Consensus number du contrôle d'accès: le cas des AllowLists et DenyLists

Davide Frey, Mathieu Gestin, Michel Raynal

Consensus number *du contrôle d'accès: le cas des AllowLists et DenyLists*

Davide Frey[1] †et Mathieu Gestin[1] et Michel Raynal[1]

[1] *Université de Rennes, INRIA, Irisa, CNRS*

Cet article étudie le pouvoir de synchronisation de deux types objets distribués : les AllowLists et les DenyLists. Cette étude est conduite au regard de la hiérarchie du consensus de Herlihy. Ces objets sont d'abord formalisés sous la forme d'objets distribués. Bien que les deux objets aient des spécifications proches, leurs *consensus number* sont foncièrement différents. L'AllowList peut être implémentée sans consensus parmi les processus du système (son *consensus number* est de 1) alors que la DenyList requiert un consensus parmi un sous ensemble spécifique des processus. Ces résultats sont ensuite utilisés pour analyser des systèmes respectant la vie privée et utilisant des AllowLists et des DenyLists.

Full version : Ce travail a été publié dans la conférence DISC 2023. Sa version complète est disponible : `https://doi.org/10.48550/arXiv.2302.06344v2`

Mots-clefs : Access control, AllowList/DenyList, Blockchain, Consensus number, Distributed objects, Modularity, Privacy, Synchronization power

1 Introduction

The advent of blockchain technologies increased the interest of the public and industry in distributed applications, giving birth to projects that have applied blockchains in a plethora of use cases. These include e-vote systems [1] or Identity Management Systems [2]. However, this use of the blockchain as a swiss-army knife that can solve numerous distributed problems highlights a lack of understanding of the actual requirements of those problems. Because of these poor specifications, implementations of these applications are often sub-optimal.

This paper thoroughly studies a class of problems widely used in distributed applications and provides a guideline to implement them with reasonable but sufficient tools. Differently from the previous approaches, it aims to understand the amount of synchronization required between processes of a system to implement *specific* distributed objects. To achieve this goal it studies such objects under the lens of Herlihy's consensus number [3]. This parameter is inherently associated to shared memory distributed objects, and has no direct correspondence in the message-passing environment. However, in some specific cases, this information is enough to provide a better understanding of the objects analyzed, and thus, to gain efficiency in message-passing implementations. For example, recent papers [4, 5] have shown that cryptocurrencies can be implemented without consensus and therefore without a blockchain. The consensus number of thoses distributed objects depends on the size of a set of well identified processes. From this study, it is possible to conclude that the consensus algorithms only need to be performed between those processes. Therefore, in these specific cases, the knowledge of the consensus number of an object can be directly used to implement more efficient message-passing-based applications. This paper proposes to extend this knowledge to a broader class of applications.

Indeed, the transfer of assets, be them cryptocurrencies or non-fungible tokens, does not constitute the only application in the Blockchain ecosystem. In particular, as previously indicated, a number of applications like e-voting [1], or Identity Management [2] use Blockchain as a tool to implement some form of

†This work was partially funded by the PriCLeSS project and by the SOTERIA H2020 project. PriCLeSS was granted by the Labex CominLabs excellence laboratory of the French ANR (ANR-10-LABX-07-01). SOTERIA received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No101018342. This content reflects only the author's view. The European Agency is not responsible for any use that may be made of the information it contains.

access control. This is often achieved by implementing two general-purpose objects : AllowLists and DenyLists. An AllowList provides an opt-in mechanism. A set of managers can maintain a list of authorized parties, namely the AllowList. To access a resource, a party (user) must prove the presence of an identifier associated with its identity in the AllowList. A DenyList provides instead an opt-out mechanism. In this case, the managers maintain a list of revoked identifiers, the DenyList. To access a resource, a party (user) must prove that no corresponding identifier has been added to the DenyList. In other words, AllowList and DenyList support, respectively, set-membership and set-non-membership proofs on a list of identifiers.

Albeit similar, the AllowList and DenyList objects differ significantly in the way they handle the proving mechanism. In the case of an AllowList, no security risk appears if access to a resource is prohibited to a process, even if a manager did grant this right. As a result, a transient period in which a user is first allowed, then denied, and then allowed again to access a resource poses no problem. Therefore, an AllowList can be implemented without consensus, i.e. it has consensus number 1. On the contrary, with a DenyList, being allowed access to a resource after being denied it poses serious security problems. Hence, the DenyList object is defined with an additional anti-flickering property prohibiting such transient periods. Thus, the consensus number of a DenyList is instead equal to the number of processes that can conduct PROVE operations on the DenyList, and that only these processes need to synchronize. Interestingly, even if both structures have different consensus numbers, they can both be implemented without relying on the network-level consensus provided by a blockchain, which opens the door to more efficient implementations of applications based on these data structures.

2 Preliminaries

Model. Let Π be a set of N asynchronous sequential crash-prone processes $p_1, \cdots, p_N$. Sequential means that each process invokes one operation of its own algorithm at a time. We assume the local processing time to be instantaneous, but the system is asynchronous. This means that non-local operations can take a finite but arbitrarily long time and that the relative speeds between the clocks of the different processes are unknown. Finally, processes are crash-prone : any number of processes can prematurely and definitely halt their executions. A process that crashes is called *faulty*. Otherwise, it is called *correct*. The system is eponymous : a unique positive integer identifies each process, and this identifier is known to all other processes.

Communication. Processes communicate via shared objects of type T. Each operation on a shared object is associated with two *events* : an *invocation* and a *response*.

Consensus number. The consensus number of an object of type T (noted cons(T)) is the largest n such that it is possible to wait-free implement a consensus object from atomic read/write registers and objects of type T in a system of n processes. If an object of type T makes it possible to wait-free implement a consensus object in a system of any number of processes, we say the consensus number of this object is ∞.

3 The AllowList and DenyList objects : Definition and consensus number

Formal definition. Distributed AllowList and DenyList object are objects that allow a set of managers to control access to a resource. The term "resource" is used here to describe the goal a user wants to achieve and which is protected by an access control policy. A user is granted access to the resource if it succeeds in proving that it is authorized to access it. First, we describe the AllowList object type. Then we consider the DenyList object type.

The AllowList object type is one of the two most common access control mechanisms. To access a resource, a process $p \in \Pi_V$ needs to prove it knows some identifier v previously authorized by a process $p_M \in \Pi_M$, where $\Pi_M \subseteq \Pi$ is the set of managers, and $\Pi_V \subseteq \Pi$ is the set of processes authorized to conduct proofs. We call verifiers the processes in Π_V. The sets Π_V and Π_M are predefined and static. They are parameters of the object. Depending on the usage, these subset can either be small, or they can contain all the processes in Π.

A process $p \in \Pi_V$ proves that v was previously authorized by invoking a PROVE(v) operation. This operation is said to be valid if some manager in Π_M previously invoked an APPEND(v) operation. Intuitively, we can see the invocation of APPEND(v) as the action of authorizing some process to access the

resource. On the other hand, the PROVE(v) operation, invoked by a prover process, $p \in \Pi_V$, proves to the other processes in Π_V that they are authorized. However, this proof is not enough in itself. The verifiers of a proof must be able to verify that a valid PROVE has been invoked. To this end, the AllowList object type is also equipped with a READ() operation. This operation can be invoked by any process in Π and returns a random permutation of all the valid PROVE invoked, along with the identity of the processes that invoked them. All processes in Π can invoke the READ operation.

An optional anonymity property can be added to the AllowList object to enable privacy-preserving implementations. This property ensures that other processes cannot learn the value v proven by a PROVE(v) operation.

Definition 1. The *AllowList* object type supports three operations : APPEND, PROVE, and READ. These operations appear as if linearized in a sequence Seq such that :

- *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- APPEND *Validity.* The invocation of APPEND(x) by a process p is valid **if** $p \in \Pi_M \subseteq \Pi$ **and** $x \in \mathscr{S}$, where $\mathscr{S}$ is a predefined set. Otherwise, the operation is invalid.
- PROVE *Validity.* **If** the invocation of op =PROVE(x) by a process p is valid, **then** $p \in \Pi_V \subseteq \Pi$ **and** a valid APPEND(x) appears before op in Seq. Otherwise, the invocation is invalid.
- *Progress.* **If** a valid APPEND(x) is invoked, **then** there exists a point in Seq such that any PROVE(x) invoked after this point by any process $p \in \Pi_V$ will be valid.
- READ *Validity.* The invocation of op =READ() by a process $p \in \Pi$ returns the list of valid invocations of PROVE that appears before op in Seq along with the names of the processes that invoked each operation.
- *Optional - Anonymity.* Let us assume the process p invokes a PROVE(v) operation. If the process p' invokes a READ() operation, then p' cannot learn the value v unless p leaks additional information. ‡

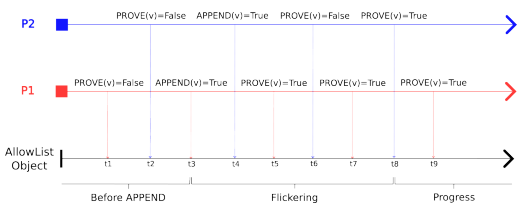
Informally, the DenyList object is an access policy where, contrary to the AllowList object type, all users are authorized to access the resource in the first place. The managers are here to revoke this authorization. A manager revokes a user by invoking the APPEND(v) operation. A user uses the PROVE(v) operation to prove that it was not revoked. All the processes in Π can verify the validity of a PROVE operation by invoking a READ() operation. This operation is similar to the AllowList's READ operation. It returns the list of valid PROVE invocations along with the name of the processes that invoked it.

There is one significant difference between the DenyList and the AllowList object types. With an AllowList, if a user cannot access a resource immediately after its authorization, no behavior can harm the system. However, with a DenyList, a revocation not taken into account can let a user access the resource and harm the system. In other words, access to the resource in the DenyList case must take into account the "most up to date" available revocation list. To this end, the DenyList object type is defined with an additional property. The anti-flickering property ensures that if an APPEND operation is taken into account by one PROVE operation, it will be taken into account by every subsequent PROVE operation.

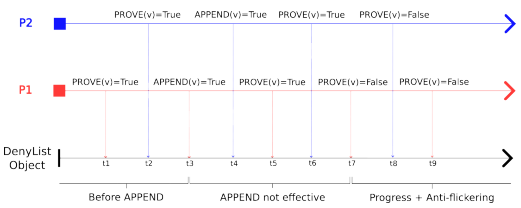
Definition 2. The *DenyList* object type supports three operations : APPEND, PROVE, and READ. These operations appear as if linearized in a sequence Seq such that :

- *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- APPEND *Validity.* The invocation of APPEND(x) by a process p is valid **if** $p \in \Pi_M \subseteq \Pi$ **and** $x \in \mathscr{S}$, where $\mathscr{S}$ is a predefined set. Otherwise, the operation is invalid.
- PROVE *Validity.* **If** the invocation of a op =PROVE(x) by a correct process p is not valid, **then** $p \notin \Pi_V \subseteq \Pi$ **or** a valid APPEND(x) appears before op in Seq. Otherwise, the operation is valid.
- PROVE *Anti-Flickering.* **If** the invocation of a operation op =PROVE(x) by a correct process $p \in \Pi_V$ is invalid, **then** any PROVE(x) that appears after op in Seq is invalid.
- READ *Validity.* The invocation of op =READ() by a process $p \in \Pi$ returns the list of valid invocations of PROVE that appears before op in Seq along with the names of the processes that invoked each

‡. The Anonymity property only protects the value v. The system considered is eponymous. Hence, the identity of the processes is already known. However, the anonymity of v makes it possible to hide other information.



(a) Example execution of an AllowList object.



(b) Example execution of a DenyList object.

operation.

— *Optional - Anonymity.* Let us assume the process p invokes a PROVE(v) operation. If the process p' invokes a READ() operation, then p' cannot learn the value v unless p leaks additional information.

Consensus Number. It is shown in the full version of this paper that the AllowList can be implemented using an atomic snapshot object. The idea is that the PROVE operations on a specific AllowList object do not need to be synchronized. Hence, and as we can see in fig. (a), operations on the object can be linearized in any order, which implies that they don't need more power than what is provided by an atomic snapshot.

Unlike the AllowList, and as we can see in fig. (b) the anti-flickering property of the DenyList object forces an order between the valid PROVE operations and the invalid PROVE operation. This order implies that, to implement a DenyList, a k-consensus algorithm is necessary. Furthermore, and thanks to this anti-flickering property, the DenyList object where the size of the set Π_V is at least k can implement a k-consensus object.

4 Discussion and conclusion

This paper presented the first formal definition of distributed AllowList and DenyList object types. These definitions made it possible to analyze their consensus number. This analysis concludes that no consensus is required to implement an AllowList object, while, with a DenyList object, all the processes that can propose a set-non-membership proof must synchronize, which makes the implementation of a DenyList more resource intensive.

The definition of AllowList and DenyList as distributed objects makes it possible to study other distributed objects that can use AllowList and DenyList as building blocks. For example, it is possible to prove that the consensus number of an e-vote system is m, where m is the number of voting booths. Furthermore, it is possible to show that an association of DenyList and AllowList objects can implement an anonymous asset transfer algorithm. This result can also be generalized to any asset transfer algorithm, where the processes act as proxies for the wallet owners. In this case, synchronization is only required between the processes that can potentially transfer money on behalf of a given wallet owner. The study of the use cases of the distributed AllowList and DenyList objects are presented in the full version of this paper.

Références

[1] Gaby G. Dagher, Praneeth Babu Marella, Matea Milojkovic, and Jordan Mohler. Broncovote : Secure voting system using ethereum's blockchain. In *ICISSP*, 2018.

[2] Sovrin Foundation. Sovrin : A protocol and token for self-sovereign identity and decentralized trust. Technical report, Sovrin Foundation, 2018.

[3] Maurice P Herlihy and Jeannette M Wing. Linearizability : A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3) :463–492, 1990.

[4] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC '19*, page 307–316, 2019.

[5] Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money Transfer Made Simple : a Specification, a Generic Algorithm, and its Proof. *Bulletin European Association for Theoretical Computer Science*, 132, October 2020.