Multi-resource Allocation with Unknown Participants

Ajoy K. Datta, Lawrence L. Larmore
School of Computer Science
University of Nevada Las Vegas
Las Vegas, USA
Firstname.Lastname@unlv.edu

Stéphane Devismes, François Kawala
VERIMAG
Université Joseph Fourier
Grenoble, France
Firstname.Lastname@imag.fr

Maria Potop-Butucaru
LIP6
Université Pierre et Marie Curie
Paris, France
Maria.Potop-Butucaru@lip6.fr

***Abstract*—We define the problem of multi-resource allocation, which is an extension of the dining philosophers problem. We apply this problem to systems where participants (here called clients) are unknown. We propose a solution for 2-resource allocation in static networks, then, explain how to modify our protocol to handle client dynamicity. Extend our solution to handle larger resource requests is let as a future work.**

I. Introduction

Research in distributed resource sharing problems (both in static and dynamic networks) has been active for more than three decades. Variants of the problem include *mutual exclusion, group mutual exclusion, k-exclusion, k-out-of-m exclusion, local mutual exclusion, dining philosophers, drinking philosophers*. (See the book of Nancy Lynch [1].)

All the above problems assume knowledge of one or more of the following parameters: the number of processes, the number of resources, the layout of the resources, and the degree of synchronization. In the recently emerging study of dynamic large scale networks (e.g. P2P, ad-hoc, sensor or robot networks) knowledge of these parameters can hardly be computed. Generally, in these systems, processes have only a partial view of the network.

In this paper, we consider asynchronous message-passing systems where a large number of participants (or clients) want to simultaneously access several resources in mutual exclusion. Due to the arbitrary large number of participants, we focus on the design of a resource allocation protocol in which participants are unknown. That is, each participant only knows its own identifier and that of the resources it needs. The main challenge is then to prevent deadlocks, as participants do not know each other, and may have conflicting requests.

In the following, we refer to the aforementioned problem as the *multiple-resource allocation problem* or the *k-resource allocation problem*. In this problem, there are $m \geq k$ resources in the system, clients know neither all resource identifiers nor m, and clients can request up to k resources simultaneously. Actually, clients only know the identifiers of the resources they need.

This problem is closely related to k-out-of-m exclusion [2], [3]. In k-out-of-m exclusion, there are m resource units of the same type and each process (client) can request up to k units. Conversely, in the k-resource allocation problem, resources may be of different types and the clients only know the identifiers of the resources they need. Moreover, resources are passive in k-out-of-m exclusion, while here, resources are active in a sense that they cooperate to resolve conflicts.

The *k-resource allocation* problem can be also compare to the *drinking philosophers problem* [4]. In this latter, on a finite undirected graph G we have two types of processes: Philosophers on G's vertex, and Bottles on G's edges. Initially every philosopher is *tranquil*. A *tranquil* philosopher may becomes *thirsty*, then he tries to acquire the (none empty) set of bottles he needs to becomes *drinking*. Later, in a finite time he stops drinking, and again becomes *tranquil*. In the *k-resource allocation problem*, clients have the same behavior. Likewise, the bottles correspond to the resources. But while in the *drinking philosophers problem*, every philosopher know his neighbors (*i.e.* on G) and have to exchange bottles with them (to satisfy every *thirsty* philosopher), in the k-resource allocation problem, clients do not know each-other, and resources are not mobile. Moreover, resources (bottles) are passive in the *drinking philosophers problem*, while here, resources are active in a sense that they cooperate to resolve conflicts.

The rest of the paper is organized as follows: In Section II, we formally define the model used throughout the paper. In Section III, we present a two-resource allocation algorithm for systems with unknown participants. In Section IV, we explain how to handle dynamicity of clients. Section V is dedicated to future works.

For space considerations, proofs have been omitted.

II. Model

A. The Problem

We consider an asynchronous message-passing system in which processes are divided into two classes named *clients* and *resources*, respectively. Each client needs to access some resource in order to execute a portion of its code called its *critical section*. Each access to a resource (that is, the critical section) is done in *finite* yet *unbounded* time and must satisfy *mutual exclusion*, *i.e.*, each resource can be used by at most one client at a time. However, a client may simultaneously access to up to k resources. Hence, we can specify our problem as follows:

- **Safety:** Each resource is used by at most one client at a time.
- **Liveness:** Each request of at most k different resources is eventually satisfied.

We refer to this problem as the *k-resource allocation problem*. In this paper, we restrict our study to the case where $k = 2$, that is, the *two-resource allocation problem*.

B. Processes

We denote by C the set of clients and by R the set of resources. We assume that C and R are disjoint and contain finitely many processes. (In most applications, there are far fewer resources than clients.) C contains n clients: $\{c_1, \ldots, c_n\}$, and R contains m resources: $\{r_1, \ldots, r_m\}$. However, n and m are unknown to the processes. By a slight abuse of notation, we will identify a process and its identifier. Identifiers are ordered, allowing us to compare two processes.

Each process executes a local algorithm and has a finite sized local memory. Communications between processes are made by passing messages through asynchronous communication links. Every process p_1 is able to send messages to any other process p_2 through a link, provided that p_1 is aware of the p_2's identifier. Initially, each client only knows its own identifier. Clients learn the identifiers of the resources they need thanks to a *resource discover oracle* [5], [6]. Then, they can communicate their identifiers to the targeted resources, and so on.

Conversely, resources are organized along a *rooted ring*. "Rooted" means that there is a unique distinguished resource called *root*.[1] Each resource r_i knows whether it is the root of the ring, thanks to the Boolean function $\mathrm{Root}(r_i)$. Each resource r_i knows its successor in the ring thanks to the function $\mathrm{Next}(r_i)$.

A client can access the output of the *resource discover oracle* using the unblocking function *getRequest*. In absence of any request *getRequest* returns $\{\perp, \perp\}$. If the application layer requests the use of one resource r_i, *getRequest* returns $\{r_i, \perp\}$. If the application requests the use of two resource r_i and r_j, *getRequest* returns $\{r_i, r_j\}$.

C. Communication Links

All the links are *reliable*, *i.e.*, a message sent by e through the link to f is delivered by f within finite time. Links also satisfy *integrity*, each message sent is delivered exactly once, and no unsent message is delivered. However, links are asynchronous, *i.e.*, there is no bound on the delay to deliver a message. Finally, there is *no assumption* on the delivery order: if a message m' is sent after a message m using the same link, then m may arrive before or after m'.

Note that the assumption on link reliability is not that strong in our setting as any unreliable link can be made reliable using a repetition mechanism similar to the alternating bit protocol [7].

[1] N.b., the only use of the root will be to initiate a perpetual token circulation in the ring.

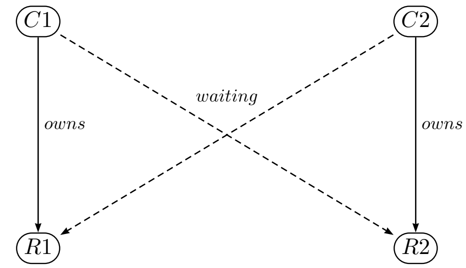

Fig. 1. Deadlock involving two clients

Each process can receive a message using the non-blocking function $receive(F, \{var_1, \ldots, var_n\})$. If there is no F-type message available in the process's reception buffer, the function returns *false*. Otherwise, the data of the message of type F is allocated in variables $\{var_1, \ldots, var_n\}$, after that the *receive* function will return *true* and pops the message from the reception buffer.

III. The Algorithm

Variable 1 for each client c.

Declarations :
  Array of ResourceID : D
  ClientID : c
Initialization :
  $D \leftarrow \{\perp, \perp\}$

Algorithm 1 The main loop of each **client :** c.

```
while True do

    if D = {⊥, ⊥} then
        D ← RequestProvider.getRequests()
        if D[1] ≠ ⊥ then
            if D[2] ≠ ⊥ then
                send⟨NewRequest, c, 2, D[1]⟩ to D[2]
            else
                send⟨NewRequest, c, 1, ⊥⟩ to D[1]
            end if
        end if
    end if
    if receive⟨resAllowed⟩ from r then
        ⟨CriticalSection⟩
        if D[2] ≠ ⊥ then
            send⟨Done⟩ to D[2]
        else
            send⟨Done⟩ to D[1]
        end if
    end if

    if receive⟨endACK⟩ from r then
        D ← {⊥, ⊥}
    end if
end while
```

Our algorithm is split into two parts: an algorithm for clients (Variable 1 and Algorithm 1) and another for resources (Variable 2, Algorithms 2, and 3). Below, we give an informal description of our solution.

A. Overview

We first present the basic principles used in our algorithm.

Variable 2 for each resource r.

Declarations :

Array of REQUESTQUEUEELEMENT : Q_{strong}, Q_{weak}, Q_{token}
REQUESTQUEUEELEMENT : $Request$
RESOURCEID : r_{min}
BOOLEANS : $Lock$, $NewHead$, $StrongReady$, $WeakReady$, $HoldingToken$, $SendingToken$

Initialization :

$Lock \leftarrow false$
$NewHead \leftarrow false$
$StrongReady \leftarrow false$
$WeakReady \leftarrow false$
$HoldingToken \leftarrow \mathrm{Root}(r)$
$Q_{Strong} \leftarrow \emptyset$
$Q_{Weak} \leftarrow \emptyset$
$Q_{token} \leftarrow \emptyset$
$r_{min} \leftarrow r$

1) Queues: In any solution to the *two-resource allocation problem*, a resource can only be allocated to one client at a time. During the time an client uses the resource, other clients may request the resource. Therefore, requests must be stored until they are satisfied. Unsatisfied requests are stored in *queues* located in targeted resources.

2) Deadlocks: When clients request several resources, deadlocks may occur. To see this, consider the following example: (1) Client $C1$ and Client $C2$ simultaneously request both Resources $R1$ and $R2$. Assume then that (2) $C1$'s request arrives to $R1$ before $C2$'s request and (3) $C2$'s request arrives to $R2$ before $C1$'s request. Assume that the requests of of $C1$ and $C2$ respectively for $R1$ and $R2$ eventually reaches the top of the associated queues. In such a situation $R1$ will not be released by $C1$ before it obtains $R2$. Reciprocally, $R2$ will not be released by $C2$ before it obtains $R1$. Hence, we obtain a deadlock configuration similar to the one presented in Figure 1. This situation can be generalized to k client/resource pairs.

3) Two different types of resources: Assume that a client requests two resources. In our algorithm, we label the first requested resource as *strong* and the second one as *weak*. According to its status (strong or weak), a request is stored at the targeted resource in its *strong* queue or its *weak* queue, respectively. If a client only requests one resource, the resource is labeled strong. A resource requested by a client is allocated to that client only when it is at the top of the resource's strong queue. In particular, when a client requests two different resources r_1 and r_2, it can access those resources only when the associated requests are *both* at the tops of the strong queues of r_1 and r_2. Thus, in order to satisfy a two-resource request, The weak request must eventually move from the weak queue to the strong queue of the resource. We use the rule that the weak request of client c moves from the weak queue to the strong queue, for that resource, only when the strong request of c is at the top of the strong queue of the requested resource.

B. Detailed Algorithm

We now give details of our algorithm.

1) Resource discover oracle: Initially, a client obtains from a *resource discover oracle* a request of one or two resources. Identifiers of the requested resources are stored in a two-cell array named D. When there is no request, the array D is equal to $(\perp, \perp)$. While $D[1] = \perp$, the client periodically calls the *resources discover oracle*. After receiving a request, $D[1]$ contains the strong resource identifier and $D[2]$ contains the weak resource identifier, if any. Next, according to the number of requested resources, we consider the two following cases.

2) One-Resource Requests: Assume that an application requests the use of only one resource. The corresponding client sends the request to that resource using a message *NewRequest*. The resource stores the request in its strong queue. Later, when that request reaches the top of the queue, the resource notifies the client (using message *resAllowed*) that it is allowed to execute its critical section. The client executes that section, then notifies the resource, using the message *Done*, that it has terminated its critical section. Consequently, the resource pops the client's request from the strong queue and finally informs the client using message *endACK* that it can propose a new request to the system.

3) Two-Resource Requests: Assume that an application of some client c requires the use of two resources, say r_1 and r_2. Then, c proceeds as illustrated in Figure 2. That is, c first sends a request to the weak resource r_2 using message *NewRequest*. The identifier of r_1 and the request type are also piggybacked onto the message. Upon receiving the message, the weak resource stores the following information about this request in its weak queue: the identifier of r_1, the identifier of c, and the request type (here 2, meaning *weak*).

After that, the weak resource forwards a message *NewRequest* to the strong resource r_1. In this message, the following information is attached: the identifiers of r_2 and c, a variable *RequestType* set to *strong* (value 1). When r_1 receives this message, the following information is stored in the strong queue of r_1: the identifier of the requester c, the requested type (here 1, meaning *strong*).

Eventually the request of c reaches the top of the strong queue of r_1. Then, r_1 notifies this r_2, by sending the message *Res1Ready*). Upon receiving the message, r_2 moves c's request from its weak queue to its strong queue.

When c is at the head of the strong queue of r_2, r_2 notifies c by sending the message *resAllowed* that it can use both r_1 and $r2$. c performs its critical section. Once the critical section is done, c asks r_1 and r_2 to remove its requests from the queues. First, it sends the message *Done* to r_2. Then, r_2 removes c's request from its *strong* queue and sends the message *Done* to r_1, causing r_1 removes c's request as well, and then send the message *endACK* to c.

C. Deadlock Resolution

Using the aforementioned mechanism, it is (still) possible to create deadlocks. Figure 3 gives an example of such a deadlock. We remark that a client can be involved in a deadlock only after its weak resource request moves from the weak queue to the strong queue of one of the targeted resource r_i. To solve this problem, we require that r_i start a deadlock detection process at that time. More precisely, assume that client c requests r_1 as a strong resource, and r_2 as a weak resource. When c's strong request has reached the

Algorithm 2 The main loop of each **Resource :** r. Commons features.

```
while True do
    if receive⟨Token⟩ from r' then
        for each Request in Q_strong do
            if Request.RqNo = 1 then
                Q_token ← Q_token ⊕ Request
            end if
        end for
        HoldingToken ← True
    end if
    if SendingToken then
        HoldingToken ← False
        SendingToken ← False
        send⟨Token⟩ to RingNextResId
    end if
    if ¬Lock then
        if receive⟨NewRequest, c,
                        RequestNo, NextResId⟩ from r' then
            if RequestNo = 1 then
                if Head(Q_Strong) = ⊥ then
                    NewHead ← True
                end if
                Q_Strong ← Q_Strong ⊕
                        ⟨c, RequestNo, NextResId, 1⟩
            else
                Q_Weak ← Q_Weak ⊕
                        ⟨c, RequestNo, NextResId, 0⟩
                send⟨NewRequest, c, 1, r⟩ to NextResId
            end if
        end if
    end if
    if NewHead then
        if Head(Q_Strong).RqNo = 1 then
            if Head(Q_Strong).NextResId= ⊥ then
                send⟨resAllowed⟩ to Head(Q_Strong).ClId
            else
                send⟨NewStrong, Head(Q_Strong).ClId⟩ to
                        Head(Q_Strong).NextResId
            end if
        else
            WeakReady ← true
        end if
        NewHead ← false
    end if
    if receive⟨NewStrong, c⟩ from r' then
        if Q_Strong = ∅ then
            NewHead ← True
        end if
        Q_Strong ← Q_Strong ⊕ search(Q_Weak, c)
        Q_Weak ← Q_Weak ⊖ search(Q_Weak, c)
        send⟨newStrongACK⟩ to r'
    end if
    if receive⟨StrongReady⟩ from r' then
        StrongReady ← true
    end if
    if StrongReady and WeakReady then
        send⟨resAllowed⟩ to Head(Q_Strong).ClId
        StrongReady ← false
        WeakReady ← false
    end if
    if receive⟨Done⟩ from r' then
        if Head(Q_Strong).RqNo = 1 then
            c ← Head(Q_Strong).ClId
            Q_Strong ← Q_Strong.pop
            if HoldingToken then
                Q_token ← Q_token.pop
            end if
            send⟨EndACK⟩ to c
        else
            Q_Strong ← Q_Strong.pop
            send⟨Done⟩ to Head(Q_Strong).NextResId
        end if
        NewHead ← (Q_Strong ≠ ∅)
        SendingToken ← (Q_token = ∅)
    end if
```

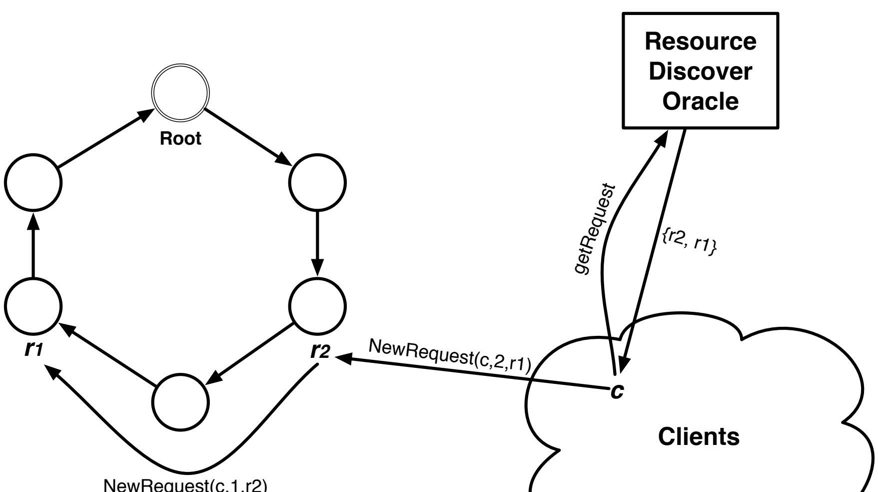

Fig. 2. Two-resource request

top of r_1's strong queue, r_1 notifies r_2 using the message *NewPriority*. Consequently, r_2 moves c's weak request from

Algorithm 3 The main loop of each **Resource :** r. Features related to the loop breaking.

```
    if receive⟨newStrongACK⟩ from r' then
        if HoldingToken then
            r_min ← ∞
        end if
        send⟨seekLoop, r_min, r⟩ to Head(Q_Strong).NextResId
    end if

    if receive⟨seekLoop, r_min, r_init⟩ from r' then
        if r = r_init then
            if r ≠ r_min then
                send⟨RemoteKillLoop⟩ to r_min
            else
                Lock ← True
                send⟨KillLoop, Head(Q_Strong).ClId⟩ to Head(Q_Strong).NextResId
            end if
        else
            if (Q_Strong) = ∅ or Head(Q_Strong).NextResId= ⊥ or Head(Q_Strong).RqNo = 2 then
                send⟨NoLoop⟩ to r_init
            else
                if ¬HoldingToken and then r_min > r then
                    r_min ← r
                end if
                send⟨seekLoop, r_min, r_init⟩ to Head(Q_Strong).NextResId
            end if
        end if
    end if

    if receive⟨NoLoop⟩ from r' then
        send⟨Res1Ready⟩ to Head(Q_Strong).NextResId
    end if

    if receive⟨RemoteKillLoop⟩ from r' then
        Lock ← True
        send⟨KillLoop, Head(Q_Strong).ClId⟩ to Head(Q_Strong).NextResId
    end if

    if receive⟨KillLoop, c⟩ from r' then
        Lock ← True
        Q_Weak ← Q_Weak ⊕ search(Q_Strong, c)
        Q_Strong ← Q_Strong ⊖ search(Q_Strong, c)
        send⟨KillACK1⟩ to r'
    end if

    if receive⟨KillLoopACK1⟩ from r' then
        Head(Q_Strong).Score←Head(Q_Strong).Score + 1
        HeadToTail(Q_Strong)
        send⟨KillACK2⟩ to r'
        NewHead ← true ; Lock ← false
    end if

    if receive⟨KillLoopACK2⟩ from r' then
        Lock ← false
    end if

end while
```

its weak queue to its strong queue and then sends the message *newPriorityACK* to r_1. Upon receipt of that message, r_1 starts a *deadlock detection process*.

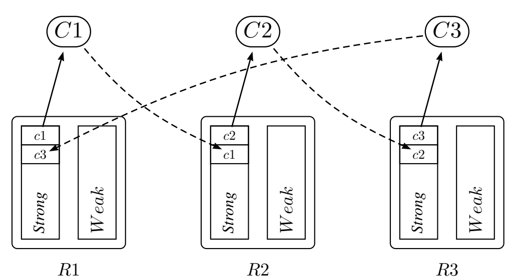

Fig. 3. Deadlock involving three clients and resources.

1) Deadlock detection: A deadlock occurs when dependencies between resources form a cycle. For example, the deadlock described on Figure 3 produces the logical cycle described on Figure 4. The dependencies can be defined as follows. Let $\mathit{Strong}(r)$ denote the strong queue of resource r and $\mathit{Head}(Q)$ denotes the head of queue Q. Let $c \in C$ be a client, rq_c^{strong} its strong request and rq_c^{weak} its weak request, then:

Definition 3.1:

$$Dependency\ r_i \mapsto r_j \Leftrightarrow \begin{cases} \mathit{Head}(\mathit{Strong}(r_i)) = rq_c^{\mathit{strong}} \\ \textbf{and} \\ rq_c^{\mathit{weak}} \in \mathit{Strong}(r_j) \end{cases} \tag{1}$$

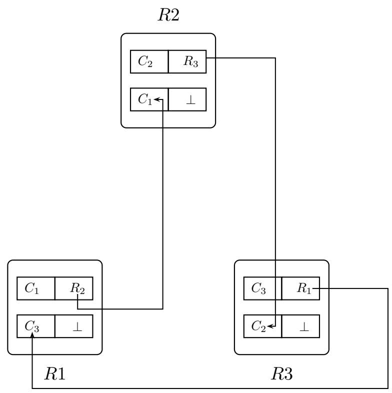

Fig. 4. Logical view of a dependency loop, involving three clients.

According to this definition, a *dependencies cycle* is defined as a sequence of resources $r_0, \ldots, r_k$ such as $\forall i < k, r_{i-1} \mapsto r_i$ and $r_k \mapsto r_0$.

Hence to detect a cycle, the deadlock detection started by the resource r_i consists of sending a message *SeekLoop* to the resource r_j satisfying $r_i \mapsto r_j$. The message is routed along the dependencies and if it comes back to r_i, then there exists a cycle and consequently there is a deadlock to break. Otherwise, the message reaches a resource r_k with no dependency, then there does not exist any cycle, and r_k sends a *NoLoop* message to the *SeekLoop* message's initiator (here r_1) in order to inform it of that fact.

If the cycle detection process finds a cycle, then the initiator of the detection is eventually aware of this fact. Consequently it starts the *unloop process*. Otherwise, the initiator is eventually informed that it is not involved into a cycle by message *NoLoop*. The reception of this message is now required before a resource sends the message *resAllowed*.

2) The Unloop Process: Assume that there is a cycle. This cycle is eventually detected thanks to the deadlock detection process presented above. Then, an *unloop process* is started. This process consists of removing a dependency to break the cycle. Assume that the unloop process decides to break the dependency $r_i \mapsto r_j$, which is due to client c. To break $r_i \mapsto r_j$, the unloop process will reorganize the queues of r_i and r_j. A way to break this dependency is to:

- move c's weak request to r_j's weak queue, and
- move c's strong request to r_i's strong queue's tail.

With this reorganization the algorithm guarantees that the dependency between r_i and r_j is broken. For sake of coherency, this reorganization is made atomically: during the reorganization of a resource's queue, no client's request can be added to it. Otherwise, new cycles could be created between the newly added requests and r_i or r_j.

The atomic reorganization occurs as follows: r_i stops to add elements in its strong queue, and sends r_j the message *killLoop*, containing the identifier of the involved client, (*i.e.* here c). Upon reception of *killLoop*, r_j also stops adding requests to its strong queue, after which r_j moves c's request from its strong queue to its weak queue. Finally r_j sends an acknowledgment to r_i using the $killLoopACK1$. When r_i receives $killLoopACK1$, it moves c's request from the top to the tail of its *strong* queue. Finally r_i sends an acknowledgment to r_j using a $killLoopACK2$ message, and again authorizes adding clients' requests to its queue. Upon the reception of $killLoopACK2$, r_j also authorizes adding new requests to its queue.

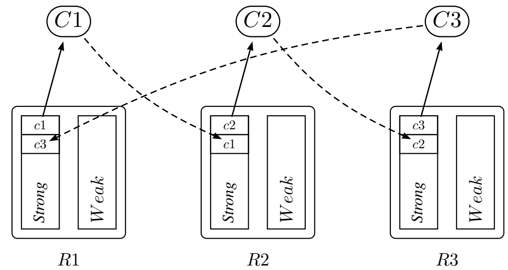

Fig. 5. A deadlock situation.

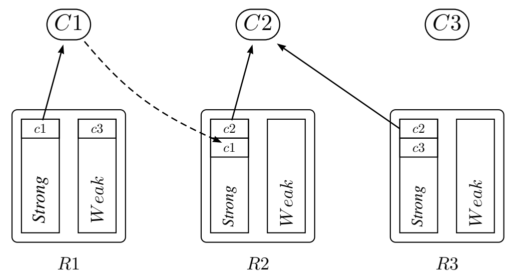

Fig. 6. Result of a reorganization.

An example of reorganization is given in Figures 5 and 6, this reorganization concerns the client c_3's requests.

3) Reorganization: Once a resource r_i has discovered a dependency loop, it has to choose which dependency the unloop process must break. A possible choice is to break the dependency $r_k \mapsto r_{k+1}$ where r_k is the resource in the cycle with the lower identifier. However, this solution is not fair. Indeed, it is possible that r_k is involved in cycles infinitely many times, and that each time the request of some client c is at the top of the strong queue of r_k, the queue is reorganized. As a consequence, the request of c is never satisfied.

To avoid this problem, we define priorities according to a token that circulates in the rooted ring defined among the resources. The priorities are given by the order relation $\prec$ defined below. In this definition, $\mathrm{Token}(r_i)$ is true when r_i holds the token, and false otherwise. The resource chosen to break a cycle is the minimum one according to $\prec$.

Definition 3.2:

$$r_a \prec r_b \Leftrightarrow \begin{cases} Token(r_b) \textbf{ and } \neg Token(r_a) \\ \textbf{or} \\ Token(r_b) = Token(r_a) \textbf{ and } r_a < r_b \end{cases} \quad (2)$$

The resource that holds the token is always maximal according to $\prec$. Hence to guarantee fairness, we must ensure that if a request is in a resource for a long time, then the request must eventually have the highest priority thanks to the token. To do that, we manage the token circulation as follows. A token is created by the root resource at its initialization. Then, when a resource r_i receives the token, r_i stores a copy of all its current strong requests in a specific queue Q_{token}. Then, each time a request is satisfied, if a copy exists in Q_{token}, it is removed. When this queue is empty (*i.e.* all requests in it have been satisfied), r_i releases the token and sends it to *Next*(r_i). Thank to the token, each resource periodically flushes its "old" strong requests, ensuring then the fairness of the algorithm.

Finally, note that the algorithm must store information in the *SeekLoop* message during the deadlock detection process to know which resource is minimum according to $\prec$. For each resource r_i, the priority level is ∞ if r_i holds the token, its identifier otherwise. When a resource initiates a new *SeekLoop* message, it stores its priority level in the *SeekLoop* message. Then, each time the message is received by some other resource, the minimum encountered priority level stored in the message is updated. Thereby a resource r_i which receives it own *SeekLoop* message, in addition to knowing the existence of a cycle, will also know the target on which initiate the unloop process. [2] If the target is not the *SeekLoop* initiator r_i, then r_i sends the message $RemoteKillLoop$ to the targeted resource. Upon receiving this message, the resource executes the unloop process.

IV. Extensions

To be used in peer-to-peer systems, we need to modify our algorithm to handle client arrivals or departures. Client arrivals are not a problem providing that each new client arrives with an identifier that was never used before. If we assume that each time a client leaves the system, it sends a message announcing its departure, then client dynamicity can be easily handled. Finally, if a client can leave the system without informing any other process, we need an additional mechanism to handle such dynamicity. *Participant detector* [8], [9] is such a mechanism. Basically, a participant detector is a function that gives information to a process about the presence of some other processes in the system. In our context, we need resources to watch the clients currently in their queues. If a client leaves the system, any resource having the client in its queue must be eventually notified, and will then remove that client's request from its queue. Moreover, all information given by the participant detector must be accurate: a resource must not remove the request of any client still in the system. Hence, we need a *perfect* participant detector, that is, a detector that satisfies: (*strong completeness*) every client that leaves the system is eventually removed from the participant lists and (*strong accuracy*) no client can be removed from a list of participants before it leaves the system. Essentially, such a participant detector is a straightforward adaptation of the work in [10] to our problem. We leave the implementation as an exercise for the reader.

[2] Note that if there is a cycle, then the minimum priority level is different from ∞ because there is only one token.

V. Perspectives

The immediate perspective of this work is to find a k-resource allocation protocol that works for all value of k. However, the generalization of your current solution seems to be difficult. Indeed, the larger k is, the harder the deadlock detection and resolution is. So, instead of an optimistic solution where queues evolve independently until a deadlock occurs and is treated, we propose to consider a pessimistic solution, that is, a solution where we prevent deadlock creation. For example, when a client c requests the use of several resources, a request is stored in the weak queues of each resource requested by c. Then, we use a token circulation to atomically move the requests of a client from the weak queues to the strong queues. Such a solution works for all value of k, but allow far less concurrency than our solution for $k = 2$.

Acknowledgment

This work has been partially supported by the ANR project *ARESA*2.

References

[1] N. Lynch, *Distributed algorithms*. Morgan Kaufmann, 1996.

[2] M. Raynal, "A distributed solution to the k-out of-m resources allocation problem." in *ICCI*, 1991, pp. 599–609.

[3] A. K. Datta, S. Devismes, F. Horn, and L. L. Larmore, "Self-stabilizing k-out-of-; exclusion in tree networks," *Int. J. Found. Comput. Sci.*, vol. 22, no. 3, pp. 657–677, 2011.

[4] K. M. Chandy and J. Misra, "The drinking philosopher's problem," *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 4, pp. 632–646, 1984.

[5] I. Abraham and D. Dolev, "Asynchronous resource discovery," *Computer Networks*, vol. 50, no. 10, pp. 1616–1629, 2006.

[6] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi, "Peer-to-peer resource discovery in grids: Models and systems," *Future Generation Comp. Syst.*, vol. 23, no. 7, pp. 864–878, 2007.

[7] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links," *Commun. ACM*, vol. 12, no. 5, pp. 260–261, 1969.

[8] F. Greve and S. Tixeuil, "Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks," in *Proceedings of IEEE International Conference on Dependable Systems and networks (DSN 2007)*. IEEE, June 2007, pp. 82–91.

[9] E. A. P. Alchieri, A. N. Bessani, J. da Silva Fraga, and F. Greve, "Byzantine consensus with unknown participants," in *OPODIS*, 2008, pp. 22–40.

[10] C. Fetzer, "Perfect failure detection in timed asynchronous systems," *IEEE Trans. Computers*, vol. 52, no. 2, pp. 99–112, 2003.