On The Impossibility Of Robust Solutions For Fair Resource Allocation

Rajeev Joshi , Jayadev Misra

Department of Computer Sciences, The University of Texas at Austin

Abstract

We show that the presence of even a single faulty process makes it impossible to design a strategy for fair allocation of a shared resource.

Key words: Concurrency; Distributed Computing; Fault tolerance

0 Introduction

A classic problem in distributed systems is the following resource allocation problem: given a set of processes sharing a resource, design a synchronisation mechanism that guarantees (i) *mutual exclusion*, i.e., at any moment, at most one process uses the resource, and (ii) *deadlock-freedom*, i.e., if some process is waiting for the resource, then eventually some process uses the resource. A simple centralised solution to this problem is the following: a boolean variable b is used to denote whether the resource is available; b is initially set to *true*. An attempt by a process to acquire the resource is successful only if b is *true*; as a result, b is set to *false*. The attempt is unsuccessful if b is *false*. When a process releases the resource after use, b is set to *true*.

A variant of this problem is the *fair* resource allocation problem, in which requirement (ii) above is replaced by the stronger condition of *starvation-freedom*, i.e., given that no process uses the resource forever, no process is denied forever. Although the above implementation is not starvation-free, it can be modified to meet this requirement by introducing a new variable q , of type *Sequence of Process Identifier* ; q is initially set to the empty sequence. An attempt by a process to acquire the resource is successful only if b is *true* and the identifier for the process is at the head of q ; as a result, the identifier is removed from q , and b is set to *false*. The attempt is unsuccessful otherwise, and the process identifier is appended to q if it is not already in q . As before, when a process releases the resource after use, b is set to *true*.

A drawback of this implementation is that it works correctly only when all processes are *persistent*, i.e., each process waiting to use the resource makes repeated attempts to acquire it until successful. In the presence of even a single *transient* process, i.e., a process not guaranteed to follow each unsuccessful attempt with another request, the above design provides not even the guarantee of deadlock-freedom, because the presence of an identifier for a transient process at the head of q can block the progress of other processes.

In this paper, we show that this difficulty is not just an artefact of the simplicity of this particular implementation – that, in fact, there is *no* solution to the problem of fair resource allocation in an asynchronous system in which some processes might be transient.

Our notion of transient processes corresponds to a class of process failures in asynchronous systems for which no finite experiment can distinguish between failure and slow execution. Note, however, that transient processes may fail only when they are not using the resource; in particular, we do not consider situations in which processes may fail while using the resource, nor situations in which the resource manager may fail: for systems with such catastrophic failures, it is easy to show that there is no implementation that provides any progress guarantees. We have made stronger assumptions about the behaviour of processes in order to obtain a nontrivial impossibility result.

Moreover, we have considered only those situations in which unsuccessful processes make repeated attempts to acquire the resource. In some implementations, an unsuccessful attempt causes a process to *block*, and wait for a signal from the resource manager. For such systems, the analogue of a transient process is one that may fail while it is blocked. Assuming, as before, that such failures are indistinguishable from slow response by finite experiments, our result applies in these situations too.

1 Preliminaries

Consider a system of N processes, numbered 0 through $N-1$ (with $N>1$), that share a resource. We represent a history of operations on the resource by a string of symbols that are of the form S_j , U_j or R_j , for $0 \leq j < N$. Each symbol denotes an operation on the resource: S_j denotes a successful attempt by process j to acquire the resource, U_j denotes an unsuccessful attempt if process j is not holding the resource, and a no-op otherwise, and R_j denotes a release if process j is holding the resource, and a no-op otherwise. (We use this convention of no-ops because it places fewer constraints on occurrences of symbols of the form U_j and R_j ; this leads to a simpler presentation.)

An implementation is given by a nonempty set of such strings, which correspond to the execution histories that the implementation admits. Not all such nonempty sets are implementations, however – implementations are required to satisfy certain additional properties, which are described in section 2.

Notational Remark. Throughout this paper, the identifiers e, f range over symbols, x, y range over finite strings, r ranges over finite and infinite strings, and i, j range over processes, i.e., $0 \leq i < N$ and $0 \leq j < N$.

For any x, r , we write $x \sqsubseteq r$ (read "x under r") to mean that x is a finite prefix of r . The empty string is denoted by ε ; it satisfies, for any string r , $(\varepsilon \sqsubseteq r)$. For any e, x , the expression $x \triangleleft e$ (read "x snoc e") denotes the string consisting of x followed by e . The operator $\triangleleft$ is left associative, i.e., $x \triangleleft e \triangleleft f$ means $(x \triangleleft e) \triangleleft f$.

Operators have the following precedences:

$$\{.\} \ , \ \{ \triangleleft , \neg \} \ , \ \{ \sqsubseteq , \in , \notin , = \} \ , \ \{ \ \wedge \ , \ \vee \ \} \ , \ \{ \ \Rightarrow \ , \ \equiv \ \}$$

where symbols within a set have the same binding power, and greater binding power than symbols in sets to the right. The infix operator '.' denotes function application.

(End of Notational Remark.)

2 Implementations

Due to the fact that implementations provide certain guarantees, e.g., mutual exclusion, a set of strings constitutes an implementation only if it satisfies certain properties. We express these properties in terms of two relations, defined, for all j, x as: (recall that x ranges over finite strings only)

(***Loose***) $(j \ \textit{loose in} \ x) \ \equiv \ (\text{every } S_j \text{ in } x \text{ is followed eventually by an } R_j)$

(***Free***) $(x \ \textit{free}) \ \equiv \ (\forall\, j :: j \ \textit{loose in} \ x)$

Informally speaking, $(j \ \textit{loose in} \ x)$ states that process j does not hold the resource after execution x , whereas $(x \ \textit{free})$ states that the resource is available after execution x .

An implementation M is a nonempty set of strings, called the *runs* of M , that satisfies the five conditions (***C0***)–(***C4***) given below. (In (***C0***)–(***C2***), variables j, x and r are quantified universally over the appropriate domains.)

(***C0***) $(x \triangleleft S_j \in M) \ \Rightarrow \ (x \ \textit{free})$

Informally speaking, (***C0***) expresses mutual exclusion, i.e., a process is successful only if the resource is available. A consequence of this condition is that strings of the form $x \triangleleft S_i \triangleleft S_j$ are not runs of any implementation.

(***C1***) $x \in M \ \wedge \ (j \ \textit{loose in} \ x) \ \Rightarrow \ (x \triangleleft S_j \in M) \ \vee \ (x \triangleleft U_j \in M)$

(***C2***) $x \in M \ \wedge \ \neg(j \ \textit{loose in} \ x) \ \Rightarrow \ (x \triangleleft R_j \in M)$

These conditions state that processes behave asynchronously: (***C1***) states that, at any point, a process not holding the resource might attempt to acquire it, and the implementation should be prepared to respond to the attempt by allowing it to be successful or unsuccessful. (***C2***) states that, at any point, a process holding the resource might release it. Note that these conditions are quite general; for instance, they admit a nondeterministic implementation in which, for some i, j, x , the strings $x \lhd S_i$, $x \lhd S_j$ and $x \lhd U_j$ are all runs of the implementation.

(***C3***) $\qquad$ (*M is prefix-closed*)

Condition (***C3***) states that every prefix of a run is also a run; from this, and the fact that implementations are nonempty, it follows that ε is a run of every implementation.

(***C4***) *Let r be any string such that*

$$(\textit{all finite prefixes of } r \textit{ are in } M) \ \wedge$$
$$(\forall\, j, x :: x \lhd U_j \sqsubseteq r \ \Rightarrow\ x \lhd S_j \notin M)$$

Then, $r \in M$ *.*

Condition (***C4***) holds trivially for finite r ; for infinite r , it asserts the following closure condition on M : if all finite prefixes of r are in M , and if unsuccessful attempts occur in r only at points at which successful attempts cannot occur, then r is in M .

Remark on (***C4***). Condition (***C4***) needs some explanation. In particular, we note that it is weaker than the requirement that M be continuous (which would state that any run, all of whose finite prefixes are in M , also be in M) . The reason we do not require continuity is that it disallows certain solutions with unbounded nondeterminism. For instance, consider a solution that initially selects an arbitrary natural number k , then rejects the first k attempts to acquire the resource, and then behaves like the starvation-free solution described in the introduction. This solution is not continuous, because an infinite string consisting only of unsuccessful attempts is not an execution history, although all its prefixes are; however, the solution satisfies conditions (***C0***)–(***C4***), so it is an implementation by the above definition.
(End of Remark.)

3 Fair Implementations

For any string r , we write (r *is fair*) to mean that, for all j ,

(***Fair***) $\qquad$ (r has infinitely many U_j) $\Rightarrow$ (r has infinitely many S_j)

Note that, by this definition, all finite strings are fair.

An implementation M is said to be fair provided that all runs in M are fair. Informally speaking, fair implementations provide the guarantee of starvation-freedom to persistent processes. (Recall from the introduction that such implementations solve the fair resource allocation problem.)

4 Nonexistence of Fair Implementations

We show that every implementation has an unfair run. The proof consists of showing how to construct such a run for a given implementation, by exploiting the fact that some process may be transient. As discussed in the introduction, this establishes the nonexistence of a robust implementation for the problem of fair resource allocation.

The main idea in the proof is to have two processes, A and B, collude in the following way: process A makes repeated attempts to acquire the resource until it is successful. If it acquires the resource, B makes an attempt, which is unsuccessful, because the implementation guarantees mutual exclusion. Next, A releases the resource, and the processes repeat this behaviour. There are two cases to consider: (i) A is always eventually successful, and (ii) after a certain point, A is always unsuccessful. In case (i), B starves for the resource; in case (ii), A starves – in both cases, the resulting run is unfair. (Note that, in case (ii), process B behaves in a transient manner, because it may not follow an unsuccessful attempt with another request.)

This informal argument is formalised in the following theorem.

Theorem. *Every implementation has an unfair run.*

Proof. Given an implementation M , we design an algorithm to construct an infinite sequence Y of finite strings such that the limit of Y is an unfair run of M . The algorithm consists of a nonterminating loop that maintains an invariant L , defined as:

$$L : \ (Y.h \ \mathit{free}) \ \wedge \ (Y.h \in M) \ \wedge \ (\forall\, j, x :: x \lhd U_j \sqsubseteq Y.h \ \Rightarrow \ x \lhd S_j \notin M)$$

where h is a program variable. Informally speaking, this invariant states that every string in the sequence constructed by the algorithm represents an execution history of M in which processes are unsuccessful only at points where they cannot be successful; we shall use this property in showing that the limit is an unfair run of M .

The complete algorithm is shown below. Note that the disjunction of the guards is *true*; thus, the loop is nonterminating, and the algorithm eventually assigns, for each k, a value to $Y.k$.

The two alternatives of the loop mirror the informal argument presented above. In the first alternative, the run being constructed is extended as follows:

```
|[ var  $h : \mathit{Natural}$
;  $h\,,\ Y.0 := 0\,,\ \varepsilon$        $\{L\}$
;  do
       $Y.h \triangleleft S_1 \in M$        $\longrightarrow$        $\{L \ \wedge\ (Y.h \triangleleft S_1 \in M)\}$
           $h\,,\ Y.(h+1) \ := \ h+1\,,\ (Y.h \triangleleft S_1 \triangleleft U_0 \triangleleft R_1)$      $\{L\}$
     [] $Y.h \triangleleft S_1 \notin M$        $\longrightarrow$        $\{L \ \wedge\ (Y.h \triangleleft S_1 \notin M)\}$
           $h\,,\ Y.(h+1) \ := \ h+1\,,\ (Y.h \triangleleft U_1)$      $\{L\}$
   od
]|
```

first, process 1 acquires the resource, then process 0 makes an attempt, which is unsuccessful (on account of (***C0***)), and then, process 1 releases the resource. In the second alternative, the run is extended with an unsuccessful attempt by process 1 .

L is initially established by setting h to 0 and $Y.0$ to ε . (Recall from section 2 that ε is a run of every implementation.) To establish the invariance of L , we show that each conjunct is preserved by the assignments in the alternatives.

For the first conjunct, viz., $(Y.h\ \mathit{free})$, this follows from (***Loose***), (***Free***) and the form of the two assignments. For the third conjunct, viz.,

$$(\forall\, j, x :: x \triangleleft U_j \sqsubseteq Y.h \ \Rightarrow\ x \triangleleft S_j \notin M)$$

we note that invariance in the first alternative follows from the fact that $Y.h \triangleleft S_1 \triangleleft S_0 \notin M$ (see comment after (***C0***)), and, in the second alternative, follows directly from the guard.

For the second conjunct, viz., $(Y.h \in M)$, we observe, for the first alternative:

$$\begin{array}{ll}
 & (Y.h \triangleleft S_1 \in M) \ \wedge\ (Y.h\ \mathit{free}) \\
\Rightarrow & \qquad \{\ (\textbf{\textit{Loose}}) \text{ and } (\textbf{\textit{Free}})\ \} \\
 & (Y.h \triangleleft S_1 \in M) \ \wedge\ (0\ \mathit{loose\ in}\ Y.h \triangleleft S_1) \\
\Rightarrow & \qquad \{\ (\textbf{\textit{C1}}), \text{ with } j, x := 0, Y.h \triangleleft S_1\ \} \\
 & (Y.h \triangleleft S_1 \triangleleft S_0 \in M) \ \vee\ (Y.h \triangleleft S_1 \triangleleft U_0 \in M) \\
\equiv & \qquad \{\ \text{first disjunct is } \mathit{false}\text{, from } (\textbf{\textit{C0}})\ \} \\
 & Y.h \triangleleft S_1 \triangleleft U_0 \in M \\
\Rightarrow & \qquad \{\ (\textbf{\textit{Loose}}) \text{ and } (\textbf{\textit{C2}}) \text{ with } j, x := 1, Y.h \triangleleft S_1 \triangleleft U_0\ \} \\
 & Y.h \triangleleft S_1 \triangleleft U_0 \triangleleft R_1 \in M
\end{array}$$

And for the second alternative:

$(Y.h \in M) \;\; \wedge \;\; (Y.h \textit{ free})$

$\Rightarrow \qquad \{ \; (1 \textit{ loose in } Y.h) \; , \; (\boldsymbol{C1}) \text{ with } j, x := 1, Y.h \; \}$

$(Y.h \triangleleft U_1 \in M) \;\; \vee \;\; (Y.h \triangleleft S_1 \in M)$

$\Rightarrow \qquad \{ \; (Y.h \triangleleft S_1 \notin M) \; , \text{ from the guard for the alternative } \}$

$Y.h \triangleleft U_1 \in M$

By construction, for all k , $Y.k \sqsubseteq Y.(k+1)$, thus the limit of Y is defined. Let r denote this limit. From the fact that L is invariant, we conclude, applying (***C4***), that r is a run of M .

We show that r is unfair as follows: if the first alternative is selected only finitely often, r has an infinite suffix in which U_1 occurs infinitely often, and S_1 does not occur; otherwise, the first alternative is selected infinitely often, U_0 occurs infinitely often in r , whereas S_0 does not occur. In either case, r does not satisfy (***Fair***).

(End of Proof of Theorem.)

5 Summary

Our result is similar in spirit to a result by Fischer, Lynch and Paterson (see [0]), which showed the impossibility of distributed consensus in the presence of even one failure. We have shown that even a single failed process makes it impossible to implement fair resource allocation.

Acknowledgements. We are grateful to the Seuss Group at the University of Texas at Austin for their comments. In particular, we thank Will Adams for pointing out a problem with an earlier proof.

References

[0] Michael J.Fischer, Nancy A. Lynch, Michael S. Paterson, *Impossibility of Distributed Consensus With One Faulty Process*. In: *Journal of the ACM*, Vol.32, No.2, pp.374-382, 1985.