The Drinking Philosophers Problem: Resource Allocation in Distributed Systems

Seminar paper on Distributed Programming

Alexander Steen,
a.steen@fu-berlin.de

Freie Universität Berlin
Department of Computer Science

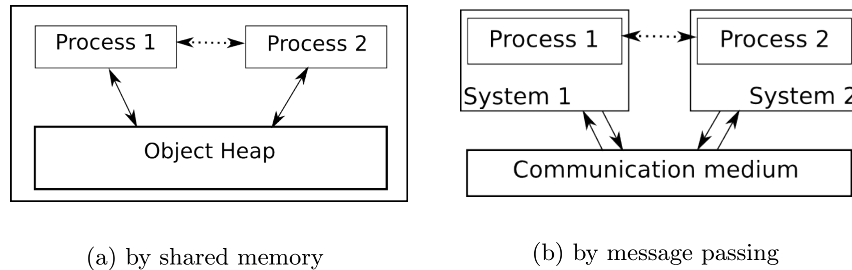
Abstract. In 1971 E. W. Dijkstra published the dining philosophers problem, which, since then, has become representative for resource allocation problems in concurrent and distributed programming. In this seminar paper, a generalization called the drinking philosophers problem by Chandy and Misra is surveyed. Since the problems has practical relevance, it has drawn a lot of attention and many different solutions have been proposed. A solution due to Ginat et al. is presented in detail. Additionally, an implementation in Go is given and discussed.

1 Introduction

One of the most fundamental and early challenges of concurrent programming are those which involve protecting resources from shared usage. These resources may for instance be files, which must not be written to in parallel. These problems emerged in 1962 when the first operating systems used multithreaded controllers for executing multiple programs while others wait for I/O actions to complete [1].

Since then, problems of concurrent programming have been deeply researched and many important formalizations have been developed. Some of the most popular and influential work is due to E.W. Dijkstra, who coined the term critical section and published the dining philosophers problem [2] [3]. The latter has become one of the most representative problems for conflict resolution between independent processes.

With the rise of computer networks in the 1980s and 1990s, the use of distributed systems and distributed algorithms have appeared. Unlike in classical concurrent systems, the communication between processes of different address spaces (possibly on different machines) is realized by passing around messages (see Fig. 1b) rather than by manipulating shared memory (Fig. 1a) [1].
The critical section problem of concurrent systems can also be applied to distributed systems. Here, critical sections may be the access to shared resources among independent systems, such as files in a distributed file system or some specialized hardware in the network. One of the main difficulties for distributed algorithms is the restricted knowledge of an individual process about the global



(a) by shared memory

(b) by message passing

Fig. 1: Communication between independent processes

state of the system. This means that each process has to act based on its local state (and received messages, of course) in order to solve a problem.

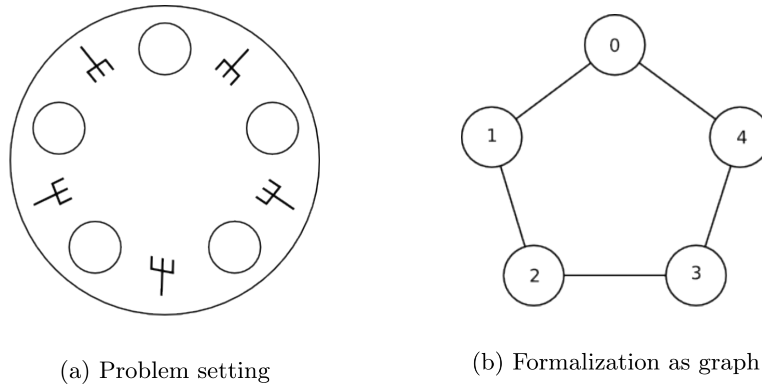
Section 2 gives a short summary about the classical dining philosophers problem as well as an introduction to the generalization, the drinking philosophers problem. The subsequent section surveys various solving strategies and discusses one of them in detail. In section 4, an implementation in Go is presented. Finally, the last section summarizes and gives credit to related problems.

2 The Problem Statement

This section gives a precise definition, what problems can be modeled in the drinking philosopher paradigm. Prior to that, the well known dining philosopher problem is recapped and described informally.

2.1 Dining Philosophers

The dining philosophers problem due to Edsger W. Dijkstra [4] is a resource allocation problem which can be described as follows: Five philosophers meet for diner and discussing at a round table with a fork placed on the table between each of them. Since discussing philosophy related problems is an exhausting thing to do, they need to eat once in a while. In order to do so, a philosopher requires the forks to his left and right, which are shared with his left and right colleague respectively. After a philosopher is done with eating, both forks will be put back on the table. It's important to note that a philosopher must hold both forks, otherwise he cannot eat at all. Furthermore, a philosopher does not speak about dining (i.e. the forks) with his colleagues and forks cannot be used by two philosophers at a time. Due to the fact that there are only five forks and each philosopher needs two of them for eating, it is obvious that at most two philosophers can eat in parallel. A solution to this problem has to ensure that multiple philosophers can eat at the same time. Additionally, deadly embraces (deadlocks) and starvation has to be avoided.



(a) Problem setting

(b) Formalization as graph

Fig. 2: Graphical representation of the dining philosophers problem

The setting of the diners problem given above can be formalized: Given an undirected graph $G = (V, E)$ with $V = \{0, 1, 2, 3, 4\}$, where each of the vertices describes a philosopher. The edges of this graph can be interpreted as resources shared between its two endpoints. Hence, $uv \in E \Leftrightarrow v = (u + 1) \mod 5$. Then, the condition to eat for a philosopher associated to node $v \in V$ is: v must acquire all the adjacent resources.
The behavior of a philosopher process is given by the following repeating cycle:

```
forever do
  think
  become hungry
  eat
```

Since this seminar paper treats distributed algorithms, a solution to this problem uses message passing for communication between the philosophers rather than shared synchronization objects, such as monitors or semaphores. While this will of course violate the constraint that philosophers are not allowed to talk to each other, it captures one important difficulties of distributed programming with shared resources, that is, the resolution of so-called conflicts. A conflict arises when multiple processes try to access a resource that must not be used by more than one process at a time.

This problem has been well researched and there exist several solutions that use either shared synchronization objects or, in the distributed case, message passing ([1],[4],[5], [6]). It is important to note that the most difficult accomplishment of giving a solution to this problem is to guarantee fairness.

The following *fair* solution is due to Chandy and Misra [7]. In order to assert fairness, the solution implements a *precedence graph* which assigns a priority to each process based on its depth in this (acyclic) graph. This is a interesting approach since it uses a fully deterministic way of giving precedence in cases of conflicts without statically pre-assigned priorities (i.e. a static hierarchy).

A precedence graph P is a directed acyclic graph where each node represents a process and each directed edge from p to q means that p has precedence over q. Then, the *depth* of a process p is defined as the length of the longest path from a process without any predecessors to p. It is easy to see that neighboring processes cannot have the same depth in P, which means that conflicts between those processes can always be resolved in favor of the process with the lowest depth.

In the solution to the diners problem, the precedence graph is implemented as follows: Each fork shared by two processes is either *clean* or *dirty*. Then, process p has precedence over q if (1) p holds the fork and the fork is clean, or (2) q holds the fork and the fork is dirty, or (3) the fork is currently being sent from q to p. Initially, all forks are dirty and must be placed (externally) such that the associated precedence graph is acyclic. The rules which each philosopher obey are:

- (i) A hungry philosopher requests all necessary but currently not owned forks from his neighbors
- (ii) (Despite hygienic conventions) Dirty forks can be used arbitrarily often to eat
- (iii) Dirty forks are being cleaned during transmission
- (iv) Clean forks remain clean until used for eating
- (v) A eating philosopher defers fork requests
- (vi) A non-eating philosopher satisfies requests for forks that are currently dirty and defers requests for forks that are clean

The transmission of forks can be simulated by sending *tokens* representing the fork.
This algorithm is fair since a philosopher will put precedence to his neighbors after eating (by making the forks dirty; (iv)), but will not give away forks freshly requested for eating (since those forks are clean; (iii), (vi)). An detailed proof of the correctness is given by Chandy and Misra [7].

2.2 Drinking Philosophers

The problem statement of section 2.1 is rather restricted: The process network describes a cycle graph, where each process always needs both of its adjacent resources to perform its critical section. In 1984, Chandy and Misra presented a generalization of this problem: The so called drinking philosophers problem [7].

In this paradigm of conflict resolution in distributed systems, the independent processes of the system correspond to nodes in a general graph. It is important to note that no restrictions on the graphs structure (e.g. connectivity, node degrees) are made. This means, that the underlying network of processes is completely abstracted from. The resources shared between the processes correspond to edges in this graph and are called *bottles*. The generalization introduced is twofold: Not only that the structure of the process graph is arbitrary, but also

the set of required bottles may vary. At each round each process chooses a subset of its adjacent bottles needed for this drinking session. This subset chosen does not necessarily have to be the same in each drinking session. In this sense, this paradigm is an abstraction of the dining problem: If we choose all of the adjacent bottles as required at each drinking session, we can simulate the dining philosophers problem. It's easy to see that, with the drinking philosopher paradigm, a lot of conflict resolution situations can be simulated.

In this paradigm, the processes again loop through states. These are *tranquil*, *thirsty* and *drinking*, representing the states *thinking*, *hungry* and *eating* (respectively) in the drinking philosophers problem, yielding

```
forever do
  be tranquil
  become thirsty
  drink
```

Just like in the diners problem, philosophers are allowed to be tranquil arbitrarily long. At any time, a process can decide to become thirsty. Then, a subset of the adjacent bottles is chosen as needed for drinking. After acquiring those bottles, a process only drinks finitely long after which it becomes tranquil and does not need the bottles anymore.

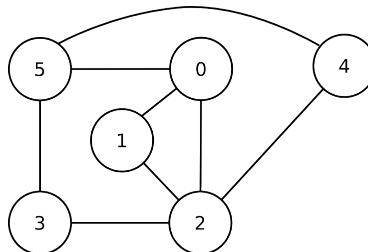


Fig. 3: Drinking philosopher setting example

Figure 3 shows an example for a drinking philosopher setting: In this example, any two processes connected by an undirected edge share a bottle (identified by this edge), e.g. process 5 shares a bottle with process 0, 3 and 4 and so on.

The solution criteria requested by Chandy and Misra [7] are:

Concurrency All simultaneous drinking sessions that are not forbidden by invariant (i.e. each fork is used at most once in parallel) must be possible.
Fairness No Philosopher is staying thirsty forever.
Symmetry Each philosopher process obeys the same set of rules.

Economy A philosopher sends and receives only a finite number of messages between state transitions.

Boundedness The number of messages in transit between any two philosophers is bounded as well as each message's size.

The fairness requirement is extremely difficult to fulfill here, since the decision whether a process has precedence over all its neighbors cannot be decided locally anymore (in contract to 2.1).

3 Solutions to the Drinking Philosophers Problem

The exist several solutions to the problem states in Section 2.2. The original solution to the problem stated by Chandy [7] uses the presented distributed solution to the diners problem as a layer to the actual solution. [yada bla]. One disadvantage of this solution is the relatively inefficient inaction between the actual solution and the diners solution layer. This is enhanced by Welch and Lynch [8] who present a more modular approach with exchangeable diners layer. Hence, the diners layer can be exchanged for a more efficient one if necessary. Besides the precedence graph approach, a non-deterministic solution might also be possible. Lehmann and Rabin proposed a non-deterministic solution for the diners problem [9].
However, the solution due to Ginat, Shankar and Agrawala which is discussed in this seminar paper uses session numbers to decide which process has precedence [10]. Before we take a closer look at the solution, *guarded commands* will be introduced. They are extremely handy for formulating conditional reactions in distributed systems and used in the algorithms description in Sect. 3.2.

3.1 Guarded Commands

In 1975 Edsger Dijkstra developed a programming language called *Guarded Command Language* (GCL) primarily used for the conceptual design of algorithms and their formal verification [11]. It allows expressing guarded (see below) actions in a straight-forward fashion which enables to specify the *behavior* of a system or algorithm without being forced to implement it with concrete control structures. In this sense, algorithm and system descriptions can be abstracted, granting a more focused and structured view on the underlying pre- and post conditions. This is why many algorithms, in particular in distributed programming, are often stated as guarded commands rather than in any specific imperative language: The behavior of the processes executing the distributed algorithm can be directly traced along the guarded commands and, in fact, more easily verified correct by proofs.
A *guarded command* is an expression of the form $\varphi \rightarrow stmt$, where φ is a Boolean expression and *stmt* is some program statement. φ is called the *guard* of the statement *stmt* and may be any Boolean expression consisting of the usual logical connectives and operations over program variables. In fact, in practical

applications, φ and *stmt* may also contain informal conditions or pseudo code (respectively). This was, however, not indented by the original description by Dijkstra, who gave a complete definition of syntax and semantics of statements and expressions in GCL.

The semantics of a guarded command $\varphi \rightarrow stmt$ is straight-forward: Whenever the system witnesses a state in which φ holds (e.g. the variables of the context make φ true), *stmt may* be executed. It is important to note that *stmt* does not necessarily have to be executed once φ holds; the system may, for example, grant the execution of another guarded command with valid guard, or do nothing at all. For practical uses it is convenient to assume that *some* guarded command with valid guard is executed (the surrounding system decides how to deal with valid guards explicitly). In the contrary case, when φ does not hold, the statement *stmt must not* be executed.
It is often handy to assume that *stmt* is executed in one atomic step.

3.2 The solution of Ginat, Shankar and Agrawala

The algorithm of of Ginat et al. presented in this section was published five years after the original problem statement by Chandy and Misra. The idea is to organize priorities by session numbers [10], which are non decreasing throughout the algorithms execution. To this end, each philosopher p has two integer variables max_rec_p and s_num_p: The first contains the highest session number encountered so far (received from neighbors), the latter the last, upcoming or current drinking session number (depending if p is tranquil, thirsty or drinking respectively). The solution makes use of the fact that adjacent philosophers will never have the same *extended session number* (s_num_p, p), which is the session number of the philosopher augmented by its identifier. In this sense, the augmentation can be regarded as *tie breaker*, since extended session numbers of adjacent philosophers cannot be equal and thus allow a conflict resolution in favor of one of the two adjacent processes. An ordering of extended session numbers is defined by the lexicographical ordering

$$\begin{aligned}(s_num_p, p) < (s_num_q, q) \;:\Longleftrightarrow \\ s_num_p < s_num_q \lor (s_num_p = s_num_q \land p < q)\end{aligned} \tag{1}$$

For each bottle b there exists an associated request token req_b which is sent by a philosopher p if b is needed for the next drinking session but not currently hold by p. This kind of request can only be sent if the process who is thirsty owns the request token. To ensure fairness, a session number greater than max_rec is picked after becoming thirsty and transmitted along with the request itself. A request message has the form (req_b, s, p) where req_b is the request token corresponding to the request, s the session number of the sender's upcoming drinking session and p the sender's identifier. A request message for bottle b is answered by the message (b) once the holder of b has lower precedence as the requesting philosopher. This is expressed by the following conflict resolution rule:

Upon receipt of (req_b, s, p) by q, q immediately sends b to p iff

$$\neg need_q(b) \vee (thirsty(q) \wedge (s, p) < (s_num_q, q))$$

Otherwise b will be released after q has finished drinking, i.e. is becoming tranquil. It is assumed that messages arrive in the order they are sent. Initially, for each process p the variables s_num_p and max_rec_p are zero and for each pair (p, q) of adjacent processes one is given b while the other one is given req_b.
For each process p, the following predicates are defined and used in their obvious meaning: $hold_p(b)$, $hold_p(req_b)$, $need_p(b)$ as well as $thirsty(p)$, $tranquil(p)$, $drinking(p)$, where $thirsty(p) \oplus tranquil(p) \oplus drinking(p)$ is invariant ($\oplus$ being exclusive or).
The following guarded command algorithm implements the above described solution [10]. Each guarded command execution is assumed to be atomic.

R1 *(becoming thirsty)* $tranquil(p)$ and p wanting to drink $\rightarrow$
  become thirsty
  for each desired bottle b do $need_p(b) \leftarrow true$
  $s_num_p \leftarrow max_rec_p + 1$

R2 *(start drinking)* $thirsty(p) \wedge \forall b : need_p(b) \Rightarrow hold_p(b) \rightarrow$
  become drinking

R3 *(stop drinking)* $drinking(p)$ and p wanting to stop drinking $\rightarrow$
  become tranquil
  for each consumed bottle b do
    $need_p(b) \leftarrow false$
    if $hold_p(req_b)$ then send(b); $hold_p(b) \leftarrow false$

R4 *(requesting a bottle)* $need_p(b) \wedge \neg hold_p(b) \wedge hold_p(req_b) \rightarrow$
  send(req_b, s_num_p, p); $hold_p(req_b) \leftarrow false$

R5 *(receiving a request)* $recv_p(req_b, s, q) \rightarrow$
  $hold_p(req_b) \leftarrow true$
  $max_rec_p \leftarrow \max(max_rec_p, s)$
  if $\neg need_p(b) \vee (thirsty(p) \wedge (s, q) < (s_num_p, p))$
    then send(b); $hold_p(b) \leftarrow false$

R6 *(receiving a bottle)* $recv_p(b) \rightarrow$
  $hold_p(b) \leftarrow true$

The abbreviation $recv_p(.)$ is used to express the event that p receives a message. $Send(b)$ and $Send(req_b, s_num_p, p)$ sends the respective message to the adjacent process with whom b is shared.
Rules R1, R2, R3 express the behavior described in Section 2.2: A process p deciding to be thirsty chooses a subset of all adjacent bottles to be needed for the upcoming drinking session. Additionally, a session number is picked (R1).

As soon as all bottles are acquired, the process is allowed to start drinking (R2). After that, the bottles aren't needed anymore and are possibly sent to neighboring processes in need (R3). Rules R4 and R6 treat requesting a bottle and the eventually receiving it (respectively). Rule R5 implements the above discussed conflict resolution rule: If the requesting process has precedence (i.e. the session number if smaller), the bottle is sent to it; otherwise the answer is postponed (see R3).

3.3 Analysis of the Algorithm

Restrictions The algorithm of Sect. 3.2 comes with some restrictions. It is required that the communication medium assures first-in-first-out message transport. This is necessary since the request token req_b (which is always sent *after* any sending of b) is required to arrive after a bottle b. Otherwise, rule execution R5 may corrupt the system's state. Furthermore, no message may be lost during transmission. A weak fair execution of the guarded commands is assumed.
The main restriction, which has to be discussed for actual practical applications, is the use of unbounded counters. The monotonicity of the session numbers is critical for the algorithm's fairness.

Correctness The proof of correctness is taken from Ginat et al.[10] with minor modifications (mostly for readability matters).
Lemma 2 and Lemma 3 are used later to prove the fairness of the algorithm. They are not proven in this paper but can be found in the original work. Additionally, the following properties of the algorithm (as stated by Ginat et al.) are used later on:

Lemma 1 (Properties). *The following properties hold for all adjacent philosophers p, q who share bottle b.*

B_1: *req_b is in transit to $q \Rightarrow b$ is held by q or in transit to q ahead of req_b*
B_2: *$hold_q(b) \Rightarrow b$ is not in transit to q*
$$C_1: hold_q(b) \wedge hold_q(req_b) \Rightarrow need_q(b) \wedge (max_rec_q \geq s_num_p) \wedge (drinking(p) \vee (s_num_q < s_num_p)) \quad \square$$

There are many more properties stated in the original work, primarily used for proving Lemma 2 and Lemma 3. The proof of Lemma 1 is omitted in favor of conciseness; a simple structural induction over the rules R1-R6 can be used for validating the properties.

The first important observation is asserted by

Lemma 2 (Extended session numbers).
(a) s_num_p and max_rec_p never decrease
(b) s_num_p does not change while p is thirsty □

We continue by defining the *dedication* of a bottle b, which will help us arguing that a philosopher will eventually gather all the required bottles. For

two thirsty adjacent philosophers p, q, let $p < q$ mean that p has precedence over q, i.e.

$$p < q :\Longleftrightarrow (s_num_p, p) < (s_num_q, q)$$

Definition 1 (Dedication). *Let p, q be two adjacent philosophers who share b. b is dedicated to p iff*

(1) $thirsty(p) \wedge need_p(b)$
(2) One of the following holds
- *(i) b is in transit from q to p*
- *(ii) $hold_p(b)$ and $p < q$*
- *(iii) $hold_p(b) \wedge \neg need_q(b) \wedge (s_num_p, p) < (max_rec_q + 1, q)$*

□

Dedication of b can only happen if and only if the philosopher really needs b (1) and has definitely precedence over the neighbor with whom he shares b (2). In cases (i) and (ii) precedence is trivial, in case (iii) the neighboring process will get a higher session number if it decided to be the thirsty, thus having a lower priority. Informally, dedication of a bottle b to p states that b is "temporally owned" by p, allowing us to state

Lemma 3 (Dedication). *If b is dedicated to p, then b will not be released by p before p is tranquil.* □

With assistance of Lemma 3, an ordering of processes is constructed, showing that eventually each process will be "first in line" and thus drink. Since this is a temporal property, the operator $\rightsquigarrow$ (read: leads-to) is introduced. The temporal character of properties of the form *whenever A holds, eventually b will hold* can be expressed with its help.

Definition 2 (Leads to). *Let A, B be two propositions. Then, $A \rightsquigarrow B$ holds iff whenever a state satisfying A is reached, somewhere in the future a state is eventually reached in which B holds.* □

Now we can easily formulate the fairness requirement which is, in fact, met as expressed by

Theorem 1 (Liveness). *For any philosopher p, $thirsty(p) \rightsquigarrow drinking(p)$.* □

Proof. Let all processes with same extended session number be grouped together and those groups be ordered by the ordering of the extended session numbers, and let further denote $pos(p)$ the position of the p's group in this ordering. Then, the theorem can be proved by proving the property

$$D(i) := (pos(p) = i \wedge thirsty(p)) \rightsquigarrow drinking(p)$$

by induction over i. Let p, q be two adjacent philosophers who share bottle b with $need_p(b)$.

1. **Basis** $i = 1$
   Let $pos(p) = 1$. Since we have $pos(p) < pos(q) \Leftrightarrow p < q$ and $pos(.) \geq 1$ it holds that $pos(q) > 1$. By Lemma 2 we know that $p < q$ as long as $thirsty(p)$ holds.
   Case 1: If b is dedicated to p, there is nothing to do, because b is (or will be) hold by b until p drunk from it.
   Case 2: b is not dedicated to p. But then b will eventually be dedicated to p, because
   Case (1): $hold_q(req_b)$:
   By B_2 it holds that $hold_q(b)$ and thus, by C_1 it follows that $drinking(q)$. Since drinking is finite, rule R3 will eventually be executed (i.e. b will be sent to p).
   Case (2): req_b is in transit to q:
   By B_1 it holds that $hold_q(b)$ upon reception of the request by p. Now, by rule R5, q either answers the request (and thus b is dedicated to p) or postpones it In the latter case, b will be dedicated to p due to Case (1).
   Case (3): $hold_p(req_b)$:
   By rule R4, req_b will be sent to q. Thus, by Case (2) b will be dedicated to p.
   By B_1 req_b cannot be in transit to p, so this case does not have to be considered. Hence, b will eventually be dedicated to p. Due to the fact that the number of bottles needed by p is finite, and, by Lemma 3, each dedicated bottle will be held by p until p is drinking, we can conclude that p will eventually start drinking (rule R2).
2. **Step** $D(1), \ldots, D(k) \rightarrow D(k+1)$
   Let $pos(p) = k + 1$. For p and q are neighbors it holds that $pos(p) \neq pos(q)$. If $pos(p) < pos(q)$, the same arguments of the induction base step can be applied. So, assume $pos(p) > pos(p)$.
   Case 1: If b is dedicated to p, there is nothing to do, because b is (or will be) hold by b until p drunk from it.
   Case 2: b is not dedicated to p. But then b will eventually be dedicated to p, because
   Case (1): $hold_q(req_b)$:
   By B_2 it holds that $hold_q(b)$ and thus, by C_1, it follows that $need_q(b)$. (A) If $drinking(q)$, rule R3 will eventually be executed (i.e. b will be sent to p), (B) if $thirsty(q)$, then (by induction hypothesis) q will be drinking in finite time and (A) can be applied.
   Case (2): $\neg hold_q(req_b)$:
   The arguments (2) and (3) of the induction basis can be applied to show that b will eventually be dedicated to p.
   By B_1 req_b cannot be in transit to p, so this case does not have to be considered. So, since p gathers all needed bottles in finite time, we can conclude by rule R2 that p will eventually start drinking.

□

Theorem 2 (Correctness). *The algorithm stated in Sect. 3.2 is correct. In particular, the algorithm provides* fairness, symmetry *and* concurrency*.*

Proof. Bottles cannot be used by multiple processes by construction. A process only starts drinking when all needed bottles are present (rule R2). Thus, the algorithm is safe (i.e. does not violate the invariant).
Since all processes use the same of rules, the solution is symmetric.
Fairness is ensured by Theorem 1.
Processes do not communicate with non-adjacent processes and only require neighboring resources. Thus, no legal execution is refused. □

Complexity. Let $G = (V, E)$ be the process graph under consideration (as described in Sect. 2.2) The message complexity of the presented algorithm can easily be determined: Let p, q be two different processes. For each bottle b shared between these processes p and q, p requires at most two messages two acquire the bottle (one message sending the request token req_b, one message sent from q containing the bottle b itself). In the best case, if p already holds b, no message has to be sent at all. Thus, for any drinking session requiring k bottles, at most $2k$ messages have to be exchanged. This means that the message complexity for each drinking session of any process p is bounded by $2\deg(p) = O(|V|)$, where $\deg(p)$ is the degree of p in G.

Since for each bottle b shared between processes p and q at most two messages can be in transit at the same time (q sending req_b to p; or p sending b to q; or p sending b to q followed by p sending req_b to q) it follows that the total amount of messages in transit is bounded by $2|E|$, yielding

Theorem 3 (Economy and Boundedness). *The number of messages in transit between any two processes is at most 2. For any process* p*, the number of messages sent and received for each drinking session of* p *is bounded by* $2\deg(p)$*. In particular, the algorithm is economical and bounded.* □

Further remarks The original solution of Chandy and Misra requires a strongly fair execution of the guarded commands. This restriction has been discussed and (partly) improved by Murphy and Shankar [12].

4 An Implementation in Google Go

The previous section presented the solution of Ginat et al. as set of guarded commands. In this section, a concrete implementation of their "theoretical" solution in Go is presented.
In Sect.4.1 a brief description of the goals is given. Then, Sect. 4.2 discusses the challenges of translating guarded commands to imperative code. Section 4.3 presents the implementation in Go; finally, the implementation is analyzed.

4.1 Goals of the Implementation

The implementation of the algorithm is not meant for providing a generic framework for distributed resource allocation. It has merely the purpose of a proof-of-concept. This is why the implementation is restricted to channel communication between local actors. Of course, the channels can be replaced by different communication objects to provide inter-network communication.
Another simplification is the use of a central coordination object, the `Allocation` object, to pose requests to. In a real distributed scenario, such a central coordination object would be replaced by an per-philosopher interface.

4.2 Translation from GCL

Since Go is an imperative language, the algorithm described in Sect. 3.2 cannot be implemented simply by adopting each rule in a sequential manner.
Let's recall the semantics of an GCL program: Any guarded command $\varphi \rightarrow \mathit{stmt}$ might be executed by the system environment as long as its guard, i.e. φ, holds. Note, that if φ holds permanently, the infinitely repeating execution of *stmt* is, of course, also a valid execution of the program[1] (even if there are further guarded commands in the program's description). This suggests that there is no fixed order in which the commands of a GCL program will be executed – revealing the difficulty of translating GCL programs to imperative ones: In imperative languages, the statements of a program describe its entire control flow, i.e. the possible sequences of execution. This means that, in the translation from GCL to Go, we have to explicitly encode *all* feasible executions of the GCL program using the imperative control structures.
If there are no blocking operations (such as $recv(.)$) in the guard of any command, a possible imperative translation is a *while*-loop containing a *if-else*-construct with one case per guarded command. Suppose a GCL program P is given by the set $\{\varphi_i \rightarrow \mathit{stmt}_i\}$ and each φ_i does not contain any blocking operation. Then, a Go translation of P is given by

```
for {
  if φ1 {
    stmt1
  } else if φ2 {
    stmt2
  } else if ...
    ...
  } else {
    return
  }
}
```

Listing 1.1: Imperative translation of P, Version 1

[1] *Fairness* assumptions of the system's scheduler might in fact restrict certain executions.

given that each $stmt_i$ has been correctly translated to Go.
In fact, the above translation is not completely correct: Although the code does not introduce any infeasible executions, it does not offer all possible ones. This is due to the *if* statement semantics of Go: The case predicates are checked for validity in a linear order $\varphi_1, \varphi_2, \ldots$, where the first statement with valid case predicate is executed. So, if φ_1 is always true, the statements $stmt_2, \ldots$ will never be executed.

A quick and easy fix for this problem is to introduce randomness to the translation, which relies on the confidence, that the implementation of Go's random generator guarantees nearly uniform distribution. With this tool, a random number i from $1, \ldots, n$ is chosen uniformly at each iteration of the *do*-loop, and the case condition of the i-th case is checked. This yields

```
for {
  i := Intn(n)
  if φ1 && i == 0 {
    stmt1
  } else if φ2 && i == 1 {
    stmt2
  } else if ...
    ...
  }
  if !φ1 && ... && !φn {
    return
  }
}
```

Listing 1.2: Imperative translation of P, Version 2

where `Intn(n)` is a function that chooses a random number from $0, \ldots, n-1$ (contained in the package `math/rand`).

To handle blocking operation in guards, each of the cases need to run in separate goroutines; this transformation is left to the reader. Note that in this translation scheme (in the presence of blocking operations in guards), the detection of termination is not trivial. Since in many cases the GCL programs describe reactive systems that are not supposed to terminate, termination detection can be ignored.

The above described transformation is naive in the sense that it creates one thread per guarded command. In many cases, this is an overkill and not necessary to express all executions of the GCL program: By reasoning about the guards and the effects of the commands, one can manually identify executions that cannot occur.
As an example, consider the following two guarded commands, where initially $i = 0$:

(i) $i = 0 \rightarrow$ `i = i + 1`
(ii) $i = 1 \rightarrow$ `i = i - 1`

Since after executing (i) its guard is false and its guard can only be made true again by (ii), an execution of the form (i), (i), ... is impossible. An analogous argument holds for (ii), and initially the guard of (ii) is false. Thus, a simple and correct imperative translation is given by

```
i := 0
for {
  i = i + 1
  i = i - 1
}
```

This kind of simplification is also used in the next section for the implementation of the Drinking Philosophers Problem. Subsequently, the correctness of the translation is proved.

4.3 Translation to Go

Adopting to a generic allocation interface. The Drinking Philosophers problem setting describes the problem (and solution) of distributed resource allocation in a fairly abstract way. It is, for example, not obvious how participants show that they "want to drink" (cf. rule R1) and consequently trigger certain guarded commands. To this end, resource allocation is modeled as interface – processes that need resources (i.e. "want to drink") solely need to invoke the appropriate function.

```
type Allocation interface {
  // Process p requests the resources
  // associated to the elements in res.
  // Blocks, until all resources has been
  // acquired.
  //
  // Pre:
  // - All elements in res are valid Process
  //   descriptors (or neighboring processes);
  // - Request(p,.) has not been called prior to
  //   this call without a subsequent call to
  //   Release(p);
  // - Start() has been executed prior to this call
  Request(p Process, res []Process)

  // Process p releases the resources
  // acquired due to the previous Request call
  //
  // Pre:
  // - Start() has been executed prior to this call
  Release(p Process)

  // Start needs to be called EXACTLY ONCE
  // before any invocation of Request and Release.
```

```
  //
  // Pre: Start has not been called in the past
  Start()
}
```

Listing 1.3: Definition of a resource allocation interface

The listing 1.3 displays this proposed interface which is derived from the usual Lock/Unlock-interfaces for mutual exclusion. The `Start` function is required to start the concurrent handling of message exchanges in the background.

Code excerpts. In the remainder of this section, only essential fragments of the Go implementation will be displayed.

Before the actual implementation of the guarded commands (rule R1 - R5) is given, the required state information of each philosopher needs to be modeled (see listing 1.4). Whether a philosopher is *tranquil*, *thirsty* or *drinking* is modeled as constant symbols of type Γ.

```
type Γ uint8
const (
  TRANQUIL Γ = iota
  THIRSTY
  DRINKING
)

type Ω struct {
  γ Γ  // the current state
  need map[Process]bool // needed bottles
  hold map[Process]bool // bottles held
  hold_req map[Process]bool // request tokens held
  max_rec Session
  s_num Session
  mutex *sync.Mutex // mutex and cond are used to
  cond *sync.Cond   // safely run the guarded command
                    // translations under mutual exclusion
}
```

Listing 1.4: Definition of philosopher state

The implementation of the predicates and extended session numbers defined in sect. 3.2 is straight-forward. Since, per philosopher, the guarded commands need to run under mutual exclusion (i.e. as observably atomic instruction), mutex-objects `mutex` are introduced. Additionally, a philosopher process may need to wait until all required bottles have been received. To implement this waiting efficiently, a monitor condition `cond` is used. All of this information in then integrated in a struct of type Ω.

The messages exchanged during the algorithm are modeled by the `Message` type displayed in listing 1.5.

```
type Message interface {
  IsReq() bool
  IsBottle() bool
  Session() Session
  From() Process
}
```

Listing 1.5: Definition of message type

The implementation of the message type is omitted here, as is the initialization of the `Allocation` implementation. The latter creates an Ω struct for each philosopher and randomly distributes the bottles between adjacent ones. The neighborhood is modeled by a simple `Network` type, which is, in fact, a graph interface. The representation of this graph is not further discussed in this paper.

Listing 1.6 below presents the implementation of the `Request` function that successively executes rules R1, R4 and R2. Each rule execution is protected by a mutex lock; the waiting is realized by a `Wait` call on the philosophers' condition variable.

```
func (a *ImpGinat) Request(proc Process, res []Process) {
...
  // Rule R1
  // becoming thirsty: choose subset of bottles needed
  // according to res.
  ω.mutex.Lock()
  ω.γ = THIRSTY
  for _,k := range res {
    ω.need[k] = true
  }
  ω.s_num = ω.max_rec + 1
  ω.mutex.Unlock()
  // end of rule R1
...
  // Rule R4
  // requesting bottles
  ω.mutex.Lock()
  for k,_ := range(ω.need) {
    if (!ω.hold[k] && ω.hold_req[k]) {
      a.request(proc,k,ω.s_num)
      delete(ω.hold_req,k)

    }
  }
  ω.mutex.Unlock()
  // end of rule R4
...
  // Rule R2
  // Checking if all desired bottles are there,
  // then drinking.
```

```
  ω.mutex.Lock()
  for ; !ω.holdsAllBottles(); {
    ...
    ω.cond.Wait()
  }
  ω.γ = DRINKING
  ω.mutex.Unlock()
  // end of rule R2
...
}
```

Listing 1.6: Implementation of the drinking philosopher algorithm, Part 1

If `Request` is called by a participating process, the invocation returns as soon as all requested resources have been acquired. In contrast to this, the `Release` function immediately returns to the caller (cf. listing 1.7).

```
func (a *ImpGinat) Release(proc Process) {
...
  // Rule R3
  // Finished drinking, possibly release send bottles
  ω.mutex.Lock()
  ω.γ = TRANQUIL
  for k,_ := range(ω.need) {
    if (ω.hold_req[k]) {
      a.sendBottle(proc,k)
      delete(ω.hold,k)
    }
    delete(ω.need,k)
  }
  ω.mutex.Unlock()
  // end of rule R3
...
}
```

Listing 1.7: Implementation of the drinking philosopher algorithm, Part 2

As discussed in the previous section, guarded commands that include blocking operations in its guards need to be executed in separate threads. The relevant guarded commands, where this condition applies, are rules R5 and R6 – both are guarded by a reception of messages. This is why the implementation of these rules are transferred into a separate function `handleMessages` and then later invoked as Goroutine via `go handleMessages(...)`. Observe that the rule R4 needs to be included as well, since rule R5 may make R4's guard valid again by sending away a bottle. The implementation of `handleMessages` is displayed in listing 1.8.

```
func (a *ImpGinat) handleMessages(proc Process) {
...
  for {
    msg := <-(a.inbox[proc])
    from := msg.From()
    if msg.IsReq() {
      // Rule R5
      // receiving a request and resolving a conflict
...
      ω.mutex.Lock()
      session := msg.Session()
      ω.hold_req[from] = true
      ω.max_rec = Max(ω.max_rec, session)
      if (!ω.need[from]) || (ω.γ == THIRSTY
            && lt(session,from,ω.s_num,proc)) {
        a.sendBottle(proc,from)
        delete(ω.hold,from)
        // hold(from) as become false
        // recheck Rule R4 for requesting if needed
        if ω.need[from] {
          a.request(proc,from,ω.s_num)
          delete(ω.hold_req,from)
...
        }
        // end of rule R4
      }
      ω.mutex.Unlock()
      // end of rule R5
    } else {
      // Rule R6
      // receiving a bottle
...
      ω.mutex.Lock()
      ω.hold[from] = true
      ω.cond.Signal()
      ω.mutex.Unlock()
      // end of rule R6
    }
...
  }
}
```

Listing 1.8: Implementation of the drinking philosopher algorithm, Part 3

The correctness of this translation is discussed in the next section.

Testing. A test case has been implemented in which a previously defined number of philosophers continuously request a randomly chosen subset of adjacent resources. The test case has been run for different numbers of philosophers several

times. It shows that each of the philosophers eventually possesses the requested resources.

4.4 Analysis

The implementation make use of Go's Goroutines, which allow to start function calls in independent threads. The guarded commands R5 and R6 are implemented in a single Goroutine since both of them talk about receiving a message. A straight-forward case distinction then decides which rule to execute.
The implementation is fair, as long as the scheduler is *weakly fair*.

Theorem 4 (Correctness). *The implementation described in the previous section does not deny any valid execution of the algorithm.*

Proof sketch.

(1) Since a philosopher starts in a *tranquil* state, only rule R1 (becoming thirsty) can be executed.
(2) After executing R1, rules R2 (start drinking) or R4 (requesting a bottle). Since, in general, a philosopher does not own all required bottles, R2 is chosen to be executed after R4. This order does not deny any valid execution because the execution of R4 is skipped, if all required bottles are already owned.
(3) R3 is only executed if `Release(.)` is called. Since this can only be done after a successful invocation of `Request(.)`, the guard of R3 holds.
(4) The Goroutine executing `handleMessages` continuously checks whether the guards of R5 and R6 are valid. If R5 (receiving a request) is executed, only R4's guard might become true again. If R6 (receiving a bottle) is executed, R2's guard might become true. The latter case is handled by signaling the waiting condition; the first is handled by explicit rechecking (and possibly executing). □

This proof can be formalized by examining the initial state of the algorithm, and then successively build up sets of possible following rule executions.

5 Related Work

Resource allocation in distributed networks has been researched intensively. A few selected development are now mentioned. A generalization is already discussed in the paper of Ginat et al. which contributed the solution discussed in sect. 3.2: Instead of associated one bottle per edge, multiple bottles (of different type) can be associated to each edge.

Another solution to the drinking philosophers problem by Lynch is based on partial ordering of resources [13]. An improvement was proposed by Styer and Peterson which uses locality of resources to optimize the waiting time [14]. Another approach is developed by Lehmann and Rabin; they use a non-deterministic approach for the dining philosopher problem [9]. It may be possible to apply an analogous technique for distributed problems.

6 Conclusion

In this seminar paper, a short historical review of the topic is given. The Drinking Philosophers Problem is introduced as a generalization of the popular Dining Philosophers Problem. Various correctness properties are discussed and proved for the solution of Ginat et al. Finally, a proof-of-concept implementation in Go is given and discussed.

"Sandy Murphy and Udaya Shankar, two researchers at the University of Maryland, recently received a reprint request for their article 'A note on the Drinking Philosophers Problem,' published in Transactions on Programming Languages and Systems.
Not too unusual, except that the request came from the Research Institute on Alcoholism in Buffalo." [16]

References

1. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (1999)
2. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9) (September 1965) 569–
3. Dijkstra, E.W.: The origin of concurrent programming. Springer-Verlag New York, Inc., New York, NY, USA (2002) 65–138
4. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Inf. **1** (1971) 115–138
5. Dijkstra, E.W.: Two starvation-free solutions of a general exclusion problem. (1977)
6. Chang, E.J.H.: n-philosophers: An exercise in distributed control. Computer Networks **4** (1980) 71–76
7. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Trans. Program. Lang. Syst. **6**(4) (October 1984) 632–646
8. Welch, J.L., Lynch, N.A.: A modular drinking philosophers algorithm. Distrib. Comput. **6**(4) (July 1993) 233–244
9. Lehmann, D., Rabin, M.O.: On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '81, New York, NY, USA, ACM (1981) 133–138
10. Ginat, D., Shankar, A., Agrawala, A.: An efficient solution to the drinking philosophers problem and its extensions. In Bermond, J.C., Raynal, M., eds.: Distributed Algorithms. Volume 392 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1989) 83–93
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8) (August 1975) 453–457
12. Murphy, S.L., Shankar, A.U.: A note on the drinking philosophers problem. ACM Trans. Program. Lang. Syst. **10**(1) (1988) 178–188

13. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: Proceedings of the twelfth annual ACM symposium on Theory of computing. STOC '80, New York, NY, USA, ACM (1980) 70–81
14. Styer, E., Peterson, G.L.: Improved algorithms for distributed resource allocation. In: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing. PODC '88, New York, NY, USA, ACM (1988) 105–116
15. Awerbuch, B., Saks, M.: A dining philosophers algorithm with polynomial response time. In: Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on. (1990) 65–74 vol.1
16. Drinking Philosophers [rec.humor.funny]. http://www.netfunny.com/rhf/jokes/89q2/drinking.319.html Accessed: 26.11.2013.