p_i as well: obviously if $r_{s\ominus 1}$ is empty, and if not, but p_j which owns $r_{s\ominus 1}$ has not invoked $entry(j,s)$, p_j is not taken into account in the evaluation of $predInit(\mathsf{1\text{-}Fair})$ at p_i in $entry(i,s)$, which hence returns.

Consider a configuration C_0 such that $Waiting_{C_0}(s) = \{p_i, p_j\}$, $Driving_{C_0}(s \ominus 1) = \{p_j\}$, $i < j$, $\forall s' \neq s : Waiting_{C_0}(s') = \emptyset$, and $\forall s' \neq s \ominus 1 : Driving_{C_0}(s') = \perp$. Process p_i is thus a new process, for which $pred(\mathsf{1\text{-}Fair})$ holds true, while $pred(\mathsf{Concur1})$ does not.

$\mathsf{Concur1} >_c \mathsf{Serial}$: In Serial, any process p_i may access a resource only if no other resource is currently taken. Obviously in such a configuration, p_i may acquire the resource with Concur1 as well.

Consider a configuration C_0 such that $Waiting_{C_0}(s) = \{p_i\}$, $Driving_{C_0}(s \oplus 1) = \{p_j\}$, $\forall s' \neq s :$ $Waiting_{C_0}(s') = \emptyset$, and $\forall s' \neq s \oplus 1 : Driving_{C_0}(s') = \perp$. Process p_i is thus a new process, for which $pred(\mathsf{Concur1})$ holds true, while $pred(\mathsf{Serial})$ does not. □

Proof of Theorem 14. (Sketch) For any resource r_s and process p_i, consider first a run R_1 where a single process p_i takes part to the problem, and p_i invokes $entry(i,s)$. Because of the no-starvation specification of the mutex problem, there is a time t at which p_i eventually returns from the invocation, and owns r_s. Now consider a run R_2 where a single process $p_j \neq p_i$ takes part and p_j invokes $entry(j,s)$. For the same reason, there is a time t' at which p_j owns r_s. Finally consider a run R_3 where processes p_i and p_j respectively invoke $entry(i,s)$ and $entry(j,s)$, and assume that all messages between p_i and p_j are delayed until $t_1 = max(t,t')$. Now we postpone the time at which p_i and p_j respectively invoke $exit(i,s)$ and $exit(j,s)$ to time $t_2 = max(t,t') + 1$. Clearly, p_i and p_j violate mutual exclusion on r_s between t_1 and t_2. □

Proof of Proposition 18. (Sketch) Assume there exists an algorithm $A \in \mathsf{ConservativeAlgorithms}$ which solves the Driving Philosophers, and which does not satisfy Proposition 18. We derive a contradiction by constructing a run R of A which violates the starvation freedom property of the Driving Philosophers problem. We consider a set of mh resources, for $m, h \geq 1$, where resources are labelled $\{r_0, \ldots, r_{mh-1}\}$. m and h are not known by processes, and are merely introduced here for deriving the contradiction. We assume that (1) m processes $p_0, \ldots, p_{m-1}$ request distinct resources (process p_j requests resource r_{jh}, $0 \leq j \leq m-1$), (2) any process stops requesting any resource upon reaching a commong resource, say resource r_s, (3) no process ever owns more than two resources simultaneously, and (4) a number of processes wants to enter at resource $r_{s\oplus 2}$. We build R such that all processes execute steps simultaneously (this is possible since we assume that invocations to the entry section occur asynchronously). We consider a process p_i that wishes to enter at resource $r_{s\ominus h}$. We derive a contradiction by extending run R, and making p_i starve:

1. First extend R such that processes $p_0, \ldots, p_{m-1}$ simultaneously acquire their respective next resource, before releasing the previous one.
2. Consider any round r in which some process p_l ($0 \leq l \leq m-1$) reaches r_s. Thus p_l stops requesting any new resource, releases r_s. In the same round, any other process proceeds to its next resource.
3. Consider the first process waiting at resource $r_{s\oplus 2}$, say process p_j. We extend R such that all processes owning a resource proceed to the next resource when p_j is granted $r_{s\oplus 2}$.
4. Two processes are separated again by at least $h-1$ empty resources, and no more that $h-1$ consecutive resources are empty.

In the resulting configuration p_i may not enter resource r_s. By repeating the construction above, we extend R such that p_i may never access resource r_s, although it invokes $entry(i, s)$. This contradicts the fact that A solves the Driving Philosophers problem, in particular it violates the starvation freedom property. □