

Chapter 11

Sets, Bags, Sequences, Arrays, Records

This section introduces notation for sets, bags, sequences, “generalized” arrays, and records. The notation is for use in programs as well as in assertions. For notational convenience, we use ∞ to denote a value higher than any integer.

11.1 Sets

A set is a collection of values where duplicates are not distinguished. The construct

$$\text{SetOf}(Z)$$

where Z is a type, defines the type of sets of Z . For example, a value of type $\text{SetOf}(\text{int})$ is a set of integers.

The empty set is denoted by \emptyset . For a set S , $S.\text{size}$ is the number of elements in S if S is finite, and the special value ∞ if S is infinite. We use standard mathematical notation for many operations: \in (membership), \cup (union), \cap (intersection), \subseteq (subset), \subset (proper subset), \supseteq (superset), \supset (proper superset), $-$ (set minus), \times (cartesian product), etc.

The construct

$$\text{set}(E :: Z \times, P)$$

where Z is a type, set, bag, or sequence, \times is a parameter, P is a predicate in \times , and E is an expression in \times , defines the set of values of E for \times ranging over the elements of Z such that P holds. For example, $\text{set}(2 * i :: \text{nat } i, i \leq 4)$ is the set $\{0, 2, 4, 8\}$. If no value of \times satisfies P , then the construct returns the empty set. P can be omitted, in which case it is taken to be true. P can be a list of predicates, in which case it is equivalent to the conjunction of the predicates. E can be a list of expressions, in which case each expression contributes elements to the set (i.e., set union). E can be an enumerated list of constant expressions, in which case the “ $::$ ” and everything to its right are omitted.

Here are some examples:

- $\text{set}(2 * i :: \text{nat } i)$ is the set of non-negative even integers.
- $\text{set}(2 * i :: \text{nat } i, i > 100, i < 200)$ is the set of even integers between 100 and 200.
- $\text{set}(3 * i, 4 * i :: \text{nat } i)$ is the union of the sets $\{0, 3, 6, \dots\}$ and $\{0, 4, 8, \dots\}$.
- $\text{set}(i :: S, i > 7)$, where S is a set of integers, is the set of elements of S exceeding 7. Another way to define this set is $\text{set}(i :: \text{int } i, i \in S, i > 7)$.
- $\text{set}(0, 1, 4, 5, 0, 1)$ is the set $\{0, 1, 4, 5\}$.
- $\text{set}(x :: \text{SetOf}(Z) \times, x \subseteq y)$, where y is of type $\text{SetOf}(Z)$, is the set of subsets of y .
- $\text{set}(x :: \text{SetOf}(Z) \times, x \subseteq y, x.\text{size} = 2)$, where y is of type $\text{SetOf}(Z)$, is the set of 2-subsets of y .

11.2 Bags

Bags are sets where duplicates are distinguished. The construct

$$\text{BagOf}(Z)$$

where Z is a type, defines the type of bags of Z .

The empty bag is denoted by \emptyset . For a bag S , $S.\text{size}$ is the number of elements in S if S is finite, and the special value ∞ if S is infinite. The operations defined for sets are also applicable to bags. If bag S has no duplicates, then S and $\text{set}(x :: S\ x)$ are equal.

The construct

$$\text{bag}(E :: Z\ x, P)$$

where Z is a type, set, bag, or sequence, x is a parameter, P is a predicate in x , and E is an expression in x , defines the bag of values of E for x ranging over the elements of Z such that P holds. If no value of x satisfies P , then the construct returns the empty bag. As with the `set` construct, P can be omitted, P can be a list of predicates, E can be a list of expressions, and E can be an enumerated list of constant expressions (in which case the “`::`” and everything to its right are omitted).

Here are some examples:

- $\text{bag}(1, 2, 4, 4, 2)$ is the bag of one 1, two 2s and two 4s (whereas $\text{set}(1, 2, 4, 4, 2)$ is the set of one 1, one 2 and one 4).
- $\text{bag}(3 * i, 4 * i :: \text{nat } i)$ is the union of the bags $\{0, 3, 6, \dots\}$ and $\{0, 4, 8, \dots\}$; note that every integer divisible by both 3 and 4 appears twice in the bag.
- $\text{bag}(2 * i :: S\ i)$, where S is a bag of integers, the bag obtained by doubling every element in S (so the bag has duplicates iff S has duplicates).

11.3 Sequences

Sequences are ordered bags. The construct

$$\text{SeqOf}(Z)$$

where Z is a type, defines the type of sequences of Z . For example, a value of type $\text{SeqOf}(\text{int})$ is a sequence of integers.

The empty sequence is denoted by $\langle \rangle$. Let S be a sequence. $S.\text{size}$ is the number of elements in S if S is finite, and the special value ∞ if S is infinite. The elements of S are indexed by the naturals, starting with 0 at the head. $S.\text{dom}$ is the sequence of indices of S ; it is the sequence $0, 1, \dots, S.\text{size} - 1$ if S is finite, and the sequence $0, 1, \dots$ if S is infinite.

The construct

$$\text{seq}(E :: Z\ x, P)$$

where Z is an ordered type or a sequence, x is a parameter, P is a predicate in x , and E is an expression in x , defines the sequence of values of E for x ranging over the elements of Z such that P holds. *The order of the elements in the derived sequence is induced from the ordering in Z .* If no value of x satisfies P , then the construct returns the empty sequence. P can be omitted, in which case it is taken to be true. P can be a list of predicates, in which case it is equivalent to the conjunction of the predicates. E can be an enumerated list of constant expressions, in which case the “`::`” and everything to its right are omitted.

Here are some examples:

- $\text{seq}(1, 2, 4, 4, 2)$ is the sequence consisting of 1, 2, 4, 4, 2 (in that order).
- $\text{seq}(4 * i :: \text{nat } i, 4 * i > 12)$ is the sequence of naturals (in increasing order) that are divisible by 4 and greater than 12.
- $\text{seq}(x :: S\ x, P)$, where S is a sequence, is the subsequence of S whose elements satisfy P .

- $\text{seq}(i :: \text{nat } i, 0 \leq i < S.\text{size})$, where S is a sequence, is equal to $S.\text{dom}$
- $\text{seq}(S[i] :: S.\text{dom } i)$, where S is a sequence, is equal to S .

We next introduce notation for common operations on sequences. These operations can be formalized in terms of the `seq` construct.

Let S be a sequence. $S.\text{head}$, for nonempty S , is $S[0]$. $S.\text{last}$, for a nonempty finite S , is $S[S.\text{size} - 1]$. $S.\text{tail}$ is S without the head, i.e., $\text{seq}(S[i] :: S.\text{dom } i, i > 0)$; it is the empty sequence if S has zero or one elements. $S[j..k]$, for integers j and k in $S.\text{dom}$, denotes $\text{seq}(S[i] :: S.\text{dom } i, j \leq i \leq k)$; it is empty if $j > k$. $S[j, k]$, for distinct integers j and k in $S.\text{dom}$, denotes $\text{seq}(S[j], S[k])$.

Let S and T be sequences. $S \text{ pfxOf } T$ denotes S is a prefix of T . $S \text{ sfxOf } T$ denotes S is a suffix of T . $S \text{ sbsqOf } T$ denotes S is a subsequence of T . $S \circ T$, for a finite sequence S and a sequence T , denotes the sequence obtained by concatenating T to the end of S . For sequence S and set of sequences Y , $S \text{ mrgOf } Y$ denotes S is a merge of the sequences in Y .

Let S be a sequence variable. $S.\text{append}(e)$, for a value e , denotes $S := S \circ \text{seq}(e)$; i.e., concatenate e to the end of S . $S.\text{remove}()$, for a nonempty S , denotes $\text{return}(S[0]); S := S.\text{tail}$; i.e., remove the head of S and return it.

11.4 Generalized arrays

Recall that $X[0..N - 1]$, where X is a type, defines the type of arrays of X indexed $0, \dots, N - 1$. We extend this notation to arrays on arbitrary types and arbitrary sets. Let X be a type or set. Let Y be a type. The type of arrays from X to Y is denoted by

$$Y[X]$$

So

$$Y[X] \ w := z$$

defines variable w to be an array from X to Y and initializes every entry to z . The “ $:= z$ ” can be omitted, in which case w is not initialized. For any v in X , $w[v]$ denotes the v th entry of w . If X is a set variable and is extended by an item u , then $w[u]$ is initialized to the initial value (if any) in w ’s declaration. We use $w.\text{dom}$ to denote the domain of w , that is, X .

11.5 Records

A record consists of named fields of varying types. The construct

$$Y = \text{RecordOf}(\begin{array}{l} X1 \ a1; \\ X2 \ a2; \\ \dots \\ Xn \ an; \end{array})$$

where $X1, X2, \dots, Xn$ are types, defines Y to be the type of records with named fields $a1, a2, \dots, an$ where a_i is of type X_i . A field can have an initial value, as in $X2 \ a2 := b$. A field can also be augmented by a predicate in previous fields, meaning that the field exists only if the predicate holds. For example,

$$Y = \text{RecordOf}(\begin{array}{l} \text{int } a1; \\ \text{int } a2 \ \text{if } a1 = 0; \end{array})$$

defines Y to be records of the type $\langle i \rangle$ and $\langle 0, j \rangle$, where $i \neq 0$ and j is any integer.

Given the record type Y , the construct

$$Y(v1, v2, \dots, vn)$$

where v_1, \dots, v_n are constant expressions, denotes a record of type Y with $a_1 = v_1, a_2 = v_2, \dots, a_n = v_n$.

Given a value w of type Y , the fields of w can be addressed via the usual “dot” notation, e.g., $w.a_1$. For notational brevity, we allow the syntax $[\text{for } w :: \dots]$ to mean that field names of Y in \dots refer to the fields of w , e.g., the predicate $[\text{for } w :: a_1 = 5 \wedge a_2 > 3]$ or the statement $[\text{for } w :: \text{if } a_2 > 3 \text{ then } a_1 := 5]$.

Chapter 12

Diners and Drinkers Services

12.1 Introduction

This chapter defines two successive generalizations of the lock service, called the diners service and the drinkers service. The diners service is parameterized by a finite undirected graph G . The users of the service are associated one-to-one with the vertices of G . Users request locks, acquire locks, and release locks. The offerer ensures that neighboring users, i.e., users that share an edge of G , do not have locks at the same time, and that every request is eventually satisfied. In terms of thinking, hungry, and eating, neighboring processes should not eat simultaneously, and every hungry process should eventually eat. The classical dining philosophers problem is the special case where G is a cyclic graph with five vertices. The mutual exclusion problem is the special case where G is a complete graph (i.e., an edge between every pair of nodes).

The drinkers service is a generalization of the diners. As in the diners service, users are associated with the vertices of a finite undirected graph G . Each user cycles between tranquil, thirsty, and drinking. When a user becomes thirsty, it designates a non-empty subset of its incident edges to drink from. Different drinking sessions can involve different subsets. The goal is to ensure that neighboring users do not drink from the same edge simultaneously, and that every thirsty process eventually eats. Thus the diners problem is the special case where a thirsty user needs to drink from all its incident edges.

Dijkstra introduced the diners problem (he called it the general exclusion problem) and provided two solutions [2]. Chandy and Misra introduced the drinkers problem and provided distributed solutions to it and to the diners problem [1]. These solutions are described in later chapters. This chapter specifies the diners service and the drinkers service.

We use the following notation for graph G . $G.vset$ is the set of vertices of G , i.e., the user ids. $G.eset$ is the set of edges. $G.uedge(i,j)$ is a boolean function that is true iff G has an edge between i and j . $G.nbs(i)$ is set of neighbors of vertex i .

12.2 Diners service

The diners service program is a straight-forward extension of the lock service program.

```
service-program DinersSrv(G) { // diners service parameterized by undirected graph G
  init() {
    int T := 0, H := 1, E := 2; // T=thinking, H=hungry, E=eating
    int[G.vset] st := T; //st[i] is T, H, or E; initially T.
  }

  dnw void Req( G.vset i ) {
    ec st[i] = T
    ac st[i] := H
  }
}
```

```

upw void Acq( G.vset i ) {
  ec st[i] = H and [∀ G.nbs(i) j : st[j] ≠ E]
  ac st[i] := E
}

dnw void Rel( G.vset i ) {
  ec st[i] = E
  ac st[i] := T
}

progress-obligations() {
  [ ∀ G.vset i:: st[i]=E ~> st[i]=T ] // if every eating user eventually stops eating
  ⇒ [ ∀ G.vset i:: st[i]=H ~> st[i]=E ] // then every hungry user eventually eats
}
} // diners service

```

12.3 Drinkers service

The drinkers service program is like the diners service program except that a request by user i indicates a subset of neighbors; the bottles shared with these neighbors are the ones needed for drinking.

```

service-program DrinkersSrv(G) { // drinkers service parameterized by undirected graph G
  init() {
    int Tr := 0, Th := 1, Dr := 2; // Tr: tranquil; Th: thirsty; Dr: drinking
    int[G.vset] st := Tr; // st[i] is Tr, Th, or Dr; initially Tr
    SetOf(G.vset)[G.vset] need := ∅; // i needs bottles shared with users in need[i]
  }

  dnw void Req( G.vset i, SetOf(G.vset) D ) {
    ec (st[i] = Tr) and (D ⊆ G.nbs(i)) ∧ (¬D.empty)
    ac st[i] := Th; need[i] := D
  }

  upw void Acq( G.vset i ) {
    ec (st[i] = Th) and [forall need[i] j : st[j]≠Dr ]
    ac st[i] := Dr
  }

  dnw void Rel( G.vset i ) {
    ec st[i] = Dr
    ac st[i] := Tr;
    need[i] := ∅
  }

  progress-obligations() {
    [ ∀ G.vset i:: st[i]=Dr ~> st[i]=Tr ] // if every drinking user eventually stops drinking
    ⇒ [ ∀ G.vset i:: st[i]=Th ~> st[i]=Dr ] // then every parched user eventually drinks
  }
} // drinkers service

```

12.4 Concluding Remarks

The diners and drinkers service specifications do not say whether the offerer is to run on a centralized or distributed platform. A distributed solution would have a component at each user location and the components would interact only via message passing.

Although the diners and drinkers services are generalizations of the lock service, note that a system that offers the lock service also offers the diners service and the drinkers service. Of course, it does not offer it as efficiently as one would like. One would like a diners solution that lets non-neighboring users eat simultaneously whenever possible. Similarly, one would like a drinkers solution that lets neighboring users drink simultaneously as long as both do not need their shared bottle. It is easy to achieve such solutions with the timestamp mechanism. For example, the diners solution would be like the mutual exclusion solution except that a hungry user can eat when its request queue has no request from a neighbor in G with a smaller extended timestamp.

In a distributed solution to the diners (or drinkers) service, one would also like that the component at a location is not disturbed if the associated user is not interested in eating (or drinking). That is, the component of a “fasting” user should not receive or send messages to components at “feasting” users. This does not hold in the timestamp solution outlined above; there, every request is received and acked by every node in G . We will see solutions that do not suffer from this.

An interesting question is whether a distributed solution can be *fully-symmetric*, which means that every process executes the same program *and* starts from the same initial condition (including process id). It turns out that the only way to get a fully-symmetric solution is to resort to *probabilistic* algorithms [6]. There are solutions that achieve the desired safety with absolute certainty and the desired progress with very high probability.

12.5 Exercises

- 12.1 Obtain a distributed system that uses the timestamp mechanism to offer the diners service.
- 12.2 Obtain a distributed system that uses the timestamp mechanism to offer the drinkers service.
- 12.3 Extend the drinkers service to one where each edge has multiple bottles and a drinking request indicates the number of bottles needed from each incident edge.

Chapter 13

Two Centralized Diners Solutions

13.1 Introduction

This chapter describes two systems that offer the diners service, obtained from the diners solutions in [2]. The first solution is fully-symmetric but it assumes rather powerful atomicity from the underlying platform and hence can not be transformed into a distributed solution. The second solution assumes reasonable atomicity and can be easily converted to a distributed solution.

In each case, the solution can be conveniently analyzed in terms of a directed graph W on the users in which a directed edge from u to v means that u and v are neighbors in G , v is non-thinking, and u will eat only after v eats. If the directed edge from u to v also means that u is non-thinking, then W can be thought of as a “wait-for” graph.

13.2 Solution 1

The solution maintains a state for every pair of neighbors u and v , indicating whether the edge between them has an arrow directed to u , or an arrow directed to v , or no arrow at all. The solution ensures that the edge is directed from u to v iff both u and v are non-thinking and u became hungry after v (and will eat only after v eats). Initially, all users are thinking and all edges are undirected.

The solution executes the following:

- L0: When a user becomes hungry, all its edges to non-thinking neighbors are directed outward in one atomic action.
- L1: A hungry user can eat when it has no outward edge.
- L2: When a user becomes thinking, all inward edges are made undirected.

Let variable W be the directed graph on the vertices of G defined by the set of directed edges. W is a wait-for graph. Initially, W has no edges. L0 adds edges to W . L2 removes edges from W .

Safety

We first establish safety. Because L0 is the only arrow-creating action, each arrow is created by its source. From this and the initial state, we conclude that the following are invariant:

- P1: (u has an outward arrow) \Rightarrow (u is hungry)
- P2: (u has an inward arrow) \Rightarrow (u is non-thinking)
- P3: (u is thinking) \Rightarrow (u 's incident edges have no arrows)
- P4: (u is eating) \Rightarrow (u has no outward arrows)

P5: (u and v are a pair of non-thinking neighbors) \Rightarrow (there is an arrow between them)

Each of P1, P2, and P4 satisfies the invariance rule. P1 and P2 imply P3. P5 satisfies the invariance rule assuming P4. P5 and P1 imply that of each pair of non-thinking neighbors, at least one must be hungry. Hence, eating neighbors are excluded.

Progress

We now establish progress, i.e., that every hungry user eventually eats. This is not trivial because a hungry user may be waiting on another hungry user. We first establish that deadlock is not possible.

Q1 : W is acyclic

Q1 holds initially. L0 does not create a cycle in W because the only arrows it introduces are outward from a thinking node u , which, by P3, had no arrows prior to L0's execution. L1 and L2 do not create arrow and so cannot create a cycle in W . So \square Q1 holds.

To establish starvation freedom, we must show that if a hungry user u has one or more outward arrows, then these arrows eventually disappear. Let $W(u)$ be the set of vertices reachable from u in W . Because W has no cycles, every path in W outgoing from u ends in a user, say v , with no outward arrows. User v is either eating or hungry with L1 continuously enabled (no other user can introduce an outward arrow for v). Hence v will eventually finish eating and do L1, at which point v leaves $W(u)$. So eventually $W(u)$ will be empty and u will eat. To summarize, the following hold: which imply the desired progress property:

Q2: $((u \text{ is hungry}) \text{ and } (W(u) = f > 0)) \rightsquigarrow ((u \text{ is hungry}) \text{ and } W(u) < f)$

Q3: $((u \text{ is hungry}) \text{ and } W(u) = 0) \rightsquigarrow (u \text{ is eating})$

The desired progress condition follows from Q2 and Q3.

System

It is straightforward to cast the above solution as a system that offers $\text{DinersSrv}(G)$.

```

system-program diners-sys-ewd1(UGraph G) { // G undirected finite graph
  init() {
    int T := 0, H := 1, E := 2; // T=thinking, H=hungry, E=eating
    int[G.vset] z := T; // z[i] is status of user i, initially T
    DEdge = RecordOf( G.vset src, dst ); // directed edge
    SetOf(DEdge) W :=  $\emptyset$ ; // wait-for edges

  xc-event Req(G.vset u)
    maps DinersSrv(G).Req(u)
    ec z[u] = T
    ac z[u] := H;
    for every v in G.nbs(u) do
      if z[v]  $\neq$  T then W := W  $\cup$  { $\langle u, v \rangle$ }

  lc-event bE(G.vset u)
    ec z[u]=H and [forall G.nbs(u) v:  $\langle u, v \rangle$  not in W]
    ac z[u] := E;
    DinersSrv(G).Acq(u)

  xc-event void Rel(G.vset u)
    maps DinersSrv(G).Rel(u)
    ec z[u] = E
    ac z[u] := T;
    for v in nbs(u) do W := W - { $\langle v, u \rangle$ }

```

```

    fairness-assumptions{ Wfair(bE(u)) }
} // diners-sys-ewd-1

```

13.3 Solution 2

The second solution associates with each edge e a (binary) semaphore, say $lck[e]$, that is initially 1. A hungry process eats only after doing P operations on the semaphores of its incident edges. When a process stops eating, it does V operations on all the semaphores of its incident edges. The only atomicity required from the underlying platform is that of P and V operations:

To avoid deadlock, the solution imposes a total ordering on the edges of G and does its P operations in increasing order. The V operations can be done in any order, but for notational convenience of analysis, we assume that they also are done in increasing order.

So user u executes $entry(u)$ before eating and $exit(u)$ after eating, where $G.edges(u)$ is the set of edges in G incident on user u .

```

entry(u) { // executed by u before eating
    for e in G.edges(u) in increasing order do P(lck[e]);
}

```

```

exit(u) { // executed by u after eating
    for e in G.edges(u) in increasing order do V(lck[e]);
}

```

Safety

Safety is easily established. A semaphore is held by at most one of its incident users, because the semaphore is initially one and a user does a V on it only if the user holds the semaphore. An eating user holds the semaphores of all its incident edges. So neighbors cannot eat simultaneously.

To cast this as an assertional proof, define the following predicates for neighbors u and v and their shared edge e :

P1: $(lck[e] = 1 \text{ and neither } u \text{ nor } v \text{ hold } lck[e])$
or $(lck[e] = 0 \text{ and } u \text{ holds } lck[e])$ or $(lck[e] = 0 \text{ and } v \text{ holds } lck[e])$

P2: $(u \text{ at } P(lck[e])) \Rightarrow (u \text{ holds an incident edge } f \text{ iff } f < e)$

P3: $(u \text{ is eating}) \Rightarrow (u \text{ holds all incident edges})$

P4: $(u \text{ at } V(lck[e])) \Rightarrow (u \text{ holds an incident edge } f \text{ iff } f > e)$

P1 through P4 together satisfy the invariance rule. P1 and P3 imply that neighbors cannot eat simultaneously.

Progress

We now turn to progress. Suppose user u is at $P(lck[e])$, where e is shared with v . It sufficient and necessary to establish that u holds $lck[e]$ eventually. Note that this need not hold if $P(lck[e])$ is executed with weak fairness (because then v can repeatedly acquire and release the lock). So our task is to show that u eventually acquires $lck[e]$ assuming that $P(lck[e])$ is executed with strong fairness.

First, define *auxiliary* variable W to be the directed graph on the vertices of G such that there is a directed edge from u to v iff u and v are neighbors in G and v holds $lck[e]$. and u is at $P(lck[e])$. Initially, W has no edges. P2 implies that W is acyclic at all times.

Suppose u is at $P(lck[e])$. Introduce the following auxiliary variables:

W_u = the sequence of vertices in the maximal directed path in W starting from u .

$S_u(i)$, for $0 \leq i < W_u.size$, the edge (of the semaphore) of the P operation that $W_u(i)$ is at.

$R_u(i)$, for $0 \leq i < W_u.size$, the number of times that $S_u(i)$ has been released since $W_u(i)$ came to $P(S_u(i))$.

Let $F(u)$ be the tuple $\langle R_u(0), S_u(0), R_u(1), S_u(1), \dots, R_u(W_u.size - 1), S_u(W_u.size - 1) \rangle$. Let the values of $F(u)$ be lexicographically ordered with the anchor point at the left.

$F(u)$ keeps increasing monotonically as long as u does not acquire the lock. Specifically, $F(u)$ can change in only the following ways as long as u is hungry:

1. W_u gets extended because the edge that the final vertex in W_u needs gets grabbed by its neighbor.
2. The final vertex in W_u grabs an edge and begins to wait on the next edge. In this case, $S_u(W_u.size - 1)$ increases (because edges are grabbed in increasing order).
3. The final vertex in W_u grabs an edge and becomes eating. In this case, it eventually stops eating and releases the edge it shares with the pre-final vertex, which results in $R_u(W_u.size - 2)$ increasing.
4. The edge that the final vertex is waiting for becomes available, in which case $R_u(W_u.size - 1)$ increases.

Furthermore, one of these is always enabled and will eventually occur. So $F(u)$ keeps increasing unless u eats. An S_u entry cannot increase indefinitely because the graph is finite. An R_u entry cannot increase indefinitely because P operations are strong fair. So u eventually eats.

System

It is straightforward to cast the above solution as a system that offers $DinersSrv(G)$. The xc event that maps $DinerSrv(G).Req(u)$ starts a thread executing a request handler function. The request handler consists of $entry(u)$ followed by a $DinerSrv(G).Acq(u)$ call. The xc event that maps $DinerSrv(G).Rel(u)$ executes $exit(u)$ (or starts a thread that executes $exit(u)$). The details are left as an exercise.

13.4 Concluding Remarks

The first solution requires the underlying execution platform to provide rather powerful atomicity, specifically, to execute $L0$ atomically. This is not easy to achieve in a distributed environment.

The second solution requires much weaker atomicity. It can be easily transformed to a distributed solution consisting of a component system, say $Z(u)$, at each location u . The lck semaphores can be effected by message passing as follows. For every edge, introduce two tokens, a “fork” which can be clean or dirty, and a “request-for-fork”. At any time, each token is at one of the locations incident on the edge or in transit from one to the other. Initially, the fork is dirty and at one location and the request is at the other location.

- $P(lck)$ operation is replaced by the following: if the corresponding fork is present, make it clean otherwise send the request-for-fork and wait for the fork to arrive.
- $V(lck)$ operation is replaced by the following: dirty the fork; if the request-for-fork is present, send the fork to the neighbor.
- Add an xc -event for reception of fork. Upon receiving a fork, make the fork clean (and consider the corresponding $P(lck)$ operation completed).
- Add an xc -event for reception of request-for-fork. Upon receiving a request-for-fork, if the fork is dirty, send it off. (If the fork is not dirty, the component has acquired it and will release it after eating.)

Note that this distributed solution requires a total ordering on the edges. We will see in the next chapter how the total ordering can be replaced by something much weaker (specifically, an acyclic partial ordering).

13.5 Exercises

13.1 Does $\text{diners-sys-ewd-1}(G)$ satisfy $\text{DinersSrv}(G)$? Prove or disprove.

13.2 In solution 2, suppose that P operations were only weak fair. Does this still satisfy P_0 ? If you answer no, provide a counter-example execution. If you answer yes, provide a total-ordering (F, \prec) where F is a function of the system variables (and any auxiliary variables you define) such that each of P_1 and P_2 given below satisfies the leads-to rule:

$$P_1 : u[i].\text{st} = H \wedge F = j \succ 0 \rightsquigarrow u[i].\text{st} = E \vee F \prec j$$

$$P_2 : u[i].\text{st} = H \wedge F = 0 \rightsquigarrow u[i].\text{st} = E$$

13.3 Repeat problem 13.2 but now assume that every semaphore obeys the additional (safety) property that waiting processes are served in fifo order.

13.4 Fill out the details of the distributed version of solution 2 outlined above.

Chapter 14

A Hygienic Diners Solution

14.1 Introduction

This chapter describes the distributed diners solution presented in [1]. The solution has a process at each user and fifo channels between every pair of neighboring processes in G . A “fork” token and a “request-for-fork” token is associated with every edge of the graph G . The fork of an edge is either at one of the incident processes or in transit from one to the other. The same holds for the request. A user can eat when its process has the forks of all its incident edges. So when a user becomes hungry, its process attempts to acquire all the incident forks. If the user’s process does not have a fork, it sends the associated request to its neighbor. When the neighboring process receives the request, it sends back the fork *under the appropriate condition*.

The crux, of course, is determining when the process of a user should honor a received request. Clearly, the associated fork must be retained if the user is eating and released if the user is thinking. What about when the user is hungry? If a hungry process always gives up a requested fork, then two hungry neighbors can keep exchanging their shared fork and hence starve. If a hungry process never gives up a requested fork, then a cycle of hungry users can become deadlocked. A priority ordering of hungry users is needed to avoid deadlock and starvation. Naturally, the ordering cannot have a cycle, otherwise there can be deadlock. The ordering cannot be static, otherwise a lower priority user can starve. So we need a dynamic acyclic ordering in which a user’s priority drops after eating.

The previous chapter showed how such an ordering can be implemented given a static total ordering of the edges. The timestamp mechanism provides another way to implement such an ordering given a static total ordering of the user ids (and at the cost of disturbing fasting nodes). This chapter shows how to achieve the desired ordering without requiring any static ordering of the edges or user ids. Essentially, a user u has priority over a neighbor v if v was the last one to use the fork shared by u and v . Initially, the forks are placed with users so that the priority ordering is acyclic. Other than this initial asymmetry, every node has the same rules and initial state.

14.2 Algorithm and analysis

The solution implements an elegant and compact ordering. A fork is either *dirty* or *clean*. Initially, all forks are dirty. A dirty fork becomes clean when it is sent. A clean fork becomes *dirty* when a process eats from it. These are the only occasions when a fork’s status changes. A user u gives up a fork in response to a request if (1) u is thinking, or (2) u is hungry and the fork is dirty. So u defers a received request iff it is eating or it is hungry and the fork is clean.

This results in the following definition of priority: a user u has *priority* over a neighbor v iff (1) u holds their shared fork and the fork is clean, or (2) v holds their shared fork and the fork is dirty, or (3) the fork is in transit from v to u . Note that this priority order changes when and only when u eats.

The priority ordering induces a dynamic directed graph H obtained by directing the edges in G such that for any two neighbors u and v , the edge is directed from u to v iff u has priority over v . The edges change

direction only when a process starts eating; at that time, all outward incident edges become directed inward. This does not introduce any cycles in H . Initially the forks are placed so that H is acyclic. Thus H remains acyclic throughout.

Note that H is a “priority” graph that differs in two ways from the wait-for graph introduced in the previous chapter. First, the edge between two non-thinking neighbors is directed in the opposite way, which is consistent with the fact that u waits for v iff v has priority over u . Second, H has a directed edge between every two neighbors in G , whereas the wait-for graph has a directed edge only between non-thinking neighbors. This “completeness” of H can often simplify notation in analysis, although there is no fundamental difference between H and a wait-for graph.

The algorithm is described below as a set of atomically-executed rules. We use f_e to denote the fork of an edge e , and reqf_e to denote its request-for-fork. When a fork or a request-for-fork is sent, it is always sent on its corresponding edge.

```

Rules at  $u$  {
  When  $u$  becomes hungry {
    for every incident edge  $e$ 
      if  $u$  has  $\text{reqf}_e$ 
         $u$  sends  $\text{reqf}_e$ 
  }

  When  $u$  is hungry and has forks of all incident edges {
     $u$  becomes eating and all its forks become dirty
  }

  When  $u$  becomes thinking {
    for every incident edge  $e$ 
      if  $u$  has  $\text{reqf}_e$ 
         $u$  sends  $f_e$ 
  }

  When  $u$  receives  $f_e$  {
     $u$  sets  $f_e$ 's status to clean
  }

  When  $u$  receives  $\text{reqf}_e$  {
    //  $u$  has  $f_e$ 
    if  $u$  is thinking
       $u$  sends  $f_e$ 
    else if ( $u$  is hungry and  $f_e$  is dirty)
      send  $f_e$ ;
      send  $\text{reqf}_e$ ;
  }

  Start eating is executed with weak-fairness
}

```

We have already seen that the following is invariant (satisfies the invariance rule):

A_1 : H is acyclic

We first establish the desired safety property. A user eats only if it holds all its incident forks. So as long as forks are not created, a pair of neighbors cannot eat simultaneously. The only way a fork can be created is if a node, upon receiving a reqf , sends a fork it does not have. This is precluded because the following is invariant (satisfies the invariance rule):

$A_2 : (\text{reqf in transit to } u) \Rightarrow (\text{corresponding fork is at } u \text{ or in transit to } u \text{ ahead of reqf})$

We now turn to progress. Throughout the following analysis, assume that eating is finite. We need to establish that $(u \text{ is hungry}) \rightsquigarrow (u \text{ is eating})$ holds, for which we must define an appropriate metric with the help of H .

Denote a user with no incoming edge in H as a “root”. Being acyclic, H always has at least one root. Define the depth of a user u , denoted $\text{depth}(u)$, to be the length of a longest path from a root to u . We shall see that the depth of a hungry user, together with the number of clean forks it holds, forms an appropriate metric for establishing progress.

We first show that the depth of a user increases only if the user eats:

$P_1 : (\text{depth}(u) = j) \text{ unless } (\text{depth}(u) < j \text{ or } (u \text{ is eating}))$

Proof of P_1 : H changes only when a user, say v , eats, which causes any outward edge of v to become directed inward. With respect to H , user v can be a successor of u (i.e., on a directed path from u), a predecessor of u (i.e., u is on a directed path from v), or neither a successor nor a predecessor of u .

If v is predecessor of u and there is only one longest path from a root to u and v is on that path, then $\text{depth}(u)$ decreases when v begins eating (it becomes either one less than the previous longest distance from v to u or equal to the previous second longest path from a root to u). If v is predecessor of u and there is a longest path from a root to u that does not go through v , then $\text{depth}(u)$ does not change when v begins eating. (v may or may not be on a longest path from a root to u). If v is a successor of u $\text{depth}(u)$ does not change when v begins eating. If v is neither a successor nor a predecessor, $\text{depth}(u)$ does not change when v begins eating. ■

Next we establish the following progress properties, where $\text{clean}(u)$ is the number of clean forks that u has:

$P_2 : [(u \text{ is hungry}) \text{ and } \text{depth}(u) = 0 \text{ and } \text{clean}(u) = k] \rightsquigarrow$
 $[(u \text{ is hungry}) \text{ and } \text{depth}(u) = 0 \text{ and } \text{clean}(u) = k + 1] \text{ or } (u \text{ is eating})]$

$P_3 : [(u \text{ is hungry}) \text{ and } \text{depth}(u) = 0] \rightsquigarrow (u \text{ is eating})$

Proof of P_2 and P_3 : P_3 follows from P_2 , since a user never gives up a clean fork before eating. We next establish P_2 . Suppose u is hungry and at depth 0 and has k clean forks. Let v be a neighbor of u such that it is not the case that their shared fork is clean and at u . Because u 's depth is 0, the edge with v is directed from u to v . So either (1) the fork is in transit to u , or (2) v holds the fork and it is dirty. If case 1 holds, u eventually gets the fork, increasing $\text{clean}(u)$. If case 2 holds, u 's request for the fork will eventually reach v , at which point either (2a) v is not eating or (2b) v is eating. If case 2a holds, v gives up the fork, resulting in case 1. If case 2b holds, v eventually stops eating, resulting in case 2a. Hence in all cases, u gets the fork and it is clean. ■

We now extend the above to users are non-zero depth:

$P_4 : [(u \text{ is hungry}) \text{ and } \text{depth}(u) = j \text{ and } \text{clean}(u) = k] \rightsquigarrow$
 $[(u \text{ is hungry}) \text{ and } \text{depth}(u) = j \text{ and } \text{clean}(u) = k + 1] \text{ or}$
 $[(u \text{ is hungry}) \text{ and } \text{depth}(u) < j] \text{ or}$
 $(u \text{ is eating})]$

$P_5 : ((u \text{ is hungry}) \text{ and } \text{depth}(u) = j) \rightsquigarrow (u \text{ is eating})$

Proof of P_4 and P_5 : We next establish $P_{4,5}$ by induction on $\text{depth}(u)$. P_3 serves as the base of the induction. Assume $P_{4,5}$ holds for all users with depth less than j . Consider a hungry user u of depth j and with k clean forks. Let v be a neighbor of u such that it is not the case that their shared fork is clean and at u .

1. Suppose $\text{depth}(u) < \text{depth}(v)$. The argument is the same as in the proof of P_2 . Either (1a) the fork is in transit to u , or (1b) v holds the fork and it is dirty. In case 1a, u eventually gets the fork, increasing

$\text{clean}(u)$. In case 1b, u 's request for the fork will eventually reach v , at which point either v is not eating and gives up the fork immediately, or v is eating and will give up the fork when it stops eating. In all cases, $\text{clean}(u)$ increases.

2. Suppose $\text{depth}(u) > \text{depth}(v)$. By the induction hypothesis, v eventually becomes eating, at which point $\text{depth}(u) < \text{depth}(v)$ holds and this reduces to case 1.

So P_4 holds for a user of depth j . P_5 for depth j follows from P_4 for depth j . So we are done. ■

14.3 Offerer specification

From the above solution, it is straightforward to obtain a distributed system that offers the diners service using point-to-point fifo channels between every pair of neighbors in G . The distributed system consists of a set of component systems, one for each vertex u in G . Below, $\text{diners-cm}(G)$ denotes the offerer system, $\text{diner-cm}(G, H_0, u)$ denotes the component system at vertex u , and $\text{fifo}(u, v, \text{Msg})$ denotes the point-to-point channel from u to v . The parameter H_0 defines the initial placement of the forks. It is a directed acyclic graph whose undirected version equals G . If H_0 has an edge directed from u to v , then initially the edge's fork is dirty and placed at v and the edge's reqf is placed at u .

So $\text{diners-cm}(G)$ is the composite system of $\{\text{diner-cm}(G, H_0, u) : G.\text{vset } u\}$, where $\text{diner-cm}(G, H_0, u)$ is defined below.

The code for a diner component would be greatly simplified if a message reception can be immediately followed by a response message send. But the message reception is an xc event, so it cannot contain the send event call. To get around this, we introduce in the diner component an outgoing fifo queue of outgoing message-destination pairs. So a message to be sent is first queued to the fifo buffer, and later dequeued and transmitted into the channel (by an lc spooler event). Because the queueing is not an event call, it can be done in the action of an xc event.

```

system-program diner-cm( UGraph G, DGraph H0, G.vset u ) { // component at u
  init() {
    int T := 0; H := 1; E := 2; // T = thinking, H = hungry, E = eating
    int F := 0, REQF := 1; // F = fork, REQF = request for fork
    Msgs = RecordOf( int tag ); // tag is F or REQF
    int z := T; // status of user u, initially T
    boolean[G.nbs(u)] f; // f[u] true iff u has fork shared with v
    boolean[G.nbs(u)] drty := true; // drty[u] true if fork shared with v is dirty and at u
    boolean[G.nbs(u)] reqf; // reqf[u] true iff u has request for fork shared with v
    // initial placement of forks and requests defined by H0
    for v in G.nbs(u) do { f[v] := H0.edge(v,u); reqf[v] := ¬ f[v] };

    // queue of outgoing message-destination pairs
    OutQEntry = RecordOf( G.vset dst; Msg msg );
    SeqOf(OutQEntry) outq := ⟨⟩; // outgoing message buffer
  } // end initialization

  lc-event Tx() { // spooler: dequeues from outq and transmits
    ec outq().size>0
    ac OutQEntry x := remove(outq);
    fifo(u, x.dst).Tx(x.msg);
  }

  xc-event void Req() { // become hungry
    maps DinersSrv(G).Req(u)
    ec z = T
    ac z := H;
  }
}

```

```

    // send requests for forks not here
    for v in G.nbs(u) do {
      if ¬f[v] then { // reqf[v] is true
        append(outq, OutEntry(v, Msg(REQF)));
        reqf[v] := false
      }
    }
  }

lc-event bE() { // start eating
  ec st=H and [forall G.nbs(u) v :: f[v]] // holding all forks
  ac st := E;
  for v in G.nbs(u) do drty[v] := true;
  DinersSrv(G).Acq(u)
}

xc-event void Rel(G.vset u) { // become thinking
  maps DinersSrv(G).Rel(u)
  ec st = E
  ac st := T;
  // honor deferred requests
  for v in G.nbs(i) do {
    if reqf[v] then {
      outq.Send(j, Msg record(F, u));
      f[v] := false;
      drty[v] := false;
    }
  }
}

xc-event void Rec(G.vset v, Msg m) { // receive fork or request
  maps fifo(v,u).Rec(m)
  ec true
  ac if m.tag = REQF then {
    reqf[v] := true;
    if st=T or ( st=H and drty[v] ) then { // fork not needed or dirty
      Send(v, Msg record(F, u)); // send clean fork
      f[v] := false;
      drty[v] := false;
    }
    if st=H then { // hungry, so fork is needed
      append(outq, OutQEntry(v, Msg(REQF))); // send request
      reqf[v] := false;
    }
  }
  } else if m.tag = F then {
    f[v] := true;
    drty[v] := false;
  }
}

progress-assumptions{ Wfair(*) }
} // diner-cm

```

It is left as an exercise to show whether or not `diners-cm(G)` offers `DinersSrv(G)` using `{fifo(u, v, Msg) :`

$G.vset\ u, v, u \neq v\}$.

14.4 Concluding remarks

This solution is perhaps as close to a fully-symmetric distributed solution as is possible without resorting to probabilistic algorithms. Neither the edges nor the node ids are subject to a static ordering, either partial or total. The only ordering is a dynamic partial one, limited to neighboring nodes.

14.5 Exercises

Chapter 15

A Drinkers Solution using Hygienic Diners

15.1 Introduction

This chapter describes the distributed solution to the drinkers problem presented in [1]. The drinkers solution, like the diners solution there, has a “bottle” token and a “request for bottle” token for each edge of G . A thirsty user can drink when its process has acquired the bottles of the edges the user needs to drink from. But unlike the diners solution, the priority between two users contending for a bottle is not decided by whether or not the bottle is clean.

Instead, the drinkers solution runs a *diners* algorithm, complete with forks and requests for forks, concurrently on the same graph G . The intention is to use the priority ordering of the diners solution to resolve contention for bottles; that is, when two neighbors contend for their shared bottle, the user with the higher diner priority gets to keep the bottle. But this is not exactly what happens, and that is because a user, say u , can determine whether or not it outranks a neighbor v only if u has the fork. If u does not have the fork, then it does not know whether the fork is clean or dirty.

So instead, the priority goes to the user that has the fork, regardless of whether it is clean or dirty. So what does a user do if it needs a bottle and does not have the corresponding fork? It becomes hungry. Once it acquires all its incident forks, it has priority over all its neighbors and so acquires all the needed incident bottles. It stops eating once it starts drinking.

So there are two conflict resolution rules, one for bottles and one for forks. The *bottle* conflict resolution rule is as follows: a process u gives up the bottle shared with a neighbor v in response to a request if (1) u does not need the bottle, or (2) u is not drinking and does not have the fork shared with v . The *fork* conflict resolution rule is as in the diners solution (i.e., u gives up the fork shared with v if u is thinking, or u is not eating and the fork is dirty).

Correspondingly, we have a fork priority ordering, which is the dynamic directed acyclic graph H defined in the diners solution, and a bottle priority ordering, which we shall not explicitly refer to below.

15.2 Algorithm and analysis

Every pair of neighbors u and v share a bottle, a request-for-bottle, a fork, and a request-for-fork. The algorithm is described below as a set of atomically-executed rules, with the diner part in boxes. The diner part is the same as in the previous chapter, and the conventions there also apply here. We use b_e and $reqb_e$ to denote, respectively, the bottle and request-for-bottle of edge e .

```
Rules at  $u$  {
  When  $u$  becomes thirsty {
    record the subset of bottles needed
  }
}
```

When u is thirsty, needs and does not have b_e , and has $reqb_e$ {
 send $reqb_e$
 }

When u is thirsty, thinking, and does not have a needed bottle nor the associated fork {

become hungry;
 for every incident edge e
 if u has $reqf_e$
 u sends $reqf_e$

}

When u is thirsty and has all the needed bottles {
 u becomes drinking;

if u is eating
 u becomes thinking;
 for every incident edge e
 if u has $reqf_e$
 u sends f_e

}

When u becomes tranquil {
 for every incident edge e
 if u has $reqb_e$
 u sends b_e
 }

When u receives $reqb_e$ {
 if (u is tranquil) or (u needs b_e and does not have f_e)
 u sends b_e
 }

```

    When u is hungry and has forks of all incident edges {
      u becomes eating and all its forks become dirty
    }

    When u becomes thinking {
      for every incident edge e
        if u has reqfe
          u sends fe
    }

    When u receives fe {
      u sets fe's status to clean
    }

    When u receives reqfe {
      if u is thinking
        u sends fe
      else if (u is hungry and fe is dirty)
        send fe;
        send reqfe;
    }
  }

```

It is easy to see that the desired safety property holds. A user drinks only if it holds all the needed bottles. Thus a pair of neighbors cannot drink simultaneously from the same bottle.

Progress, although not as easy to show, is still easier to show than in the case of the diners algorithm. Throughout the following, assume that drinking is finite.

We first show that an eating thirsty user u eventually gets the bottles it needs:

$P_1 : (u \text{ is eating and thirsty}) \rightsquigarrow (u \text{ is drinking and thinking})$

Suppose u needs the bottle shared with a neighbor v and v has the bottle. Then u has the request-for-bottle and will send it to v . When v receives the request, there are three cases: (1) if v does not need the bottle, it releases the bottle; (2) if v needs the bottle but is not drinking, it releases the bottle (because v does not have the fork); (3) if v is drinking from the bottle, it eventually stops drinking and then releases the bottle. Thus in all cases, u will get the bottle and not give it up before drinking. So u eventually starts drinking, at which point it stops eating. ■

P_1 establishes that eating is finite in the drinkers algorithm. Hence the progress result for diners, established earlier under the assumption of finite eating, holds here:

$P_2 : (u \text{ is hungry}) \rightsquigarrow (u \text{ is eating})$

P_2 and P_1 establish the desired result:

$P_3 : (u \text{ is thirsty}) \rightsquigarrow (u \text{ is drinking})$

The above solution can be cast as a system that offers $\text{DrinkersSrv}(G)$ using the fifo channels, exactly as in the case of the diners solution. The details are left as an exercise.

15.3 Concluding remarks

Recall that we would like the drinkers solution that (1) allows different users can drink simultaneously from different bottles, and (2) requires the components at two users, say u and v , to interact only if u and v share

an edge and both want to drink from the shared edge. The above drinkers solution meets these conditions almost, but not fully. Consider two neighbors, say u and v , such that u and v drink repeatedly but never from their shared bottle. Hence there is no need for u and v to disturb each other. However, the above solution makes them repeatedly acquire forks and hence disturb each other repeatedly. It is not clear if this can be avoided in theory.

The next chapter presents a timestamp-based solution that avoids this periodic disturbance, but requires occasional global coordination if one is to avoid timestamps in messages from growing unboundedly.

Chapter 16

A Drinkers Solution using Session Numbers

16.1 Introduction

This chapter describes a drinkers solution that uses timestamps [3]. The solution has bottles and requests for bottles as in [1], but it does not involve a diners solution.

Every drinking session is associated with a timestamp, referred to as a *session number*, chosen when the user became thirsty. Each request for a bottle is accompanied by the session number. A thirsty user u gives up a bottle it needs if the session number of the request is less than u 's session number. Ties are resolved in favor of the user with the smaller id. The number of messages per drinking session is of the order of bottles needed, rather than the order of neighbors as in [1], but occasional coordinated resets are needed to bound the session numbers.

16.2 Algorithm

Each user has a distinct id from some total order. (In fact, it suffices if neighbors have different ids; ids of nonneighboring users may be the same.)

Each user u maintains two nondecreasing integer variables, s_num_u and max_rec_u . s_num_u , referred to as u 's session number, identifies u 's last drinking session if u is tranquil, u 's upcoming drinking session if u is thirsty, and u 's current drinking session if u is drinking. max_rec_u indicates the highest session number received by u from its neighbors so far.

We refer to $\langle s_num_u, u \rangle$ as u 's *extended session number*. As usual [5],[7]., $\langle s_num_u, u \rangle < \langle s_num_v, v \rangle$ iff $s_num_u < s_num_v$, or $s_num_u = s_num_v$ and $u < v$. Thus, if u and v are neighbors, their extended session numbers are never equal.

When a user u becomes thirsty, it sets s_num_u to a value higher than max_rec_u . If u needs and does not hold a bottle b shared with neighbor v , it sends to v the request for the bottle accompanied by its session number and id, i.e., the message $(reqb, s_num_u, u)$.

When v receives a request $(reqb, s, u)$, it obeys following *conflict resolution rule*:

v releases b if v does not need b or if v is thirsty and $\langle s, u \rangle < \langle s_num_v, v \rangle$; otherwise, v releases b after it completes drinking.

Note that in either case, v will next send $reqb$ only after v has released b . Thus, when $reqb$ is in transit from v to u , b is either in transit from v to u ahead of $reqb$, or already at u . s_num_u does not change while u is thirsty. Therefore, once b is released by v , it will not return to v before u drinks from it.

Initially, every user u is tranquil and $s_num_u = max_rec_u = 0$. For every pair of neighbors, their shared bottle is with one user and the request is with the other user. Each user u obeys the rules given below:

Upon becoming thirsty {

```

    record the subset of bottles needed;
    s_num_u ← max_rec_u + 1
}

When u is thirsty and has all the needed bottles {
    become drinking
}

When u becomes tranquil {
    for every reqb at u
        Send the corresponding bottle
}

When u needs a bottle and has the corresponding request {
    send ( reqb , s_num_u , u )
}

When u receives a ( reqb , s , v ) {
    max_rec_u ← max ( max_rec_u , s );
    if (u does not need bottle shared with v) or
        (u needs the bottle and ⟨ s_num_u , u ⟩ > ⟨ s , v ⟩)
        then send the bottle to v
}

When u receives a bottle from v {
    no-op
}

```

16.3 Analysis

The following derive from the problem definition and are invariant (each satisfies the invariance rule):

A_0 : For every two users u and v that share an edge: the associated bottle is either held by u or held by v or in transit between u and v .

A_1 : (a) u is not tranquil \Leftrightarrow u needs at least one bottle.
 (b) u is drinking \Rightarrow u holds the bottles it needs.

A_0 and A_1 imply that two neighboring users do not drink simultaneously.

The following predicates relate the locations of $reqb$ and b for any pair of neighbors u and v , and are invariant (satisfy invariance rule assuming A_0):

B_0 : $reqb$ is either at u or at v or in transit between u and v .

B_1 : $reqb$ is in transit to $u \Rightarrow b$ is either held by u or is in transit to u ahead of $reqb$.

B_2 : $reqb$ is held by $u \Rightarrow b$ is not in transit to u .

B_3 : u does not need $b \Rightarrow$
 [(b is not in transit to u) and ($reqb$ is not in transit to v) and
 (neither u nor v holds both b and $reqb$)]

Note that B_1 assures us that b is held by u when u receives $reqb$.

For any two neighboring users u and v , the following predicates relate their extended session numbers and their shared request and bottle. They are invariant (satisfy invariance rule assuming A_0 and $B_{0..3}$):

- C_0 : (b is in transit to u) \Rightarrow
 $[\max_rec_v \geq s_num_u \text{ and } ((v \text{ does not need } b) \text{ or } \langle s_num_u, u \rangle < \langle s_num_v, v \rangle)]$
- C_1 : (u holds b and reqb) \Rightarrow
 $[(u \text{ needs } b) \text{ and } (\max_rec_u \geq s_num_v) \text{ and } ((u \text{ is drinking from } b) \text{ or } \langle s_num_u, u \rangle < \langle s_num_v, v \rangle)]$
- C_2 : (reqb, s, v) is in transit to u \Rightarrow (s, v) = $\langle s_num_v, v \rangle$.
- C_3 : $s_num_u \leq \max_rec_u + 1$

The proof of invariance of the above assertions is as follows. All of them hold initially. A_0 and A_1 are each preserved by every event. $B_{0..3}$ is preserved by every event, assuming A_0 holds before the event execution. $C_{0..3}$ is preserved by every event, assuming A_0 and $B_{0..3}$ hold before the event execution.

C_3 ensures that R1 does not decrease s_num_u . Hence, we have the following:

- C_4 : s_num_u and \max_rec_u never decrease.
- C_5 : s_num_u does not change while u is thirsty.

Let u and v be two neighboring users who share bottle b. We say that b is *dedicated to* u if u is thirsty and needs b and one of the following holds: (1) b is in transit to u; or (2) u holds b and $\langle s_num_u, u \rangle < \langle s_num_v, v \rangle$; or (3) u holds b, v does not need b, and $\langle \max_rec_v + 1, v \rangle > \langle s_num_u, u \rangle$.

C_6 : If b is dedicated to u, then b will not be released by u before u drinks from it.

Proof Consider the period of time while u is thirsty and needs b. From $C_{4,5}$ we know that s_num_u does not change during this period. Given that b is dedicated to u, it can be shown using $B_{1..3}$ and $C_{0..2}$ that a request (REQB, s, v) arriving at u during this period must have (s, v) higher than $\langle s_num_u, u \rangle$, and is therefore deferred. ■

We now prove that every thirsty user eventually drinks, by introducing a dynamic ordering of the users according to their extended session numbers. For any user u, let $pos(u)$ be the number of users whose extended session number is less than u's extended session number. (If nonneighboring users have the same id, then $pos(u)$ would be the number of distinct extended session numbers ahead of u's extended session number.)

We prove progress by proving the following assertion D(i), for any i:

- D(i) : drinking is finite \Rightarrow
 $[(pos(u) = i \text{ and } u \text{ is thirsty}) \rightsquigarrow (u \text{ is drinking})]$

We prove D(i), for any i, by induction on i.

Proof of D(1) The induction basis. We first prove the following assertion:

- E_B : $pos(u) = 1$ and u is thirsty and needs bottle b \rightsquigarrow b is dedicated to u.

Let u be a thirsty user who needs bottle b that u shares with user v, and let $pos(u) = 1$. $\langle s_num_u, u \rangle < \langle s_num_v, v \rangle$ since u is in the lowest class, and two neighboring users cannot be in the same class. From $C_{4,5}$, we know that $\langle s_num_u, u \rangle < \langle s_num_v, v \rangle$ holds continuously during the time that u is thirsty. If b is not already dedicated to u, then b must either be held by v or in transit to v. We show that b will eventually be dedicated to u by examining the possible locations of reqb.

- (1) Assume reqb is at v. b cannot be in transit to v, by B_2 . Therefore, b is held by v, and from C_1 we can infer that v must be drinking from b. v will complete drinking in finite time, at which point rule R3 will be executed at v, resulting in b being sent to u and hence dedicated to u.
- (2) Assume (reqb, s, u) is in transit to v. It will arrive at v in finite time, when b is there (by B_1), and with (s, u) = $\langle s_num_u, u \rangle$ (by C_2). Upon reception of reqb, v will either release b immediately and b will become dedicated to u, or v will defer u's request and we are back in (1).

- (3) Assume reqb is with u . $R4$ is continuously enabled at u . Thus, u will eventually send reqb to v and we are back in (2).
- (4) reqb cannot be in transit to u , by B_1 .

At this point, we can conclude $D(1)$. The number of bottles that u needs for drinking is finite. Each one of these bottles will be dedicated to u and held by u in finite time, after which it will not be released before u drinks, by C_5 . Therefore, once all of these bottles are held by u , $R2$ is continuously enabled at u , and u will eventually start drinking. ■

Proof of $D(k+1)$ assuming $D(1), \dots, D(k)$. The induction step. We first prove the following assertion:

E_5 : $\text{pos}(u) = k+1$ and u is thirsty and needs bottle $b \rightsquigarrow b$ is dedicated to u or u is drinking.

Again, let u be a thirsty user who needs bottle b that it shares with user v , and let $\text{pos}(u) = k+1$. $\text{pos}(v) \neq k+1$ because u and v are neighbors. If $\text{pos}(v) \geq k+2$, then E_5 is proved using the same arguments given in E_B 's proof. We prove E_5 for the case where $\text{pos}(v) \leq k$.

If b is not already dedicated to u , then b is not in transit to u . If b is held by u then it may be held by u until u starts drinking, or it may be released by u (upon receiving v 's request) while u is still thirsty. The latter reduces to the case where b is not held by u .

If b is not held by u , then b is either held by v or in transit to v . We show that b will eventually be dedicated to u by examining the possible locations of reqb :

- (1) Assume reqb is at v . b must be held by v , by B_2 , and v needs b , by C_1 . If v is thirsty, then v will be drinking in finite time, since we assume $D(1), \dots, D(k)$. If v is drinking, it will complete drinking in finite time, and b will become dedicated to u .
- (2) Assume reqb is not at v . The arguments to show that b will eventually be dedicated to u are the same as those used in E_B 's proof under cases (2), (3), (4).

At this point we can conclude $D(k+1)$. Either u will start drinking before all the bottles u needs are dedicated to u , or u will first have all the bottles it needs dedicated to itself and then have $R2$ continuously enabled and start drinking. ■

16.4 Bounding session numbers

The session numbers, and hence the size of request messages, in the above solution can not be upper bounded, and the reason for this is that a user u interacts with a neighbor v only if v has a bottle that u needs to drink. Thus two philosophers may drink repeatedly from a bottle they share and increment their session numbers, while no change occurs in the session number of a third user who is their neighbor (because it does not want to drink).

To bound session numbers, we require user u to reset its max_rec to zero if it wants to drink when max_rec equals some bound L . If u resets its max_rec without coordinating the reset with its neighbors, starvation can occur, and this is true for any value of L . We now outline one of several possible solutions for a coordinated reset of the max_rec variables.

We suggest a global reset protocol coordinated by a distinguished user r , who is the root of a fixed spanning tree T defined over G . At each user u , introduce a boolean variable resetting , which is true iff u is participating in a coordinated reset, and a boolean variable waiting , indicating that u is waiting for a coordinated reset.

If a tranquil user u wants to drink when its max_rec equals L , then instead of becoming thirsty it enters the waiting state and sends a request_reset message to its parent in T . This message is propagated up the tree T to r . When r receives the request_reset message, it starts a **two-phase reset protocol**. Each phase is a diffusion computation in T ; the first involves acknowledgments and the second does not.

In the first phase, r sets its resetting to true and sends a hold message to each of its children in T . Each user v , upon receiving a hold message from its parent in T , sets its resetting to true and propagates the hold message to each of its children (if any) in T . If a tranquil user v whose resetting is true wants to drink,

it enters the waiting state (irrespective of its `max_rec` value) instead of becoming thirsty. Each user v with `resetting` true sends an `ack` message to its parent when it is not thirsty and has received an `ack` message from each of its children (if any). Thus a user v who was thirsty when its `resetting` became true sends its `ack` message only after it starts drinking. When r receives `ack` messages from all its children and r is not thirsty, no user is thirsty.

At this point, r starts the second phase. Here, `reset` messages are propagated down the tree T . Each user, upon receiving a `reset` message (and r upon starting the second phase), sends `reset` messages to all its children in T , sets its `resetting` to false, zeros its `max_rec`, and executes the `become thirsty` rule if waiting. At this point, the user stops its participation in the reset protocol.

In order to have only one coordinated reset when simultaneous reset requests occur, it suffices if a user v ignores a `request_reset` message received while its `resetting` is true or if v has propagated a `request_reset` message but its `resetting` is not true yet.

The coordinated reset may cause one extra retransmission of each bottle, as follows. Suppose a philosopher u has its `resetting` true and its neighbor v has already finished participating in the reset. u may receive a request for the bottle from v and release the bottle. After u ends its participation in the reset, it may need the bottle and request it with a session number lower than v 's. If u 's request arrives at v while v is still thirsty (waiting for other bottles), v releases the bottle before drinking. v will request b again, and this time will not release it before drinking.

Each execution of the reset protocol costs between $3(N-1)$ to $4(N-1) + 2|E|$ messages: $3(N-1)$ messages for the protocol's two phases, upto $N-1$ messages for simultaneous requests of its execution, and upto $2|E|$ messages due to retransmission of bottles. A user may be blocked in the waiting state for the duration of the protocol. If no user is thirsty upon receiving a `hold` message, then the duration of the protocol is the time to propagate messages over $3D$ successive hops, where D is the diameter of T . If some users are thirsty when the propagation of `hold` messages is completed and the maximum waiting chain length is W , $0 < W < N$, then the protocol duration will also include the time to perform W successive drinking sessions. However, at least L drinking sessions must be performed in G between every two successive initiations of the reset protocol. For a relatively small messages size of $C \log N$ we can have $L = N^C$, for some $C > 2$ (e.g., $C = 10$). This implies a rare occurrence of the reset protocol, and negligible contribution of its messages to the message complexity per drinking session.

16.5 Concluding remarks

It is straightforward to cast the above solution into a distributed system that offers `DrinkersSrv(G)` using fifo channels. The details are left as an exercise.

The solution given here can be easily modified to handle the extension of the drinkers service in which each edge in G is associated with multiple instances of multiple types of bottles [4].

Bibliography

- [1] K.M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM TOPLAS*, 1984.
- [2] E.W. Dijkstra. Two Starvation-free Solutions of a General Exclusion Problem. Technical report, Plataanstraat 5, 5671, AL Nuenen, The Netherlands, 1978. EWD-625.
- [3] D. Ginat. A Simple Near-Optimal Solution to the Drinking Philosophers Problem. Technical report, University of Maryland, Computer Science Department, November 1989. CSTR-2122.1.
- [4] D. Ginat and A.U. Shankar. Adaptive Ordering of Contending Processes in Distributed Systems. Technical report, University of Maryland, Computer Science Department, October 1989. PhD Thesis, CSTR-2335.
- [5] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 1978.
- [6] D. Lehman and M.O. Rabin. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution of the Dining Philosophers Problem. In *Proceedings of the 8th ACM POPL*, 1981.
- [7] G. Ricart and A.K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 1981.