The Disclosure Power of Shared Objects

Peva Blanchard, Rachid Guerraoui, Julien Stainer, and Igor Zablotchi(✉)

EPFL, Lausanne, Switzerland
{peva.blanchard,rachid.guerraoui,julien.stainer,igor.zablotchi}@epfl.ch

Abstract. Shared objects are the means by which processes gather and exchange information about the state of a distributed system. Objects that disclose more information about the system are therefore more desirable. In this paper, we propose the schedule reconstruction (SR) problem as a new metric for the disclosure power of shared memory objects. In schedule reconstruction, processes take steps which are interleaved to form a schedule; each process needs to be able to reconstruct the schedule up to its last step. We show that objects can be ranked in a hierarchy according to their ability to solve SR. In this hierarchy, stronger objects can implement weaker objects via a SR-based universal construction. We identify a connection between SR and consensus and prove that SR is at least as hard as consensus. Perhaps surprisingly, we show that objects that are powerful in solving consensus—such as compare-and-swap—are not always powerful in their ability to solve SR.

1 Introduction

Programming a computing system in a centralized way is significantly more powerful than doing so in a distributed way. The main difficulty of distributed programming comes from the lack of knowledge that a process has about the state of the other processes and the overall state of the system. The more information a process has about the state of the system, the easier it is to write an algorithm for that process to achieve a task in coordination with the other processes. In a distributed system, this information can only be obtained by processes from shared objects. So, intuitively, the more information an object discloses about the rest of the system, the more appealing it is.

In this paper, we propose the schedule reconstruction (SR) problem as a new metric for the disclosure power of shared objects. In order to solve SR, processes in a shared memory system need to be able to accurately identify the interleaving of steps (shared memory accesses) taken by all processes (the schedule). It is easy to see why objects that can identify the schedule are desirable. Knowing the schedule basically equates knowing the full system state and thus overcoming the main difficulty of distributed programming, as mentioned above.

We associate a SR number with each object A, representing the maximum number of processes of a system in which A can solve SR. Objects can thus be organized in a hierarchy, with each level corresponding to a different SR number.

This work has been supported in part by the European ERC Grant 339539 (AOC).

A. El Abbadi and B. Garbinato (Eds.): NETYS 2017, LNCS 10299, pp. 222–227, 2017.
DOI: 10.1007/978-3-319-59647-1_17

There is a natural connection between disclosure power as measured by the SR number and synchronization power as measured by the consensus number [2]. Intuitively, synchronization is a means of restricting the very large space of executions of a concurrent algorithm, whereas SR is a way of identifying which one of these possible executions actually occurred. At first glance, one would expect that objects with a high reconstruction power (high SR number) should also have a high synchronization power (high consensus number). We confirm this intuition by showing that SR is at least as hard as consensus: the SR number of an object is at most its consensus number.

Due to this connection between SR and consensus, intuition might also predict the inverse relationship to hold true: that objects with a high consensus number should also possess a high SR number. Surprisingly, this is not always the case, as we show in this paper. We prove that compare-and-swap, a very powerful, even universal [2], synchronization primitive, is no more powerful than simple read-write registers in terms of schedule reconstruction.

An object A's position in the SR hierarchy also determines A's power to implement other objects. We show that in the SR hierarchy a stronger object A is always able to implement a weaker object B, by providing a universal construction based on SR objects. We also show that B is unable to implement A in such a way that the implementation maintains the same disclosure power as A. In other words, implementing a stronger object from a weaker one always entails losing disclosure power.

2 Model and Problem Statement

2.1 Processes

We consider a set of n processes $P = \{P_1, ..., P_n\}$ that communicate through shared memory using a set of memory access primitives. The processes are executing an algorithm $\mathcal{A}$, which consists of a sequence of shared memory accesses and local steps. We assume local steps to be instantaneous and shared memory steps to be atomic.

An execution of algorithm $\mathcal{A}$ by a set of processes P is modeled by a *schedule*—a finite or infinite sequence of process identifiers which represents the interleaving of steps taken by the processes. When describing a schedule, we ignore local steps, so a schedule defines a global total order on the shared memory accesses done by all processes participating in the execution.

2.2 Schedule Reconstruction Object

A *schedule reconstruction object* (or *SR object*) provides two methods, `step` and `reconstruct`, neither of which takes any arguments. Basically, a call to `reconstruct` by a process p returns the schedule up to the last `step` call by p. The two methods need to satisfy the following conditions:

- the execution of each call to `step` performs exactly one primitive shared memory access and any number of local steps.
- `reconstruct` may only be implemented using local steps and shared memory accesses *that do not modify the state of shared memory* (such as reads).
- a call to `reconstruct` by process p returns the schedule as a mapping from step numbers to process ids or an empty mapping if there are no `step` calls by p preceding the `reconstruct` call.

We are interested in wait-free implementations of SR objects that correctly reconstruct *any possible schedule* (any interleaving of `step` calls). We call any such implementation a *SR algorithm*. A class C of objects solves n-process schedule reconstruction if there exists an SR algorithm $\mathcal{A}$ that solves n-process schedule reconstruction using any number of objects of class C and any number of atomic registers. We define the *schedule reconstruction number* (or *SR number*) of a class C to be the largest n for which C solves n-process schedule reconstruction. If no largest n exists, we say that the SR number of the class is *infinite*.

3 SR and Consensus

In this section, we establish a connection between SR and consensus: SR is at least as hard as consensus.

Theorem 1. *Any class C of objects that solves n-process SR also solves n-process consensus.*

Proof. Let $\mathcal{A}$ be an algorithm solving n-process SR using only objects of class C and atomic registers. We use $\mathcal{A}$ to solve consensus. Each process writes its proposed value in a single-writer, multi-reader register. Then, each processes calls `step` once and then calls `reconstruct`. Thus, every process knows the schedule and is able to decide on the value proposed by the process which was scheduled first.

Corollary 1. *The SR number of a class C is at most equal to its consensus number.*

4 The SR Hierarchy

We examine specific classes of objects according to their ability to solve SR. Due to space limitations, we omit full proofs throughout this section and refer the reader to the full version of the paper [1]. Proof sketches are provided.

4.1 Fetch-and-Increment

Fetch-and-increment objects have consensus number 2 [3] and thus have SR number at most 2 (Corollary 1). We now show that they have SR number exactly 2.

Theorem 2. *Fetch-and-increment has SR number 2.*

Proof. Consider the following protocol for 2-process SR. The two processes share a fetch-and-increment object which initially has value 0. A `step` call simply invokes `getAndIncrement` and receives a (unique) ticket number, which it appends the result to a local list of observations. A `reconstruct` call by p simply assigns to p the steps corresponding to the tickets in p's local observation list and assigns to the other process the steps corresponding to the gaps in p's observation list.

4.2 Compare-and-Swap: A Surprising Result

In this section, we show that the SR number of compare-and-swap (CAS) is 1. We know that it is (trivially) at least 1, by the same argument used for atomic registers. It remains to show that it is also at most 1.

Theorem 3. *CAS has SR number at most 1.*

Proof. We assume towards a contradiction that there exists some algorithm $\mathcal{A}$ for 2-process SR using only CAS objects and registers and examine the first step of $\mathcal{A}$ by each process. This first step cannot be a read, since reads do not modify the observable state of the system and thus cannot be reconstructed. The first step of a process p cannot be a register write either, because immediately after a write p cannot establish whether its write was performed before or after the other process's step. Thus, the first step of both processes must be a CAS. Both CAS's must succeed, because a failed CAS does not modify the observable system state. Moreover, both CAS's must be executed on the same memory location, otherwise they would commute. However, if two CAS's succeed in some schedule S, at least one of them will fail in a schedule S' in which the order of the CAS's is reversed—making S' not reconstructible by $\mathcal{A}$, a contradiction.

4.3 Multiple Atomic Append: Every Level Is Populated

An *append register* is similar to a regular register, except that every write appends its value to the current value of the register, instead of overwriting it. A k-writer append register is an append register from which any number of processes can read but to which only k processes can append. Interestingly, append registers have been studied in a Byzantine setting as well [4].

Theorem 4. *k-writer append registers have SR number $s = k$.*

Proof. First, we show that $s \geq k$. A SR algorithm for k processes using a shared k-writer append register r is as follows. A `step` call appends the invoking process's id to r. A `reconstruct` call reads r and assigns step numbers to processes according to the order of id's in r. It remains to show that $s < k + 1$. Assume towards a contradiction that there exists a SR algorithm for $k + 1$ processes using only k-writer append registers and atomic read-write registers. We consider the first step of the algorithm for each process. Similarly to the proof of Theorem 3,

all processes must access append registers during their first step. Because there are $k+1$ processes but the append registers only support k writers, there must exist two processes which do not write to the same append register for their first step. Thus, their appends commute and are not reconstructible.

4.4 `SWAP3`: the Hierarchy is Infinite

We define a new primitive called `SWAP3`. `SWAP3` takes three arguments `a`, `b` and `c`. It atomically writes the value of `b` into `c` and the value of `a` into `b`.

Theorem 5. *`SWAP3` has SR number ∞.*

Proof. We describe an algorithm that solves SR for any number of processes. The processes maintain a shared linked list which encodes the schedule. A `step` call prepares a new node with the invoking process's id and appends it to the head of the list (a single global step using `SWAP3`: atomically assign the head of the list to point to the new node and the new node's `next` field to the old value of the head). Reconstructing the schedule is done by traversing the linked list and assigning step numbers to processes in reverse order.

5 A SR-Based Universal Construction

In this section, we examine the relationships between the levels in the SR hierarchy. We give two main results: a positive one—stronger objects can implement weaker objects—and a negative one—weaker objects cannot implement stronger objects in a way that preserves reconstructibility.

We begin with the positive result: in a system of n processes, given any object A with SR number $\geq n$ and any deterministic object B, A implements B. By definition of SR number, A can be used to implement SR objects in a system of n processes. Furthermore, B can be implemented from SR objects in the following way (full details in our paper [1]). The processes use an SR object to determine the order in which their invocations take effect and then use this information to simulate the execution on local copies of B.

We have just shown that in the SR hierarchy, as in the consensus hierarchy, there exist objects that are *universal*. Given sufficiently many of them, any object with a sequential specification can be implemented in a wait-free linearizable way.

We now turn to the negative result: in a system of n processes, given any object A with SR number $\geq n$ and any object B with SR number $< n$, B cannot 1-implement A. We say that A 1-implements B if A implements B and the implementation performs at most one shared memory accesses per call to B's methods. Towards a contradiction, assume that B can 1-implement A in a system of n processes. Since A has SR number $\geq n$, there exists an implementation of a SR object from A and atomic registers. By replacing A in this implementation with its 1-implementation from B, we obtain a valid implementation of an SR object from B, a contradiction of the fact that B's SR number is less than n.

Note that this negative result does not contradict the universality of objects in the sense of consensus, which states that an object with consensus number at least n can implement any object in a system of n or less processes. Our negative result states that objects with lower SR number cannot *1-implement* objects with higher SR number. So, for instance, CAS has infinite consensus number, so it can implement any object, but it has SR number 1, so it cannot *implement in a single step* any object with SR number larger than 1 (e.g., fetch-and-increment) in a system of 2 or more processes. In other words, no such object can be implemented from CAS in such a way that the implementation has the same SR number as the abstract object.

6 Conclusion

In this paper, we propose the schedule reconstruction problem and the SR number as a new measure for the disclosure power of objects in shared memory systems. Objects can be organized in a dense hierarchy where strong objects implement weaker objects via a universal construction based on SR. Furthermore, we identify a link between SR and consensus and show that SR is at least as hard as consensus. Finally, we evaluate the SR number of well known objects and show that universal consensus objects are not always universal SR objects.

References

1. Blanchard, P., Guerraoui, R., Stainer, J., Zablotchi, I.: The disclosure power of shared objects. Technical report, EPFL (2017)
2. Herlihy, M.: Wait-free synchronization. TOPLAS **13**(1), 124–149 (1991)
3. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)
4. Imbs, D., Rajsbaum, S., Raynal, M., Stainer, J.: Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. JPDC **93**, 1–9 (2016)