The Complexity of Reliable and Secure Distributed Transactions

THÈSE N^{O} 8761 (2018)

PRÉSENTÉE LE 6 SEPTEMBRE 2018
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE PROGRAMMATION DISTRIBUÉE
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jingjing WANG

acceptée sur proposition du jury:

Prof. J.-Y. Le Boudec, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. G. Alonso, rapporteur
Prof. R. Oshman, rapporteuse
Prof. C. Koch, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2018

A journey of a thousand miles
must begin with a single step.
— Laozi

To my parents,
Mingfang Xie and Hongyu Wang

Acknowledgements

First and foremost, I would like to thank my advisor, Rachid Guerraoui for his constant support and guidance. I do not only receive guidance from him on technical work but also learn a lot of soft skills including presentation skills, marketing skills and tenacity. More importantly, Rachid has taught me to be optimistic. Our work is technically challenging and practically interesting. We are confident in our work and thus all we need to do is to present properly our solid work and eventually people will be interested in reading it.

I must also thank the committee members for my private Ph.D. defense, namely Professor Le Boudec Jean-Yves for being the president of my Ph.D. committee, Professor Koch Christoph for being the internal examiner and Professor Alonso Gustavo and Professor Oshman Rotem for being my external examiners. I would like to thank for their time and insights that help improve this dissertation work.

I am also grateful to my colleagues. To Rhicheek Patra for being always available for discussion for our project on recommender systems and to Mahammad Valiyev for the same project. To Antoine Rault and Davide Frey for their time and efforts in our collaboration on private KNN computation even when we were geographically separated. To Diego Didona for his insights from the perspective of practitioners for our hard work on causal transactions. I would like also to thank Professor Anne-Marie Kermarrec, Professor François Taïani and Professor Willy Zwaenepoel for their patience, advice, and insightful feedback for my research projects, which I learned from greatly. Our previous secretary of the laboratory, Kristine Verhamme and our current secretary France Faille also helped me a lot on the administrative steps to attending conferences and to going through my Ph.D. journey etc., while our system administrator Fabien Salvi helped me out in installing and configuring necessary software for my research.

Last but not least, I would like to especially thank my family and my friends. Without my friends in Switzerland, back in China and even far away in the United States who made my Ph.D. days an enjoyable experience, I could not possibly survive my Ph.D. journey. Moreover, my education (i.e., diploma, M.Sc., and Ph.D.) to date has lasted for 12 years. I could not possibly be here and sustain all the pressure (e.g., of the financial expenses) without the help and support of my family. I want to deeply thank my parents for being there for me throughout all these years.

Lausanne, 2018 Jingjing Wang

Preface

This dissertation concerns the PhD work I did under the supervision of Prof. Rachid Guerraoui at the School of Computer and Communication Sciences, EPFL, from 2013 to 2018. The main results of this dissertation appeared originally in the following publications (author names are in alphabetical order).

1. Rachid Guerraoui, Jingjing Wang. "Optimal Fair Computation". Proceedings of the 30th International Symposium on Distributed Computing (DISC). Springer, 2016. (Chapter 4)
2. Rachid Guerraoui, JingjingWang. "How Fast can a Distributed Transaction Commit?". Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (SIGMOD/PODS). ACM, 2017. (Chapter 2)
3. Diego Didona, Rachid Guerraoui, Jingjing Wang, Willy Zwaenepoel. "Distributed Transactions: Dissecting the Nightmare". Under submission. (Chapter 3)

Besides the work presented in this thesis, I also worked on the following publications (author names are in alphabetical order).

1. Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Antoine Rault, François Taïani, Jingjing Wang. "Hide & Share: Landmark-based Similarity for Private KNN Computation". Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2015.
2. Rachid Guerraoui, Anne-Marie Kermarrec, Rhicheek Patra, Mahammad Valiyev, Jingjing Wang. "I know nothing about you but here is what you might like". Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2017.
3. Rachid Guerraoui, Jingjing Wang. "On the Unfairness of Blockchain". Proceedings of the 6th International Conference on Networked Systems (NETYS). Springer, 2018.
4. Diego Didona, Rachid Guerraoui, Jingjing Wang, Willy Zwaenepoel. "Causal Consistency and Latency Optimality: Friend or Foe?". Proceedings of the 44th International Conference on Very Large Data Bases (VLDB). 2018.

Abstract

The use of transactions in distributed systems dates back to the 70's. The last decade has also seen the proliferation of transactional systems. In the existing transactional systems, many protocols employ a centralized approach in executing a distributed transaction where one single process coordinates the participants of a transaction. The centralized approach is usually straightforward and efficient in the failure-free setting, yet the coordinator then turns to be a single point of failure, undermining reliability/security in the failure-prone setting, or even be a performance bottleneck in practice.

In this dissertation, we explore the complexity of decentralized solutions for reliable and secure distributed transactions, which do not use a distinguished coordinator or use the coordinator as little as possible. We show that for some problems in reliable distributed transactions, there are decentralized solutions that perform as efficiently as the classical centralized one, while for some others, we determine the complexity limitations by proving lower and upper bounds to have a better understanding of the state-of-the-art solutions.

We first study the complexity on two aspects of reliable transactions: atomicity and consistency. More specifically, we do a systematic study on the time and message complexity of *non-blocking atomic commit* of a distributed transaction, and investigate intrinsic limitations of *causally consistent* transactions. Our study of distributed transaction commit focuses on the complexity of the most frequent executions in practice, i.e., failure-free, and willing to commit. Through our systematic study, we close many open questions like the complexity of synchronous non-blocking atomic commit. We also present an effective protocol which solves what we call *indulgent atomic commit* that tolerates practical distributed database systems which are synchronous "most of the time", and can perform as efficiently as the two-phase commit protocol widely used in distributed database systems.

Our investigation of causal transactions focuses on the limitations of read-only transactions, which are considered the most frequent in practice. We consider "fast" read-only transactions where operations are executed within one round-trip message exchange between a client seeking an object and the server storing it (in which no process can be a coordinator). We show two impossibility results regarding "fast" read-only transactions. By our impossibility results, when read-only transactions are "fast", they have to be "visible", i.e., they induce inherent updates on the servers. We also present a "fast" read-only transaction protocol that is "visible" as an upper bound on the complexity of inherent updates.

We then study the complexity of secure transactions in the model of *secure multiparty computation*: even in the face of malicious parties, no party obtains the computation result unless all other parties obtain the same result. As it is impossible to achieve without any trusted party, we focus on *optimism* where if all parties are honest, they can obtain the computation result without resorting to a trusted third party, and the complexity of every optimistic execution where all parties are honest. We prove a tight lower bound on the message complexity by relating the number of messages to the length of the *permutation sequence* in combinatorics, a necessary pattern for messages in every optimistic execution.

Keywords: complexity, failures, distributed transactions, non-blocking atomic commit, indulgent atomic commit, causal consistency, optimistic secure multiparty computation, permutation sequence

Résumé

L'utilisation des transactions dans les systèmes distribués remonte aux années 70. La dernière décennie a également vu la prolifération des systèmes transactionnels. Dans les systèmes transactionnels existants, de nombreux protocoles utilisent une approche centralisée pour exécuter une transaction distribuée : un seul processus coordonne les processus rattachés à la transaction. L'approche centralisée est généralement simple, et elle est efficace en l'absence de défaillance. Et pourtant, le coordinateur devient un point unique de défaillance, compromettant la fiabilité / la sécurité en cas de défaillance, ou même un goulot d'étranglement de performances en pratique.

Dans ce mémoire, nous examinons la complexité des solutions décentralisées pour des transactions distribuées fiables et sécurisées, qui n'utilisent pas un coordinateur ou utilisent le coordinateur le moins possible. Nous montrons que pour certains problèmes de transaction distribuée fiable, il y a des solutions décentralisées qui fonctionnent aussi efficacement que les solutions centralisées classiques, tandis que pour d'autres, nons fournissons les limites de complexité par la détermination des limites inférieures et supérieures, afin de mieux comprendre ce qu'est « l'état de l'art ».

Nous présentons d'abord deux analyse de la complexité sur deux propriétés des transactions fiables, atomicité et cohérence, respectivement. Plus spécifiquement, nous effectuons une étude systématique de la complexité en temps et message de *validation atomique non-bloquante* d'une transaction distribuée, et étudions les limitations intrinsèques des transactions *causalement cohérentes*. Notre étude de la validation des transactions distribuées se concentre sur la complexité des exécutions en l'absence de défaillance où la décision est « valider », qui sont considérées comme étant les exécutions les plus fréquentes en pratique. Grâce à notre étude systématique, nous résolvons de nombreuses questions ouvertes comme la complexité de la validation atomique non-bloquante synchrone. Nous présentons également un protocole efficace qui résout ce que nous appelons *validation atomique indulgent* qui tolère la pratique où les systèmes de base de données distribués sont synchrones « la plupart du temps ». Le protocole peut fonctionner aussi efficacement que le protocole de validation à deux phases largement utilisé dans les systèmes de bases de données distribuées.

Notre étude des transactions causales met l'accent sur les limites des transactions en lecture seule, qui sont considérées comme les plus fréquentes en pratique. Nous considérons les transactions en lecture seule « rapide » où les opérations sont exécutées dans un échange de messages d'un aller et retour entre un client cherchant un objet et le serveur le stockant (où aucun processus ne peut être un coordinateur). Nous montrons deux résultats d'impossibilité concernant les transactions « rapides » en lecture seule. Selon nos résultats d'impossibilité, lorsque les transactions en lecture seule sont « rapides », elles doivent être « visibles », c'est-à-dire qu'elles induisent des mises à jour inhérentes sur les serveurs. Nous présentons également un protocole de transaction « rapide » en lecture seule qui est « visible » en tant que limite supérieure de la complexité des mises à jour inhérentes.

Nous étudions ensuite la complexité des transactions sécurisées dans le modèle de *calcul multipartite sécurisé* : même face à des parties malveillantes, aucune partie n'obtient le résultat du calcul à moins que toutes les autres parties n'obtiennent le même résultat. Comme il est impossible de réaliser sans aucune partie de confiance, nous nous concentrons sur *optimisme* où si toutes les parties sont honnêtes, elles peuvent obtenir le résultat sans recourir à une tierce partie de confiance. Nous nous concentrons sur la complexité de chaque exécution optimiste où toutes les parties sont honnêtes. Nous montrons une limite inférieure serrée de la complexité en message en reliant le nombre de messages à la longueur de la *séquence de permutation* en combinatoire. La séquence de permutation représente un schéma nécessaire dans l'échange de messages de chaque exécution optimiste.

Mots-clés : complexité, défaillance, transactions distribuées, validation atomique non-bloquante, validation atomique indulgent, cohérence causale, calcul multipartite sécurisé optimisé, séquence de permutation

Contents

4 The Complexity of Optimistic Secure Transactions 101

List of Figures

List of Tables

1 Introduction

A distributed transaction is a transaction that spans multiple participants [1]. Gray [2] described a transaction as a group of actions that transform the state of multiple data items in a consistent way. Distributed transactions lie at the heart of many recent distributed database systems such as Helios [3], where database nodes are participants and jointly decide the outcome of a distributed transaction. Distributed transactions are also a key component in distributed transactional storage systems such as Cassandra [4], where both storage servers and users of storage are participants and where a user may interact with multiple servers via a transaction. Distributed transactions also play a role in electronic commerce, focusing on security. These transactions are called secure transactions and fall into the category of secure multi-party computation [5, 6].

A lot of effort has been devoted to improving the performance of distributed transactions. To further improve the performance, we study the complexity of distributed transactions and in this dissertation, we focus on reliable and secure transactions. Roughly speaking, reliable and secure transactions ensure the correctness of transaction execution in the face of failures. Distributed transaction would be easy to implement if there were no failure. Here no failure is two-fold: (1) no participant crashes; all participants follow the assigned protocol (for distributed transactions) faithfully; and (2) the communication delay is upper-bounded by a known value. In the case of no failure, although the transaction spans multiple participants, the following centralized solution can be proposed. One distinguished process called coordinator orchestrates database nodes, multiple storage servers, or all the participants involved in the transaction. The coordinator simply receives a request from each participant, and computes a response for each participant, which then completes the transaction. However, the coordinator itself thus becomes a single point of failure and may be considered a performance bottleneck as well. In addition, according to reports on network failure [7, 8], storage failure [9] and node failure [10, 11], failures largely exist. As a result, a solution that avoids a coordinator and takes failures into account is practically appealing. The motivation of this dissertation is to study the complexity and propose optimal protocols of such solution in the context of reliable and secure distributed transactions.

1.1 Reliable and secure transactions

1.1.1 Reliable transactions

A reliable transaction is required to follow the ACID properties [2, 12]: *atomicity*, *consistency*, *isolation* and *durability*, among which in this dissertation, we focus on atomicity and isolation. If a transaction is atomic, then either the transaction executes to its completion and its effects persist (i.e., a transaction *commits*), or the transaction appears to have not executed at all (i.e., a transaction *aborts*). Isolation levels are defined based on the behavior of concurrent transactions [13, 14]. Hereafter we use the terminology "transaction consistency" to refer to isolation. In conventions, the term consistency may refer to (1) the enforcement of predefined rules on data by transactions; or (2) criteria on the behavior of transactions, for example, the outcome and/or actions of concurrent transactions. Our usage of the term refers to the latter, where consistency and isolation are indeed correlated concepts [15].

Atomicity

In a distributed transaction that spans multiple database nodes, each node executes a subtransaction. Here each subtransaction may contribute to aborting the final transaction if the subtransaction is denied access to some data (for example, due to lock conflict or requirements of certain isolation levels). Each node can be considered to have the right to cast a vote of 0 (abort) or 1 (commit) according to the failure or success of its subtransaction. To preserve the atomicity of the distributed transaction, all nodes have to agree on one single decision. Clearly, these nodes agree to commit only if all votes are 1. The protocol defined to orchestrate distributed transaction commit is called a *commit protocol* [16].

The commit protocol employed by many distributed database systems (for instance, Sinfonia [17], Percolator [18], Spanner [19], Clock-SI [20] and Yesquel [21]) is two-phase commit (2PC). The 2PC protocol can be considered as one centralized solution mentioned above. Roughly speaking, a coordinator receives the vote from each node, decides the outcome based on all votes and then informs each node of the decision. In its original form [22], as explained by [23], when the coordinator *crashes*, then the outcome on the transaction is unknown and can block nodes and clients which wait for an outcome.

Various methods have been implemented to mitigate the risk over the crash failure of the coordinator. For example, a distinguished node can probe the coordinator for failure detection and can coordinate the rest of the nodes to continue the commit protocol [17, 21]. If locks are left on some data, then the coordinator of later transactions may try to remove these locks [18]. Another way is to replicate the coordinator (as well as each node) with Paxos state machines [24] so that the coordinator is implemented by multiple physical nodes to mitigate the crash failure itself [19]. Despite extra effort in dealing with crash failures, previous non-blocking protocols such as three-phase commit [16] are not widely used due to their additional time complexity compared with 2PC (for example in Sinfonia [17]). On the other hand, few commit protocols are designed for a practical network where messages can be delayed out of some bounds from time to time (we say a *network failure* occurs). In addition, little was known on the complexity of commit protocols except for few results on the case where only crash failures are considered [1, 25, 26].

Transaction consistency

As is mentioned previously, transaction consistency here refers to the isolation property of reliable transactions. ANSI SQL standard [13] specified four levels of isolation: read uncommitted, read committed, repeatable read, anomaly serializable (named by a later article [14]). Roughly speaking, ANSI SQL isolation levels define the output of reads in a transaction when some update transaction is concurrent [13, 14]. In addition to ANSI isolation levels, many database systems including MS SQL server [27], Oracle Berkeley DB [28] and PostgreSQL [29] support snapshot isolation [14] or serializable snapshot isolation [30]. As transaction consistency criteria, snapshot isolation and serializable snapshot isolation require that at least one between two concurrent transactions that write the same object must abort [14, 30].

In a update-anywhere implementation of data storage (as in geo-distributed storage Walter [31]), data are replicated such that multiple physical nodes (called replicas) can respond to requests of access to the same item [32]. In this setting, snapshot isolation and serializable snapshot isolation cannot be implemented without synchronous communication among replicas during a transaction [33]. Here synchronous communication means the completion of a transaction waits for some responses from other replicas. The possibility of network partition and consideration over latency between geo-distributed database nodes unfavours these isolation levels [33]. There is a trend for distributed data store and database services to choose not to support isolation levels as the traditional SQL databases above. For instance, Amazon Dynamo [34] and Cassandra [4] adopt eventual consistency, which allows an update to be eventually communicated with all replicas. While in the original definition of eventual consistency, transactions are not considered, eventually consistent storage like Dynamo and Cassandra indeed does not support transactions by default.

Recently, quite a few transactional storage systems adopt causal transactions such as COPS [35], Eiger [36], Orbe [37], GentleRain [38], SwiftCloud [39], Cure [40] and Occult [41]. Causal transactions allow conflicts of concurrent updates to be resolved asynchronously and in these storage systems, causal transactions are implemented without synchronous communication among replicas during a transaction. Causal consistency was initially defined for single accesses of read or write in memory [42] and was then extended to transactions [43]. Different from traditional transactions (which for example under snapshot isolation, can abort some concurrent transactions), causal transactions do not need to abort as shown in existing systems [35, 36, 37, 38, 39].

Recent work [36, 38] compares causal transactions with a group of data accesses (called a transaction as an abuse of notations) under eventual consistency. If causal transactions always take two-round communication, then the latency of causal transactions doubles that of eventually consistent transactions [36]. Although there is a gap of performance, causal transactions in most transactional storage [35, 36, 38, 40, 41] can induce more than one-round communication, where a client or some server plays the role of a coordinator. The COPS-SNOW algorithm [44] has causal transactions in one-round communication while the design decisions seem contrived and sometimes, the performance results might not meet expectation [44]. A better understanding of the complexity of causal transactions is thus necessary (even for the best case where crash or network failures, considered for commit protocols, are not taken into account).

1.1.2 Secure transactions

In electronic commerce, a transaction refers to the exchange of goods and services. As the standard Secure Electronic Transaction (SET) [45] shows, different from database transactions, distributed electronic transactions primarily focus on security properties [46, 6], including privacy and authenticity. SET can be considered as a protocol between two parties: buyer and merchant, where their respective banks are trusted and coordinate the exchange between the two parties [45]. Some proposal follows the idea of SET and extends it to more parties. For example, STP [47] stipulates a subtransaction to involve only two party but allows multiple subtransactions with different pairs of parties to join in the same transaction. These proposals involve trusted parties (called a trusted third party in general) in every execution of a protocol.

In general, disputes may arise from the execution of a protocol where different parties in the protocol claim different results of the same transaction. In the case of electronic commerce, a merchant may claim a successful transaction to have failed and double-charge a buyer while a buyer may claim a failed transaction to be successful and ask for e-goods from a merchant. Such behavior deviates from the given protocol, and is considered to be *malicious*. (On the other hand, a party which follows faithfully the given protocol is said to be *honest*.) In face of malicious parties, *fairness*, in the sense that either all parties terminate the transaction with the same output or none of them does, is a necessary security property of secure transactions [48]. Fairness problem is difficult to solve in a truly distributed setting as shown by (1) the FLP impossibility [49] where agreement cannot be achieved if a single party can crash (considered as malicious in the context of secure transactions), for deterministic solutions, and (2) the impossibility of a coin flip [50] where if two parties jointly generate a random bit and one of them can be malicious, then the random bit can always be biased, for randomized algorithms. The difficulty lies in the fact that some malicious behavior can be indistinguishable from some behavior of the *asynchronous* network where a message is only guaranteed to be eventually received, yet honest parties are still guaranteed to have fairness.

Thus a trusted third party (assumed to be honest) is necessarily introduced. However, as discussed previously, a trusted third party can be a single point of failure or a performance bottleneck. As a result, optimistic fair exchange [48], where the trusted third party is not involved when all parties are honest, is appealing. To generalize the possible function executed by a distributed transaction, we consider multi-party computation in general (rather than two-party exchange). In this dissertation, we consider optimistic fair multi-party computation [48, 6] as an equivalent to optimistic secure transaction. Many results have been published on problems related to fair computation [51, 52, 53, 54, 55, 56]. Yet the complexity of optimistic fair multi-party computation is still unknown.

1.2 Contributions

In this dissertation, we study the complexity of the following three specific problems in the context of reliable and secure distributed transactions.

1.2.1 Distributed transaction commit

First, we study the complexity of atomic commit protocols which lie at the heart of reliable distributed transactions. The commit problem can be abstracted as follows. A set of processes (database nodes) aim to agree on whether to commit or abort a transaction (*agreement* property). The commit decision can only be taken if all processes are initially willing to commit the transaction, and this decision must be taken if all processes are willing to commit *and* there is no failure (*validity* property). An atomic commit protocol is said to be *non-blocking* if every correct process (a database node that does not fail) eventually reaches a decision (commit or abort) even if there are failures elsewhere in the distributed database system (*termination* property).

We present the first systematic complexity study of the atomic commit problem. Our result is systematic in two ways: (1) both crash and network failures are considered and (2) we study the complexity according to the *robustness* of the protocol in the face of failures. More specifically, we define a subset of the properties above (validity, agreement, and termination) to be satisfied in failure-prone scenarios (where crash failure or network failure or both can occur) as a robustness metric, and study the complexity of all combinations of all subsets and all failure-prone scenarios.

In Chapter 2, we present this complexity result (time and message complexity) of our systematic study. We measure the best-case complexity [57], in the executions that are considered the most frequent in practice, i.e., failure-free, with all processes willing to commit. Through our systematic study, we answer many open questions like the complexity of *synchronous* non-blocking atomic commit (designed for the system where only crash failures occur). We also present optimal protocols which may be of independent interest. In particular, we present an effective protocol which solves what we call *indulgent atomic commit* that tolerates practical distributed database systems which are synchronous “most of the time”.

1.2.2 Causal transactions

Second, we study the complexity of causal transactions which are a trend of recent transactional storage systems that need not ensure strong consistency but only causality. Departing from strong consistency models, causal transactions can be abstracted as follows. Clients interact with servers (storage) via transactions which group read and write operations of objects in the storage. The causality relation basically defines the order between any two transactions in the following three ways: (1) two transactions performed by the same client ordered according to when the client performs the transactions (program-order causality relation), (2) two transactions of which the latter includes a read which returns the value written by the former (read-from causality relation), and (3) transitivity. Causally consistency ensures that transactions can be ordered in a way that respects causality.

As read-only transactions are usually considered the most frequent in practice, we ask whether read-only transactions can be "fast", i.e., their operations can be executed within one round-trip message exchange between a client seeking an object and the server storing it. Our goal is to have a better understanding of the current design choices and performance results of causal transactions.

In Chapter 3, we present the first study of the inherent cost of "fast" read-only causal transactions, contributing to this understanding. In general storage systems where some transactions are read-only and some also involve write operations, we show that even read-only transactions cannot be "fast". In such systems (as sometimes implemented today) where all transactions are read-only, i.e., updates are performed as individual operations outside transactions, read-only transactions can indeed be "fast", but we prove that they need to be "visible" to the servers in the sense that they induce inherent updates on these servers. The updates in turn impact the overall performance of the transactional storage.

1.2.3 Optimistic secure transactions

Finally, we study the message complexity of optimistic fair computation, as a generalized form of optimistic secure transactions. More specifically, in the problem of multi-party computation, a set of n parties aim to jointly compute a function given their inputs, where the function is previously agreed by all parties. No party obtains the computation result unless all other $n-1$ parties obtain the same result (*fairness* property). If all n parties are honest, then they can obtain the computation result without resorting to a trusted third party (*optimism* property). Different from reliable transactions, to ensure security against malicious behavior, the definition of fairness for optimistic secure transactions follows the classical formulation of secure multi-party computation [46, 6]. Following our complexity study of reliable transactions, we measure the complexity of optimistic fair computation for the best case as well, which is considered the most frequent in practice. Namely, we study the complexity of any *optimistic* execution (of a protocol) where all parties are honest.

In Chapter 4, we prove a lower bound on the message complexity of optimistic fair computation for n parties among which $n-1$ can be malicious in an asynchronous network for any function. We also show the tightness of the lower bound by presenting a matching protocol of optimistic fair exchange (an important function in electronic transactions). In both our proof and our design of an optimal protocol, we relate the optimal message complexity of optimistic fair computation to the length of the shortest permutation sequence in combinatorics [58, 59, 60].

1.3 Thesis Roadmap

The rest of the dissertation is organized as follows.

- Chapter 2 presents an exhaustive study of the complexity of distributed commit protocols.
- Chapter 3 investigates the complexity of read-only transactions in causally consistent systems.
- Chapter 4 presents the message complexity of optimistic fair computation.
- Chapter 5 concludes this dissertation and discusses potential future work.

2 The Complexity of Distributed Transaction Commit[1]

2.1 Introduction

The use of transactions to ensure the consistency of distributed databases systems despite concurrency and failures dates back to the 70's [62, 22, 63], and is still prominent today. Many modern distributed information systems are transactional, including HP's Sinfonia [17], Yahoo's PNUTS [64], Google's Percolator [18] and Spanner [19], Clock-SI [20] and Yesquel [21].

At the heart of those distributed transaction processing systems lies the fundamental *atomic commit* problem [22]. To illustrate the nature of the problem, consider a distributed database system that ensures the serializability of transactions by tracking their concurrency conflicts across datacenters (nodes) as in Helios [3]. In short, each datacenter $\mathscr{D}$ votes to abort every transaction tx that causes a conflict at $\mathscr{D}$. Transaction tx is committed if no datacenter detects any conflict involving tx. To orchestrate the termination of tx, coordination is necessary among datacenters: all have to agree on whether to commit or abort tx, despite failures, and tx cannot be committed if at least one datacenter votes to abort. This coordination is called a distributed commit protocol and its complexity impacts the performance of the entire distributed database system [3].

2.1.1 Problem statement

More specifically, the atomic commit problem consists for a set of nodes of the distributed database system (we simply call them *processes*) to decide whether to *abort* or *commit* a transaction. The decision is based on the *vote* of each process about the local faith of the transaction. A process votes "no" if the transaction did not execute correctly at that process (due to a full disk, a concurrency control problem, etc.). A process votes "yes" (willingness to commit) if the transaction did execute correctly at that process. The processes (a) commit the transaction *only* if all vote to commit, and (b) *have to* commit the transaction if all vote to

---

[1] Postprint version of the article published in SIGMOD/PODS 2017: Rachid Guerraoui and Jingjing Wang. *"How Fast can a Distributed Transaction Commit?"* [61]

commit and there is no failure. This property is usually called *validity* [65, 66, 67, 68, 69]. All processes need to agree on the same decision. This property is called *agreement* [65, 66, 67, 68, 69]. If one additionally stipulates that correct processes (those that do not crash) need to eventually decide (commit or abort) despite failures (e.g., crashes of other processes), then this property is called *termination* [68, 69], and the resulting problem, where processes need to ensure validity, agreement as well as termination, is called *non-blocking atomic commit* (NBAC) [16]. NBAC has been investigated since the 70's by the database and distributed system communities [16, 1, 70, 65, 71, 66, 72, 73].

In this chapter, we present a systematic study of the time and message complexity of the atomic commit problem and study the exact tradeoff between robustness and *best-case* complexity (in the sense of Lamport [57]), i.e., the complexity of any failure-free execution where all processes vote to commit. Such executions, called *nice* executions in this chapter, are arguably the most frequent in practice and are those for which protocols are usually optimized.

Not surprisingly, this complexity depends on *robustness*, i.e., on which property (validity, agreement, termination) is required in which executions (including less likely executions with failures). The most robust form of atomic commit protocol is, roughly speaking, the one that tolerates both *crash failures* (i.e., some process crashes) and *network failures* (e.g., a network partition occurs and later recovers), i.e., all executions with such failures have to solve NBAC. However, by the impossibility result of consensus [74, 49], the most robust form (in an asynchronous network where at least one process can crash) has infinite complexity. On the contrary, the least robust form of atomic commit, of which only *failure-free* executions are required to solve NBAC, is clearly easy to solve in finite complexity. Although there is obviously a tradeoff between robustness and complexity, the exact tradeoff was not clear. Furthermore, between the least and most robust forms of atomic commit, the situation is more complicated and the complexity results harder to obtain.

We exhaustively study complexity in the cases between two extremes, assuming certain robustness of an atomic commit protocol. More precisely, we determine the optimal number of message delays/messages in nice executions of a protocol π assuming that, in π, (1) every *crash-failure* execution satisfies X and (2) every *network-failure* execution satisfies Y, where X and Y are subsets of these three properties: agreement, validity, and termination. With two kinds of failure-prone executions (crash-failure and network-failure) and three properties, we end up with $(2^3)^2 = 64$ possibilities, as shown in Table 2.1. Since a property satisfied in every network-failure execution is also satisfied in every crash-failure execution, the 64 possibilities reduce to 27 different cases, the non-empty cells in Table 2.1.

2.1.2 Previous results

Many distributed database systems (Sinfonia [17], Percolator [18], Spanner [19], Clock-SI [20] and Yesquel [21], for instance) guarantee validity and agreement in crash-failure executions through a two-phase commit (2PC) protocol [22]. 2PC induces two communication rounds

Table 2.1 – Complexity of Atomic Commit. NF = network-failure executions; CF = crash-failure executions; A = agreement; V = validity; T = termination. Fraction d/m in a cell (X, Y) means that the tight lower bounds are d message delays, m messages respectively if (1) every failure-free execution solves NBAC, (2) every crash-failure execution satisfies a set X of properties and (3) every network-failure execution satisfies a set Y of properties. For every empty cell (X, Y), there exists a non-empty cell (Z, Y) such that $X \cup Y = Z$.

CF NF	∅	A	V	T	AV	AT	VT	AVT
∅	$1/0$	$1/0$	$1/n-1+f$	$1/0$	$1/n-1+f$	$1/0$	$1/n-1+f$	$1/n-1+f$
A		$1/0$			$1/n-1+f$	$1/0$		$2/2n-2+f$
V			$1/2n-2$		$1/2n-2$		$1/2n-2$	$1/2n-2$
T				$1/0$		$1/0$	$1/n-1+f$	$1/n-1+f$
AV					$1/2n-2$			$2/2n-2+f$
AT						$1/0$		$2/2n-2+f$
VT							$1/2n-2$	$1/2n-2$
AVT								$2/2n-2+f$

among processes. Although efficient, 2PC does not solve NBAC in crash-failure executions since it does not guarantee termination. However, NBAC can actually be solved in crash-failure executions (by a three-phase commit protocol [16], which has only finite complexity).

Except for some results on the number of messages necessary for synchronous NBAC protocols (which solve NBAC in every crash-failure execution) [1, 25, 26], the fundamental question of the complexity of synchronous NBAC has actually been open for more than three decades [16, 1]. In fact, only the lower bound of $2n-2$ messages in the face of $n-1$ crashes [1] was known before. Although important, little was known on the complexity of atomic commit (e.g., when network failures are also considered) or its tradeoff with robustness, which we address in this chapter.

2.1.3 Our results

Table 2.1 summarizes our results for the 27 atomic commit problems considered. Besides the tradeoff between complexity and robustness (which properties are required in which execution), we also highlight a tradeoff between time and message complexity. We prove that in 18 out of 27 problem variants, the optimal number of message delays and the optimal number of messages cannot be achieved at the same time.

Among the 27 variants, the most robust one, which we call *indulgent atomic commit*, is particularly appealing.[2] Indulgent atomic commit captures the best robustness[3] of a distributed

---

[2] We define indulgent atomic commit in the same vein as indulgent consensus [75, 76] protocols like Paxos [24], CHT [69] and others [77, 78, 79, 80].

[3] The most robust form is in the setting of an asynchronous network where at least one process can crash, which cannot be achieved. The best robustness achieved here tolerates network failures in the setting of an eventually

commit protocol, i.e., despite failures, agreement, validity and termination are still satisfied. We propose a protocol, which we denote by INBAC, that matches the lower bound of two message delays of indulgent atomic commit. Moreover, we prove that INBAC is optimal in the number of messages among all delay-optimal indulgent atomic commit protocols. Thus, in practical distributed database systems that are synchronous “most of the time" [81][4], and where practitioners consider violations of timeouts (e.g., due to network failures), if rare, to be acceptable, INBAC tolerates such violations and is also optimal in complexity for the arguably most frequent executions. Comparing our INBAC protocol with the popular 2PC protocol, we show, interestingly, that (1) INBAC has the same best-case message delay as 2PC if all processes start spontaneously, and (2) in the special case where at most one process can crash (among n processes), INBAC and 2PC use $2n$ and $2n-2$ messages respectively. In this sense, INBAC may be of independent interest, as a more robust yet efficient alternative to 2PC for implementing distributed transactions.

At the same time, we close the question of the complexity of synchronous NBAC (which is one among the 27 cases we consider). We show, for the first time, that for synchronous NBAC, one message delay is optimal. We also generalize Dwork and Skeen’s lower bound of $2n-2$ messages [1] to $n-1+f$ messages in the face of f crashes and propose a matching message-optimal synchronous NBAC protocol.

2.1.4 Techniques

We denote a cell in Table 2.1 by a property pair (X,Y). (X,Y) is less robust than another pair (U,V) if $X \subseteq U$ and $Y \subseteq V$. Then our proof goes through two main steps. First, we group the pairs (X,Y) that give the same number of message delays/messages in Table 2.1 and prove the lower bound for the least robust pair in each group. To design matching protocols, by symmetry, we look for “the most robust pair” in each group. However, as shown in Table 2.1, in some groups, there is no “most robust pair”. Thus, our second step is to choose, in each group, the pairs that are locally maximal in robustness and present a protocol that matches the lower bound for each local maximum.

Three techniques are key to our results.

1. To prove our lower bounds, we introduce and leverage the notion of “process reachability”, the arrival of a message m at process Q that makes Q know process P’s vote, which is necessary in the context of a network-failure execution. (Dwork and Skeen [1] used “process coloring” in proving lower bounds for synchronous NBAC. Compared with our notion, theirs does not distinguish the arrival from the departure of a message, since they solely focus on crash-failure executions, featuring bounded message delays.)

---

synchronous network, which we define precisely in Section 2.2.

[4] It was experimentally shown, e.g., in [81], that the latency of a communication round is below some seconds (most of the time) if the link does not lose too many messages.

2. To design our optimal protocols, we introduce and leverage "implicit" votes for the willingness to commit. For example, to achieve 0-message protocols, instead of receiving a message telling process P process Q's vote, P may know that Q votes 1 by *not* receiving a certain message. We support an optimal nice execution by a complex failure-free execution that aborts.
3. Another technique we use is "helping". To reach the smallest number of messages or message delays in any nice execution, if some failure occurs, then processes must ask for help. To enable helping, backing up votes at other processes is necessary while sometimes a message of acknowledgement (that confirms the success of the backup) is also necessary. Both are key ideas behind INBAC.

The rest of the chapter is organized as follows. Section 2.2 presents the distributed database models we consider, defines the non-blocking atomic commit problem and introduces the building blocks (modules) of the optimal protocols proposed in this chapter. Section 2.3 establishes our lower bounds. Section 2.4 describes atomic commit protocols that meet the lower bounds. Section 2.5 presents indulgent atomic commit, our protocol INBAC and a proof of its optimality. Section 2.6 discusses related work.

2.2 Models and Definitions

2.2.1 Processes and channels

We consider a set Ω of n processes $P_1, P_2, \ldots, P_n$ (sometimes also denoted by O, P, Q, R). Here processes represent database nodes. Processes communicate by exchanging messages, through the network.

We assume that no process deviates from its specification and at most $f, 1 \leq f \leq n-1$ processes can *crash*. After a process crashes, it does not send any message. If a process does not crash, it is said to be *correct*.

Communication channels do not modify, inject, duplicate or lose messages. Every message sent is eventually received.

2.2.2 Failures and executions

We assume *synchronous computation*: there is a known upper bound on the time to execute a local step, which includes the delivery of a message by a process, its local processing by that process, as well as the sending of a message as a consequence of that processing.

Communication is said to be *synchronous* if there is a known upper bound on message transmission delays. Communication is said to be *eventually synchronous* if the delay on message transmission might be unbounded but only until some, possibly unknown, *global stabilization* *time* (after which there is a known upper bound on delays).[5] We accordingly consider two kinds of system models (or simply systems): a synchronous system [1] and an eventually synchronous system [82], based on their respective assumptions on communication.

An execution of a synchronous system is either *failure-free* or has *crash failures*: either all processes are correct, or some process crashes, while all message transmission delays are smaller than some known upper bound which we denote by U.[6] If, in some execution, some message transmission delay is greater than U, then the system is no longer synchronous: we say that a *network failure* occurs. An execution of an eventually synchronous system can be failure-free, has crash failures, or network failures. We call a *failure-free* execution an execution where no failure occurs, a *crash-failure* execution one execution of a synchronous system (where only crash failures are possible) and a *network-failure* execution one execution of an eventually synchronous system (where network failures are also possible). We accordingly call a synchronous system and an eventually synchronous system, a crash-failure system and a network-failure system, respectively.

2.2.3 Non-blocking atomic commit

We consider the problem of non-blocking atomic commit (NBAC) in the classical sense of Skeen [16], which was later refined in [68, 69].

Definition 1 (NBAC [68, 69, 16]). A protocol π is an atomic commit protocol if π is defined by two events:

- *Propose*: $P_i, i = 1, 2, \ldots, n$ proposes value $v = 1$ (vote "yes") or $v = 0$ (vote "no").
- *Decide*: $P_i, i = 1, 2, \ldots, n$ outputs the decided value.

An execution of π solves NBAC if it satisfies the following three properties:[7]

- *Validity*: If a process decides 0, then some process proposes 0 or a failure occurred. If a process decides 1, then no process proposes 0.
- *Termination*: Every correct process eventually decides.
- *Agreement*: No two processes decide differently.

Given a system $\mathscr{S}$ (crash-failure or network-failure), π solves NBAC in $\mathscr{S}$ if every execution of π in $\mathscr{S}$ solves NBAC.

---

[5] Recall that, in an *asynchronous* system, without any communication bound, agreement problems like consensus and NBAC are impossible [49].

[6] For simplicity, we assume hereafter sending messages and processing messages are considered negligible in time. The assumption is equivalent to say that U is an upper bound on time spent on the local computation and synchronous communication.

[7] An execution of an atomic commit protocol also satisfies a property called *integrity*, i.e., no process decides twice in any execution. This is immediate to satisfy in our context so we omit it for presentation simplicity.

(Later in the chapter, in Section 2.5, we will introduce our new variant of the problem: *indulgent atomic commit.*)

A comparison with previous definitions from the literature is now in order. A synchronous NBAC protocol [16, 1] is a protocol which solves NBAC in a crash-failure system (and thus the complexity is covered by our study). In previous impossibility results [68, 83, 84, 75, 76], the definition of validity depended on which failure may occur. *(Strong) validity* stipulates that 1 must be agreed if no crash failure occurs and every process proposes 1, which does not fit the context where no crash failure occurs but network failures can happen. On the contrary, a weak form of validity, *weak validity*, allows processes to abort a transaction (decide 0) in this context (even if all processes propose 1). In this chapter, we do a systematic study where some atomic commit problem is solved so that validity is satisfied in every crash-failure execution and weak validity is satisfied in every network-failure execution. Hence we unify in Definition 1 validity and weak validity for presentation clarity and consistency with previous impossibility results.

2.2.4 Modules

The pseudo code which we use to describe the full protocols in this chapter follows the approach of Cachin et al. [85]. The pseudo code uses “callbacks”: an algorithm is described as a set of event handlers where a process reacts to incoming events by possibly triggering new events.

When presenting optimal protocols, we consider each case in Table 2.1 a different abstraction of the non-blocking atomic commit problem as a set of event handlers. More specifically, each abstraction (an instance of which is denoted by $name$) defines two events (in addition to event <$name$, Init> which performs the initialization of the module once for all): <$name$, Propose | v> and <$name$, Decide | d> where v is the value proposed to the instance $name$ and d is the decision of the instance $name$.

Every optimal protocol is built upon communication channels and a few of them employ a timer. The communication channels are abstracted as a module called *PerfectPointToPointLinks*, denoted by *pl*. The module defines two events: <pl, Send|r, m> and <pl, Deliver|s, m>, where r is the receiver of the sending event, s is the sender of the message delivery event and m represents the message. The timer is abstracted as a module called *Timer*, denoted by *timer*. The module defines two events: <$timer$, Timeout> and **set** timer, where $timer$ timeouts at the time set previously. A timer may be set several times at one process.

Some of our optimal protocols use an underlying consensus module. The module solves *consensus* in a network-failure system [74, 82, 66], which we recall in Definition 2. [8] Many solutions to consensus have been devised, e.g., Paxos and its variants [24, 87], but the correctness of INBAC or the best-case complexity of it does not rely on a particular algorithm among

---

[8] Consensus in this sense is sometimes called *uniform consensus* in the literature [86].

these solutions. The modular approach (using consensus as a service) has been also taken in other distributed algorithms [73, 88].

Definition 2 (Consensus [74])**.** A *consensus* protocol is defined by two events: *propose*, by which a process proposes a value $v = 0$ or 1, and *decide*, which outputs a decision to the process; furthermore, every execution satisfies the following properties: *termination*, *agreement* (similar to those properties of NBAC) and the following *validity* property:

- *Validity*: If a process decides v, then v was proposed by some process.

The indulgent uniform consensus [78] solves uniform consensus (as Definition 2) in every network-failure execution and is abstracted as a module called *IndulgentUniformConsensus*, denoted by *iuc*. The module defines two events: <iuc, Propose | v> and <iuc, Decide | d>, where v is the value proposed to iuc and d is the decision of iuc. We also consider a module called *UniformConsensus*, denoted by *uc*. The module defines two events: <uc, Propose | v> and <uc, Decide | d>, where v is the value proposed to uc and d is the decision of uc. The module solves uniform consensus (as Definition 2) in every crash-failure execution, while it needs only to satisfy a subset of properties of uniform consensus depending on the optimal protocol where the module is employed: if the optimal protocol satisfies property P of Definition 1 in every network-failure execution, then the module also satisfies P of Definition 2 in every network-failure execution.

2.2.5 Complexity measures

We define a *nice execution* of an atomic commit protocol as a failure-free execution in which every process proposes 1. We study in this chapter *best-case complexity*, i.e., the complexity over nice executions (which are arguably the most frequent in practice). We consider two complexity measures: the number of messages and the number of message delays. Here (as in Lamport [57, 89]), for any message m, one message delay is a period of time between two events: the sending of m and the reception of m [57, 89]. Thus if local computation is instantaneous (negligible), and every message is received exactly one unit of time after it was sent, then the number of message delays of an execution is the number of units of time of that execution [57].

2.3 Lower Bounds

In this section, we establish lower bounds on the number of message delays, and then lower bounds on the number of messages. For each lower bound, we prove by contradiction that some messages are necessary in every nice execution and then count the number of these messages. We show that assuming a nice execution E that does not contain some of the necessary messages, we can construct a crash-failure (or network-failure) execution indistinguishable from E that violates a certain property.

2.3.1 Message delays

As shown in Table 2.1, there are two possibilities for the lower bound on the number of message delays: 1 and 2. There are four non-empty cells in Table 2.1 of which the lower bound is 2: (AVT, A), (AVT, AV), (AVT, AT), and (AVT, AVT). Among them, (AVT, A) is the least robust. The rest of the non-empty cells have 1 as the lower bound, among which ($\emptyset$, $\emptyset$) is the least robust. Thus we need only to prove lower bounds for two cells: ($\emptyset$, $\emptyset$), (AVT, A) respectively, as summarized in Theorem 1.

Theorem 1 (Lower bound on message delays)**.** *Let $\mathscr{P}_1$ and $\mathscr{P}_2$ be two subsets of $\mathscr{P} = \{agreement, validity, termination\}$. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathscr{P}_1$ in every crash-failure execution and (c) satisfies $\mathscr{P}_2$ in every network-failure execution. Let d be the smallest number of message delays among all nice executions of π. If for π, $\mathscr{P}_1 = \mathscr{P}_2 = \emptyset$, then $d \geq 1$. If for π, $\mathscr{P}_1 = \mathscr{P}$ and $\mathscr{P}_2 = \{agreement\}$, then $d \geq 2$.*

The proof of the first part of Theorem 1 is immediate: to satisfy validity in every failure-free execution, no process can decide immediately; i.e., the process has to wait for at least one message delay to know other processes' votes.

The proof of the second part is less obvious, and goes through an intermediary lemma. This lemma makes use of the notion of "process reachability", which we introduce here and use in all our lower bound proofs.

Definition 3 (Reaching a process)**.** If a protocol instructs a process src to send a message m to another process $dest$, then we say that src is the *source* of m and $dest$, the *destination* of m. Let E be any execution. In E, if src sends m at time t, then we may interchangeably say that m *leaves* from src (for $dest$) at t; if at time t, $dest$ receives m, then we may interchangeably say that m *arrives* at $dest$ at t.

Let $\underline{m} = \{m_1, m_2, \ldots, m_l\}$ be a sequence of messages such that (a) the source of m_1 is P, (b) the destination of m_l is $Q, Q \neq P$, (c) the source src_i of m_i is the destination of m_{i-1} for $i = 2, 3, \ldots, l$, and (d) m_i leaves from src_i later than or at the time at which m_{i-1} arrives at src_i for $i = 2, 3, \ldots, l$. If $\underline{m}$ exists for two processes P, Q and $l \geq 1$ in E, then we say that P *reaches* Q in E.

If m_l arrives at Q at time t or earlier and $\underline{m}$ is the earliest sequence of messages for P (according to t) to reach Q in E, then we say that P *has reached* Q at time t in E.[9]

By Definition 3, if a process P reaches another process Q, it is possible that, by a sequence of messages, P *backs up* P's vote at Q. The intuition of the lower bound in question, captured by Lemma 1 below, is then that (the arrival of) the messages by which P backups P's vote precede (the departure of) the message after the reception of which P decides.

---

[9] The time t mentioned in Definition 3 is only for convenience of our proof: the time is assumed to be an accurate global clock, but no process necessarily has access to the global clock.

Lemma 1 (Backups)**.** *Let π be any protocol that solves NBAC in every crash-failure execution and ensures agreement in every network-failure execution. Let E be any nice execution of π. Let P decide at time t_1 in E. Among the messages whose destination is P, let $\mathscr{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathscr{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m \in \mathscr{M}} t_m$.*

Then at t_2, P has reached at least f processes.

Proof. By contradiction. Suppose that at t_2, P has reached at most $f-1$ processes. To show a contradiction, we first construct a crash-failure execution E_0 where these $f-1$ processes as well as P (denoted by Φ) crash and every correct process R decides 0. We then construct a network-failure execution E_{async} that is indistinguishable from E to P, and also indistinguishable from E_0 to R; then P and R decide differently in E_{async}, which breaks *agreement*, contradictory to the definition of π.

We first construct E_0. For any process $Q \in \Phi \backslash \{P\}$, denote by τ_Q the time at which P reaches Q in E. In E_0, P crashes at time 0 (before sending any message). For Q, E_0 is the same as E until Q crashes at τ_Q (before possibly notifying P's crash). Let P propose 0, let every process other than P propose 1 and let no process in $\Omega \backslash \Phi$ crash. Then as $|\Phi| \leq f$, E_0 is a legitimate crash-failure execution. Let R be the earliest correct process that decides. Denote by t_3 the time at which R decides. Since π solves NBAC in every crash-failure execution, R decides 0 in E_0.

We then build E_{async} based on E and E_0. In E_{async}, every process proposes 1 and no process crashes. We construct E_{async} such that E_{async} starts as E and:

a. Every message from P to a process in $\Omega \backslash \Phi$ arrives later than $\max(t_1, t_3)$;

b. Every message from Q to a process in $\Omega \backslash \Phi$ sent after or at time τ_Q arrives later than $\max(t_1, t_3)$;

c. Every message sent after t_2 to a process in Φ arrives later than t_1 at the process.

Delays in (a) and (b) ensure that E_{async} is the same as E_0 for R before R decides: any process in Φ seems to have crashed. Delays in (c) ensure that E_{async} and E are indistinguishable for P before P decides: those messages and only those messages in $\mathscr{M}$ arrive for P's decision. □

Lemma 1 additionally shows that for P's vote, at least f backups are necessary. Using Lemma 1, we now prove the necessary number of message delays in Theorem 1.

Proof. (Proof of the second part of Theorem 1.) Let t_2 be defined as in Lemma 1 for the earliest process P that decides in any nice execution. Then for $f \geq 1$, by Lemma 1, at t_2, at least one message from P must have arrived while another message just leaves from its source for P. This, in total, gives at least two message delays before any process decides. □

2.3.2 Messages

As shown in Table 2.1, there are four possibilities for the lower bound on the number of messages: 0, $n-1+f$, $2n-2$ and $2n-2+f$. We group the cells in Table 2.1 with the same value, and then prove the lower bound for the least robust atomic commit in each group. Thus we need only to prove lower bounds for four cells in Table 2.1: ($\emptyset$, $\emptyset$), (V, $\emptyset$), (V, V), and (AVT, A) respectively, as summarized in Theorem 2. While proving our lower bounds, we highlight the intuition behind the increasing lower bounds (from 0 to $2n-2+f$), and a tradeoff between time and message complexity (for the 14 variants of the atomic commit problem that have $n-1+f$ messages or $2n-2$ messages as lower bounds).

Theorem 2 (Lower bounds on messages)**.** *Let $\mathscr{P}_1$ and $\mathscr{P}_2$ be two subsets of $\mathscr{P} = \{$agreement, validity, termination$\}$. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathscr{P}_1$ in every crash-failure execution and (c) satisfies $\mathscr{P}_2$ in every network-failure execution. Let m be the smallest number of messages among all nice executions of π. If for π, $\mathscr{P}_1 = \mathscr{P}_2 = \emptyset$, then $m \geq 0$. If for π, $\mathscr{P}_1 = \{$validity$\}$ and $\mathscr{P}_2 = \emptyset$, then $m \geq n-1+f$. If for π, $\mathscr{P}_1 = \mathscr{P}_2 = \{$validity$\}$, then $m \geq 2n-2$. If for π, $\mathscr{P}_1 = \mathscr{P}$ and $\mathscr{P}_2 = \{$agreement$\}$, then $m \geq 2n-2+f$.*

The proof of 0 message for the cell ($\emptyset$, $\emptyset$) is trivial and omitted. In what follows, we count the number of necessary messages in the other three cases separately.

Lower bound of $n-1+f$ messages. We generalize here the lower bound of $2n-2$ messages for synchronous NBAC from Dwork and Skeen [1]. As in their proof, we first present a preliminary lemma, Lemma 2, which we phrase here in terms of "process reachability". As Lemma 2 is a (straightforward) generalization of the preliminary lemma in Dwork and Skeen's proof, the proof of Lemma 2 is omitted.[10]

Lemma 2 (Validity despite crashes)**.** *Let π be any protocol that (a) solves NBAC in every failure-free execution and (b) ensures validity in every crash-failure execution. Then in any nice execution of π, every process reaches at least f processes.*

Lemma 2 captures the intuition that at least f backups are necessary in the face of at most f crashes. This leads to $n-1+f$ messages as the lower bound for cell (V, $\emptyset$): by Lemma 2, every process has to reach at least f processes in every nice execution, and thus at least $n-1+f$ messages have to be exchanged.

---

[10] We note, however, that the original preliminary lemma in [1] does not distinguish between the necessity of sending a message (before a certain point in time) and the necessity of receiving a message (before a certain point in time). It is thus only appropriate for a crash-failure execution (as after a message m is sent, it is predictable that m is received within some time period) and does not apply, as is, to the setting of a network-failure execution as we study in this chapter. Hence the need to rephrase the preliminary lemma.

Lower bound of $2n-2$ messages. Before counting the number of necessary messages for cell (V, V), we introduce a preliminary lemma.

Lemma 3 (Validity in every execution)**.** *Let π be any protocol that (a) solves NBAC in every failure-free execution and (b) ensures validity in every network-failure execution. Then in every nice execution of π, for any process Q, every other process P reaches Q before or when Q decides.*

Proof. By contradiction. Consider a nice execution E with two processes P and Q such that P has not reached Q when Q decides 1. In E, let Q decide at time t; let Φ be the set of processes which P has reached before or at t; for every $R \in \Phi$, let τ_R be the time at which P reaches R. To show a contradiction, we construct a network-failure execution E_{async} such that P crashes before sending any message and P votes 0, but for Q, E_{async} is indistinguishable from E (where Q decides 1). In E_{async}, every process (except P) votes 1; for them, E_{async} starts as E. In addition, for every $R \in \Phi$, every message from R sent at or after τ_R arrives later than t. Since in E, Q does not expect any message from R sent at or after τ_R and Q does not expect any message from P either, then Q does not distinguish E and E_{async} and thus decides 1 at t again in E_{async}, which violates validity. □

By Lemma 3, now every process P must know every vote *explicitly*, while in Lemma 2, some process Q's vote of 1 may be *implicit* (i.e., in a nice execution, P knows Q's vote of 1 by not receiving a certain message). The requirement of explicit votes clearly adds extra messages, due to the validity satisfied in every network-failure execution. For cell (V, V), we count the number of necessary messages as follows. Let R be the latest process that decides in a nice execution. By Lemma 3, before or when R decides, for any process Q, every process $P \neq Q$ has reached Q. As a result, before or when R decides, at least $2n-2$ messages are exchanged.

We note that for atomic commit problems with $n-1+f$ messages and $2n-2$ messages as lower bounds, the lower bound on the number of message delays is 1. It is easy to show that the lower bound on the number of messages and that on the number of message delays cannot achieved at the same time: all those problems feature *validity* at least in every crash-failure execution and thus a 1-delay protocol must use at least $n(n-1)$ messages. This shows that for those problems (14 cases among totally 27 ones which we consider), there is a tradeoff between the number of messages and that of message delays. (Later in Section 5, we show tradeoffs between time and message complexity for other 4 cases related to indulgent atomic commit.)

Lower bound of $2n-2+f$ messages. Before counting the number of necessary messages for cell (AVT, A), we again introduce a preliminary lemma.

Lemma 4 (Agreement in every execution)**.** *Assume $f \geq 2$. Let π be any protocol that solves NBAC in every crash-failure execution and ensures agreement in every network-failure execution. Let E be any nice execution. Let P decide at time t_1 in E. Among the messages whose destination* *is P, let $\mathcal{M}_P$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathcal{M}_P$, let $t_{m,P}$ be the time at which m leaves from its source and $t_{2,P} = \max_{m\in\mathcal{M}_P} t_{m,P}$.*

Then at $t_{2,P}$ in E, every process has reached at least $f-1$ processes.

Let Π denote the class of protocols considered in Lemma 4 above. This is the same class of protocols considered in Lemma 1. The proof of Lemma 4 is actually similar to that of Lemma 1, and thus omitted.

By the robust relation, Π is incomparable with the class of protocols considered in Lemma 3 (with $2n-2$ messages as the lower bound) but is more robust than the class of protocols considered in Lemma 2 (with $n-1+f$ messages as the lower bound). We highlight the increase from $n-1+f$ messages to the lower bound of Π due to the improvement of robustness. To do so, we actually compare Lemma 1 (which considers also Π) with Lemma 2. Although both lemmas show that P backups at (at least) f processes, Lemma 1 demonstrates that for Π, after P backups, it is necessary for some message to leave for P, which increases the number of necessary messages.

We use Lemma 4 to count the exact number of necessary messages for cell (AVT, A). Let $t_{2,P}$ be defined as in the statement of Lemma 4 for any process P in any execution. Let $t_2 = \min_{P\in\Omega} t_{2,P}$. Then at and after t_2, at least n messages have to leave their sources respectively. Since at t_2, every process has reached at least $f-1$ processes, then before or at t_2, at least $n-2+f$ messages have arrived at their destinations respectively. Therefore, at least $2n-2+f$ messages are exchanged in every nice execution.

2.4 Matching Protocols

In this section, we prove the tightness of the lower bounds by presenting matching commit protocols. For each protocol, we describe first its nice executions, and then sketch the executions that deviate from nice executions due to some vote of 0 or failure. We include full protocols and their proofs of correctness for completeness. We present matching protocols for the number of message delays and the number of messages separately.

2.4.1 Delay-optimal protocols

Recall that in Table 2.1, there are two possibilities for the lower bound on the number of message delays: 1 and 2. Recall also that there are four cells in Table 2.1 of which the lower bound is 2: (AVT, A), (AVT, AV), (AVT, AT), and (AVT, AVT), among which the last one is the most robust. The rest of the non-empty cells correspond to a lower bound of 1 delay, among which (AV, AV), (AT, AT) and (AVT, VT) are three local maximum by the relation of robustness. Thus we need only to present delay-optimal protocols for four cells, as summarized in Table 2.2 as well as in Theorem 3.

Table 2.2 – Delay-optimal Protocols. 1NBAC is a synchronous NBAC protocol. Each protocol achieves its lower bound in every nice execution.

Cell	AV, AV	AT, AT	AVT, VT	AVT, AVT
Protocol	avmNBAC	0NBAC	1NBAC	INBAC

Theorem 3 (Delay-optimal protocols)**.** *Let* $\mathscr{P}_1$ *and* $\mathscr{P}_2$ *be any two subsets of* $\mathscr{P} = \{$*agreement, validity, termination*$\}$*. Let* π *be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies* $\mathscr{P}_1$ *in every crash-failure execution and (c) satisfies* $\mathscr{P}_2$ *in every network-failure execution. Let* d *be the smallest number of message delays among all nice executions of* π*. If* $d = 1$*, then it is possible that* $\mathscr{P}_1 = \mathscr{P}_2 = \{$*agreement, validity*$\}$*, or* $\mathscr{P}_1 = \mathscr{P}_2 = \{$*agreement, termination*$\}$*, or* $\mathscr{P}_1 = \mathscr{P}$ *and* $\mathscr{P}_2 = \{$*validity, termination*$\}$*. If* $d = 2$*, then it is possible that* $\mathscr{P}_1 = \mathscr{P}_2 = \mathscr{P}$.

Among the protocols of Table 2.2, INBAC solves what we call *indulgent atomic commit*, which we will discuss in Section 2.5 separately. 0NBAC is an optimal protocol also for the number of messages, which we will discuss with other message-optimal protocols. For avmNBAC is similar to 1NBAC, we only sketch the former.

Protocol sketches

1NBAC. During a failure-free execution of 1NBAC, a process (a) sends its vote to every process, (b) collects all n votes, (c) sends the logical AND of all n votes to every process, and then (d) decides. Thus in every failure-free execution (as well as in every nice execution), every process decides the logical AND of all n votes within one message delay.

In other executions, every process starts by sending its vote to every (other) process, but then since failures may occur, a process P may collect fewer than n votes at the end of the first message delay. If so, P waits for the logical AND of all n votes sent by another process for one message delay. (This is the key to agreement in any crash-failure execution, since in a crash-failure execution, if some process Q decides at step (d), then Q's messages sent at step (c) must arrive at their receivers in one message delay.) Denoted by [D, d] a message that contains the logical AND, d, of all n votes. If P receives any [D, d] before or at the end of the second message delay, then P proposes d to consensus uc; otherwise, P proposes 0 to uc. (Recall the definition of consensus in Section 2.2.) Then P decides the same as uc.

avmNBAC. As 1NBAC, avmNBAC starts by having every process send its vote to every other process. Unlike 1NBAC, avmNBAC does not require termination if a failure occurs; thus every process decides if and only if it collects all the votes at the end of the first message delay. The full description of avmNBAC, which is similar to that of 1NBAC, is omitted.

Full protocol

1NBAC.

The pseudo code of the full protocol here (as well as all the other protocols described in this chapter) uses the following assumptions and notations if not explicitly stated otherwise. (a) We assume that every process knows its own ID stored in the local variable i of that process. (b) We assume that a message delivery event has a higher priority than a timeout event; i.e., if both events occur at a process at the same time, the process is first triggered by the delivery event and then the timeout event. (c) Sometimes a process is triggered by both the delivery of some message m and a logical condition ℓ; we assume that if m arrives earlier than when ℓ is satisfied, then m (as well as the delivery of m) is queued to wait for the satisfaction of ℓ. (d) If a protocol is designed to satisfy some properties in every crash-failure execution, then we use timers in the protocol and assume that one unit at the timer at every process is set to the known upper bound of the message delay of the given crash-failure system. (Clearly, in a network-failure execution of the protocol, message delays might violate the upper bound, and as a result, although the timer timeouts, a process does not necessarily receive the message which it sets the timer to wait for.) (e) The timer starts at time 0 when every process proposes its value (if we do not say otherwise explicitly).

Here we present 1NBAC that (a) solves NBAC in every crash-failure execution, (b) satisfies validity and termination in every network-failure execution and (c) decides in one message delay in every nice execution. Algorithm 1 presents the full protocol.

Proof. (Proof of correctness of 1NBAC.)

Termination. Every correct process proposes a value and sets a timer when $phase = 0$. When the timer timeouts, every correct process either decides, or sets again the timer and assigns $phase = 1$. When the timer timeouts again, the correct process proposes a value to uc. Thus, by the *termination* property of consensus, every correct process decides.

Commit-Validity. If process P decides 1, then by the *validity* property of consensus and the protocol itself, there exists process Q (not necessarily P) who sends [D, 1] in phase 0 and therefore every process proposes 1. Thus, the *commit-validity* property is satisfied.

Abort-Validity. If process P decides 0, then either some process P decides 0 in phase 0, which implies that some process proposes 0, or by the *validity* property of consensus, some process Q proposes 0 to uc in phase 1, which implies that some process proposes 0 or Q receives fewer than n messages in phase 0. The latter shows that a failure occurs. Thus, the *abort-validity* property is satisfied.

Algorithm 1 Algorithm 1NBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.
  UniformConsensus, instance uc.

upon event <1nbac, Init> do
  phase := 0;
  proposed := FALSE;
  decided := FALSE;
  decision := ⊥;
  collection0 := ∅;
  collection1 := ∅;

upon event <1nbac, Propose | v> do
  decision := v;
  forall q ∈ Ω do
    trigger <pl, Send | q, [V, v]>;
  set timer to 1;

upon event  <pl, Deliver | p, [V, v]> do
  collection0 := collection0 ∪ {p};
  decision := decision AND v;

upon event <timer, Timeout> and phase = 0 do
  if collection0 = Ω then
    forall q ∈ Ω do
      trigger <pl, Send | q, [D, decision]>;
    if not decided then
      decided := TRUE;
      trigger <1nbac, Decide | decision>;
  else
    phase := 1;
    set timer to 2;

upon event <pl, Deliver | p, [D, d]> do
  collection1 := collection1 ∪ {p};
  decision := d;

upon event <timer, Timeout> and phase = 1 do
  if not decided then
    if collection1 = ∅ then
      decision := 0;
    proposed := TRUE;
    trigger <uc, Propose | decision>;
```

```
upon event <uc, Decide | d> do
  if not decided then
    decided := TRUE;
    trigger <1nbac, Decide | d>;
```

Table 2.3 – Message-optimal Protocols. Protocol (n-1+f)NBAC is a synchronous NBAC protocol. Each protocol achieves its lower bound in every nice execution.

Cell	AT, AT	AVT, T	AV, A	AVT, VT	AV, AV	AVT, AVT
Protocol	0NBAC	(n-1+f)NBAC	aNBAC	(2n-2)NBAC	2PC	(2n-2+f)NBAC

Agreement. By contradiction. Suppose that two different processes P and Q decide 1 and 0 respectively, in a crash-failure execution. Then by the *commit-validity* property and the *abort-validity* property, every process proposes 1 and some process crashes before Q decides. By the *agreement* property of consensus in a crash-failure execution, P and Q cannot both follow the decision of uc to decide. Thus P decides 1 in phase 0 and Q decides 0 as a decision of uc.

Since P decides in phase 0, P succeeds in sending $[D, 1]$ to every other process. Moreover, since every process proposes 1 to $1nbac$, no process sends $[D, 0]$ after the first message delay. Thus thanks to the synchronous communication, every process that has not decided yet receives $[D, 1]$ and proposes 1 to uc. Thus by the *validity* property of consensus, Q cannot decide 0 as a decision of uc. A contradiction. □

2.4.2 Message-optimal protocols

As shown in Table 2.1, there are four lower bounds on the number of message delays: 0, $n-1+f$, $2n-2$, and $2n-2+f$. Similarly, we group the cells of which the lower bound takes the same value in Table 2.1, and find the most robust one or the local maximum in each group. Thus we need only to present message-optimal protocols for six cells, as summarized in Table 2.3 as well as in Theorem 4. Among these cells, cell (AV, AV) has $2n-2$ as a lower bound on the number of messages and hence the classical protocol, 2PC, is a matching protocol, for which we do not need to propose a new one.

Theorem 4 (Message-optimal protocols)**.** *Let $\mathscr{P}_1$ and $\mathscr{P}_2$ be any two subsets of $\mathscr{P}$ = {agreement, validity, termination}. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathscr{P}_1$ in every crash-failure execution and (c) satisfies $\mathscr{P}_2$ in every network-failure execution. Let m be the smallest number of messages among all nice executions of π. If $m = 0$, then it is possible that $\mathscr{P}_1 = \mathscr{P}_2$ = {agreement, termination}. If $m = n-1+f$, then it is possible that $\mathscr{P}_1$ = {agreement, validity} and $\mathscr{P}_2$ = {agreement}, or $\mathscr{P}_1 = \mathscr{P}$ and $\mathscr{P}_2$ = {termination}. If $m = 2n-2$, then it is possible that $\mathscr{P}_1 = \mathscr{P}_2$ = {agreement, validity}, or $\mathscr{P}_1 = \mathscr{P}$ and $\mathscr{P}_2$ = {validity, termination}. If $m = 2n-2+f$, then it is possible that $\mathscr{P}_1 = \mathscr{P}_2 = \mathscr{P}$.*

Below we sketch only 0NBAC, (n-1+f)-NBAC, and (2n-2)NBAC, since aNBAC and (2n-2+f)NBAC are close to (n-1+f)NBAC. The full descriptions of the protocols (and their full proofs of correctness) cover all message-optimal protocols (except for 2PC). As shown in the sketches below of the three protocols, 0NBAC, (n-1+f)NBAC, and (2n-2)NBAC, the primary technique to achieve an optimal number of messages is to support nice executions by complex executions that abort; namely, processes take complex steps before a decision of 0: they try to inform every other of this decision.

Protocol sketches

0NBAC. During every nice execution, no process sends a message, and after one message delay, a process (who votes 1) decides 1 if it has received no message. In other executions, a process who votes 1 still sends no message at the beginning, while a process who votes 0 sends [V, 0] to every (other) process. Then after one message delay, n processes are divided into three categories: (1) those who vote 0, (2) those who vote 1 and receive [V, 0], and (3) those who vote 1 but do not receive any message. The last category decides 1 again immediately, while the other two later propose a value to consensus iuc (and decide the same as iuc). The second category now sends [B, 0] to every other process. Any receiver of [*, 0] who has not decided yet acknowledges to the sender. If a process in category (1) or (2) receives $n-1$ acknowledgements, then it proposes 0 to iuc, and otherwise, 1. Clearly, both categories (1) and (2) may potentially decide 0 and thus they try to inform the others of this decision. The key to agreement here is to agree with the last category which may have already decided 1 (at the end of the first message delay). However, since by the protocol, the third category does not acknowledge to [*, 0], if the third category is non-empty, then all other processes must propose 1 to iuc and decide 1, satisfying agreement.

For best-case complexity, it is easy to see that in every nice execution, no message is ever sent, and furthermore, every process decides after one message delay. 0NBAC achieves the lower bound on the number of messages and that on the number of message delays at the same time. As a result, for the 9 cases (among 27 cases) covered by this protocol (using the robustness relation), no tradeoff is necessary.

(n-1+f)NBAC. During every nice execution of this protocol, the communication steps among processes are totally ordered. The totally-ordered sequence is: P_1, $P_2, \ldots, P_n$ and subsequently $P_1, P_2, \ldots, P_f$. Then (a) P_1 starts by sending P_1's vote to P_2; (b) each process in the sequence, upon receiving its predecessor's message, sends the collection of the votes so far to its successor except P_f which is at the end of the sequence; (c) (after $n-1+f$ steps above) every process waits (i.e., does no-ops) for $f+1$ message delays; and (d) during step (c), a process does not receive any message and thus decides 1.

In other executions, for any process P, if P votes 0, then P sends no message to the successor (when P first occurs in the sequence). If P does not receive its predecessor's message, then P

sends no message to its successor as well except that P is in the suffix $P_n, P_1, P_2, \ldots, P_f$. In the suffix, if P does not receive its predecessor's message or receives 0 from its predecessor, then P sends 0 to every other process. Subsequently, if any process receives a message of 0, then the process sends 0 as well to every other process. Every process decides at the same time as in a nice execution (i.e., step (d) in a nice execution). At the end, if a process has ever received a message of 0, then it decides 0 (and 1 otherwise).

The number of messages in any nice execution is thus $n-1+f$, matching the lower bound. To match the lower bound, in any nice execution, some process P decides 1 without being reached by every process: some votes of 1 are only implicit to P. In (n-1+f)NBAC, the decision at step (d) ensures that those who accept implicit votes (as votes of 1) can be notified of a decision of 0 in the face of at most f crashes in any crash-failure execution.

(2n-2)NBAC. During every nice execution, (a) every process sends its vote to P_n spontaneously, (b) then P_n sends the logical AND of all n votes to every process, and (c) every process waits for $f+1$ message delays, and then decides 1. When a failure occurs or some process votes 0, at step (b), P_n sends 0 to every process. Then at step (c), a process can receive no message from P_n or a message of 0 from P_n. If so, the process sends 0 to every process. Later, any process who receives a message of 0 also sends 0 to every process. Every process decides at the same time as in a nice execution (i.e., the end of step (c) in a nice execution). At the end, if a process has ever received a message of 0, it decides 0 (and 1 otherwise).

The number of messages in any nice execution is thus $2n-2$. Similar to (n-1+f)NBAC, here any process who decides 0 tries to inform every other process before the decision, while the decision at the end of step (c) ensures that at least one process succeeds in notifying every correct process of the potential decision of 0 in every crash-failure execution, to satisfy agreement.

Full protocols

0NBAC. Here we present our 0NBAC protocol in Algorithm 2. For 0NBAC, every failure-free execution solves NBAC, every network-failure execution satisfies agreement and termination, and n processes exchange 0 message in every nice execution.

Proof. (Proof of correctness of 0NBAC.)

Termination. Every correct process P proposes a vote v and sets $timer$ to 1. Then when $timer$ first timeouts, P either decides, or again sets $timer$. At the second timeout of $timer$, every correct process (which has not yet decided) proposes to iuc, which eventually decides by the *termination* property of iuc in a network-failure execution.

Algorithm 2 Algorithm 0NBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  UniformConsensus, instance iuc.
  Timer, instance timer.

upon event <0nbac, Init> do
  myvote := ⊥;
  myack := ∅;
  decided := FALSE;
  zero := FALSE;
  phase := 0;

upon event <0nbac, Propose | v> do
  myvote := v;
  if v = 0 then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, [V, 0]>;
  set timer to time 1;
  phase := 1;

upon event <pl, Deliver | p, [V, v]> and phase = 1 do
  zero := TRUE;
  trigger <pl, Send | p, [ACK]>;

upon event <pl, Deliver | p, [B, b]> and phase = 2 do
  if not (myvote = 1 and decided) then
    trigger <pl, Send | p, [ACK]>;

upon event <pl, Deliver | p, [ACK]> do
  myack := myack ∪ {p};

upon event <timer, Timeout> and phase = 1 do
  phase = 2;
  if zero = FALSE and myvote = 1 then
    decided := TRUE;
    trigger <0nbac, Decide | 1>;
  else if zero = TRUE and myvote = 1 then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, [B, 0]>;
    set timer to time 3;
  else
    set timer to time 2;
```

```
upon event <timer, Timeout> and phase = 2 do
  if myack ⊂ Ω then
    trigger <iuc, Propose | 1>;
  else
    trigger <iuc, Propose | 0>;

upon event <iuc, Decide | d> and not decided do
  trigger <0nbac, Decide | d>;
  decided := TRUE;
```

Commit-Validity. We only need to prove validity in every failure-free execution. If in a failure-free execution, a process P decides 1, then either P decides 1 at the first timeout or P decides the decision of consensus iuc. If P decides at the first timeout, then P does not receive any message [V, 0], which implies that every process proposes 1. If P decides the decision of iuc, then some process Q proposes 1 at Q's second timeout. Either Q's vote is 0 or Q receives a message [V, 0] before the first timeout. In either case, message [V, 0] is sent to all process when the local variable $phase$ at every process is 1. Then in a failure-free execution, no process decides at the first timeout. However, for Q to propose 1, again in a failure-free execution, there must be some process R such that R's vote is 1 and R has decided at the first timeout, which leads to a contradiction. Therefore P cannot decide the decision of iuc in a failure-free execution. As a result, P can only decide at the first timeout and every process proposes 1, which satisfies *commit-validity*.

Abort-Validity. We only need to prove validity in every failure-free execution. If a process P decides 0, then some process Q has proposed 0 to iuc. Then Q has votes 0 or has received a vote of 0, which satisfies the *abort-validity* property.

Agreement. By contradiction. Suppose that E is a network-failure execution in which two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. By the *agreement* property of consensus, Q's decision must be a decision of iuc while P's decision is not. Then some process R must have proposed 0 to iuc. As a result, R has $timer$ timeout twice at itself. Suppose that the vote of R to the commit protocol is 0. Then the second timeout is at time 2. We argue that R cannot receive P's acknowledgement of R's message $[V, 0]$ at the second timeout. If R can, then P's local variable $zero$ turns true before P's first timeout, which contradicts to P's decision of 1. Thus, the vote of R to the commit protocol can only be 1, and R's second timeout is at time 3. Similarly, we argue that R cannot receive P's acknowledgement of R's message $[B, 0]$ at the second timeout. If R can, then P's local variables $phase = 2$ and $decided$ = FALSE must hold when P sends the acknowledgment, which again contradicts to P's decision of 1 (without invoking iuc). □

Message-optimal protocol for synchronous NBAC: (n-1+f)NBAC. We present the full protocol in Algorithm 3. Hereafter we use the following notation convention: symbol $\%n$ represents modulo n except that if the remainder is 0, the result of % is n instead of 0. The terminology of the timer is slightly different from the other protocols: the timer here starts at time 1 when the first sending event happens.

Proof. (Proof of correctness of (n-1+f)NBAC.)

Termination. When a process proposes a value or its local $timer$ timeouts, it assigns a value to $phase$. Each time a process assigns a value to $phase$, it sets a timer. Since every correct process proposes a value, then every correct process enters phase 3 and has $timer$ timeout at $n+2f+1$. Every correct process decides at $n+2f+1$.

Commit-Validity. We only need to prove validity in every crash-failure execution. If process P decides 1, then at time $n+2f+1$, P's local variable $decision = 1$. This leads to three facts: (a) that P has received no 0 from other processes; (b) that P's local variable $delivered$ is TRUE when the timeout event for phase 1 (and if P is among $P_1, P_2, \ldots, P_f$, P's local variable $delivered$ is TRUE when the timeout event for phase 2 occur at P); and (c) that P proposes 1. If P is among $P_1, P_2, \ldots, P_f$, then according to (b), P has received 1 at phase 2, which implies that every process proposes 1. If P is P_n, then according to (b), P has received 1 at phase 1, which implies that every process proposes 1. If P is not among $P_1, P_2, \ldots, P_f, P_n$, then according to (a), P does not receive 0 from $P_n, P_1, \ldots, P_f$ at time $n+1, \ldots, n+f+1$ respectively. Since at most f processes can crash, one process Q among $P_n, P_1, \ldots, P_f$ is instructed by the protocol to not send 0 to P. This implies that Q has received 1 at phase 2 if Q is among $P_1, P_2, \ldots, P_f$ or Q has received 1 at phase 1 if Q is P_n. Therefore, every process proposes 1.

Abort-Validity. We only need to prove validity in every crash-failure execution. If process P decides 0, then P's local variable $decision = 0$. Then either (a) P has proposed $v = 0$, or (b) P has received 0 from other processes, or (c) P's local variable $delivered$ is FALSE when the timeout event for phase 1 or the timeout event for phase 2 occurs at P.

If P receives 0 from another process Q, then since a process only sends its local variable $decision$ to other processes (if it sends any message), w.l.o.g., we may assume that Q is the earliest process that has local variable $decision = 0$. As a result, either Q has proposed $v = 0$, or Q's local variable $delivered$ is FALSE when the timeout event for phase 1 or the timeout event for phase 2 occurs at Q.

Then, to examine the *abort-validity* property, we need only to examine the case where $delivered$ is FALSE for Q, and case (c) for P. Let P_i be either process. As $delivered$ is FALSE, P_i does not receive any message from $P_{(i-1)\%n}$ before the timeout event for phase

Algorithm 3 (n-1+f)NBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.

upon event <nbac, Init> do
  decision := ⊥;
  decided := FALSE;
  delivered := FALSE;
  phase := 0;

upon event <nbac, Propose | v> do
  decision := v;
  if $i = 1$ then
    trigger <pl, Send | $P_2$, decision>;
  if $i = 1$ then
    set timer to time $n + 1$;
    phase := 2;
  else
    set timer to time $i$;
    phase := 1;

upon event <pl, Deliver | p, v> do
  decision := decision AND v;
  if phase ≤ 2 then
    if $p = P_{(i-1)\%n}$ then
      delivered := TRUE;
  else if not decided then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, decision>;

upon event <timer, Timeout> and phase = 1 do
  if delivered = FALSE then
    decision := 0;
  if decision = 1 then
    trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
  else if $i = n$ then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, decision>;
  delivered := FALSE;
  if $i \geq f + 1$ then
    set timer to time $n + 2f + 1$;
    phase := 3;
  else
    set timer to time $n + i$;
    phase := 2;
```

```
upon event <timer, Timeout> and phase = 2 do
  if delivered = FALSE then
    decision := 0;
  if decision = 1 and $i \neq f$ then
    trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
  if decision = 0 then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, decision>;
  delivered := FALSE;
  set timer to time $n+2f+1$;
  phase := 3;

upon event <timer, Timeout> and phase = 3 do
  decided := TRUE;
  trigger <nbac, Decide | decision>;
```

1 or the timeout event for phase 2 occurs. At the same time, for $2 \leq i \leq n$, $P_{(i-1)\%n}$ is instructed by the protocol to send a message to $P_{i\%n}$ in phase 1 if $P_1, P_2, \ldots, P_{i-2}$ do not crash and $P_1, P_2, \ldots, P_{i-1}$ propose 1; for $i = 1$, $P_{(i-1)\%n}$ is instructed by the protocol to send a message to $P_{i\%n}$ in phase 2; and for $2 \leq i \leq f$, $P_{(i-1)\%n}$ is instructed by the protocol to send a message to $P_{i\%n}$ in phase 2. As $delivered$ is FALSE, then some process crashes or some process proposes 0.

In conclusion, the *abort-validity* property is satisfied.

Agreement. By contradiction. Suppose that two different processes P and Q decide 1 and 0 respectively in a crash-failure execution. Then by the *commit-validity* property and the *abort-validity* property, every process proposes 1 and some process crashes before Q decides.

Since Q decides 0, then Q's local variable $decision$ is assigned to 0 at some point in phase 1, phase 2, or phase 3. Suppose that Q assigns $decision$ to 0 in phase 1. If $Q \neq P_n$, then Q refuses to send a message, which would lead P to decide 0; if $Q = P_n$, then Q sends 0 to P, which would also lead P to decide 0. A contradiction. Suppose that Q assigns $decision$ to 0 in phase 2, then Q also sends 0 to P, which would again lead P to decide 0. A contradiction.

Suppose that Q assigns $decision$ to 0 in phase 3, then Q only does the assignment at time $n+2f+1$ or later. Otherwise, since both P and Q are alive at $n+2f+1$, when Q sends $decision$ to P after the assignment, then the network could schedule the message so that P receives 0 before time $n+2f+1$ and decide 0 at time $n+2f+1$. Now that Q does the assignment at time $n+2f+1$, some process must send 0 to Q at time $n+2f$ or later. In fact, in order for Q to receive 0, between time $n+f$ and time $n+2f$, there must be at least $f+1$ process that try to send 0 to every process. However, those processes all fail to send 0 to P. This gives a contradiction: $f+1$ processes must have crashed (to make all those attempts fail) while at most f processes may crash. □

aNBAC. The full protocol is presented in Algorithm 4. Similar to (n-1+f)NBAC, the timer is slightly changed: the timer here starts at time 1 when the first sending event happens. Two timers are used in the algorithm and identified by different names.

Proof. (Proof of correctness of aNBAC.)

Termination. We only need to prove that a process decides in a failure-free execution. Clearly, a process proposes a vote before or at time 1. If every process proposes 1, then every process eventually timeouts at time $n+2f+1$, and $noop$ is never assigned to TRUE. In a failure-free execution, every process has their local variable $delivered$ to be TRUE at their timeout. As a result, at time $n+2f+1$, every process decides 1. Otherwise, if some process votes 0, then every process who votes 0 timeouts at time 3 and decides while every process who votes 1 timeouts eventually at time 4 and also decides, Thus every process decides in a failure-free execution.

Commit-Validity. We only need to prove validity in every crash-failure execution. If process P decides 1, then P decides at time $n+2f+1$ when P's local variable $decision = 1$ and $noop$ is FALSE. Since $decision = 1$, this leads to three facts: (a) that P has received no 0 from other processes; (b) that P's local variable $delivered$ is TRUE when the timeout event for phases 1 or 2 occurs at P; and (c) that P proposes 1. If P is P_n, then according to (b), P has received 1 at phase 1, which means that the logical AND of all votes is 1 and every process proposes 1. If P is among $P_1, P_2, \ldots, P_f$, then according to (b), P has received 1 at phase 2, which implies that every process proposes 1. If P is not among $P_1, P_2, \ldots, P_f, P_n$, then according to (a), P does not receive 0 from $P_n, P_1, \ldots, P_f$ at time $n+1, \ldots, n+f+1$ respectively. Since at most f processes can crash, at least one process Q among $P_n, P_1, \ldots, P_f$ is instructed by the protocol to not send 0 to P. This implies that Q has received 1 at phase 2 if Q is among $P_1, P_2, \ldots, P_f$ or Q has received 1 at phase 1 if Q is P_n. Therefore, every process proposes 1.

Abort-Validity. We only need to prove validity in every crash-failure execution. If process P decides 0, then P only decides 0 at time 3 or at time 4. Then P either has voted 0, or has received a vote of 0 from some other process. Therefore, the abort-validity property is satisfied.

Agreement. By contradiction. Suppose that two different processes P and Q decide 1 and 0 respectively, in a network-failure execution. Then P decides at time $n+2f+1$ and Q decides at time 3 or at time 4.

Algorithm 4 aNBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.
  Timer, instance timer0.

upon event <anbac, Init> do
  decision := ⊥;
  decided := FALSE;
  delivered := FALSE;
  phase := 0;
  vote := ⊥;
  delivered_V := FALSE;
  collection_V := ∅;
  collection_B := ∅;
  noop := FALSE;
  phase0 := 0;

upon event <anbac, Propose | v> do
  decision := v;
  vote := v;
  if i = 1 then
    trigger <pl, Send | $P_2$, decision>;
  if i = 1 then
    set timer to time $n + 1$;
    phase := 2;
  else
    set timer to time i;
    phase := 1;
  if v = 0 then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, [V, 0]>;
    set timer0 to time 3;
  else
    set timer0 to time 2;

upon event <pl, Deliver | p, [V, 0]> do
  decision := 0;
  delivered_V := TRUE;
  trigger <pl, Send | p, [ACK, V]>;

upon event <pl, Deliver | p, [B, 0]> do
  decision := 0;
  trigger <pl, Send | p, [ACK, B]>;
```

```
upon event <timer0, Timeout> and vote = 1 and delivered_V and phase0 = 0 do
  forall $q \in \Omega$ do
    trigger <pl, Send | q, [B, 0]>;
  set timer0 to time 4;
  phase0 := 1;

upon event <pl, Deliver | p, [ACK, V]> do
  collection_V := collection_V $\cup$ {p};

upon event <pl, Deliver | p, [ACK, B]> do
  collection_B := collection_B $\cup$ {p};

upon event <timer0, Timeout> and vote = 0 do
  if collection_V = $\Omega$ and decided = FALSE then
    decided := TRUE;
    trigger <anbac, Decide | 0>;
  else
    noop := TRUE;

upon event <timer0, Timeout> and vote = 1 and delivered_V and phase0 = 1 do
  if collection_B = $\Omega$ and decided = FALSE then
    decided := TRUE;
    trigger <anbac, Decide | 0>;
  else
    noop := TRUE;

upon event <pl, Deliver | p, v> do
  decision := decision AND v;
  if phase $\leq$ 2 then
    if $p = P_{(i-1)\%n}$ then
      delivered := TRUE;
  else if not decided then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, decision>;

upon event <timer, Timeout> and phase = 1 do
  if delivered = FALSE then
    decision := 0;
  if decision = 1 then
    trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
  else if $i = n$ then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, decision>;
  delivered := FALSE;
  if $i \geq f + 1$ then
    set timer to time $n + 2f + 1$;
    phase := 3;
```

```
  else
    set $timer$ to time $n+i$;
    $phase$ := 2;

upon event <$timer$, $Timeout$> and $phase = 2$ do
  if $delivered$ = FALSE then
    $decision$ := 0;
  if $decision = 1$ and $i \neq f$ then
    trigger <$pl$, $Send$ | $P_{(i+1)\%n}$, $decision$>;
  if $decision = 0$ then
    forall $q \in \Omega$ do
      trigger <$pl$, $Send$ | $q$, $decision$>;
  $delivered$ := FALSE;
  set $timer$ to time $n+2f+1$;
  $phase$ := 3;

upon event <$timer$, $Timeout$> and $phase = 3$ and not $decided$ do
  if $decision = 1$ and not $noop$ then
    $decided$ := TRUE;
    trigger <$anbac$, $Decide$ | $decision$>;
```

When Q decides at time t ($t = 3$ or 4), Q must have received an [ACK, V] or [ACK, B] from each process before or at t. On the other hand, when P decides, P's local variable $decision$ is 1, which means that P has not received any message [B, 0] or [V, 0] before or when P decides. Since $n+2f+1 \geq 2+2+1 = 5 > t$, P cannot manage to send [ACK, V] or [ACK, B] so that Q receives the message before or at t. A contradiction. □

(2n-2)NBAC. The full protocol is presented in Algorithm 5. As in (n-1+f)NBAC, the timer here starts at time 1 when the first sending event happens.

Proof. (Proof of correctness of (2n-2)NBAC.) We show that every crash-failure execution of (2n-2)NBAC solves NBAC. While doing so, we show that every execution of (2n-2)NBAC satisfies validity and termination. Recall that in every crash-failure execution, every message arrives in time while in an execution, timeouts may be violated.

Termination. Every correct process decides at time $3+f$.

Commit-Validity. In every execution, if a process P decides 1, then at time $3+f$, the local variable $votes$ is 1. If $P = P_n$, then at time 2, P must have received all n votes which are all 1. If $P \neq P_n$, then at time 3, P must have received a message [B, 1] from P_n, which implies that all

Algorithm 5 (2n-2)NBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.

upon event <(2n-2)nbac, Init> do
  votes := 1;
  received_B := FALSE;
  phase := 0;
  collection := {$P_i$};

upon event <(2n-2)nbac, Propose | v> do
  votes := votes AND v;
  if $1 \leq i \leq n-1$ then
    trigger <pl, Send | $P_n$, [V, v]>;
    set timer to time 3;
  else
    set timer to time 2;

upon event <pl, Deliver | p, [V, v]> do
  votes := votes AND v;
  collection := collection ∪ {p};

upon event <timer, Timeout> and phase = 0 and $i = n$ do
  if votes = 1 and collection = $\Omega$ then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, [B, 1]>;
  else
    votes := 0;
    forall $q \in \Omega$ do
      trigger <pl, Send | q, [B, 0]>;
  set timer to time $3 + f$;
  phase := 1;

upon event <timer, Timeout> and phase = 0 and $1 \leq i \leq n-1$ do
  if $received\_B$ = FALSE then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, [B, 0]>;
    votes := 0;
  set timer to time $3 + f$;
  phase := 1;

upon event <pl, Deliver | p, [B, v]> do
  received_B := TRUE;
```

```
  votes := v;
  if votes = 0 then
    forall q ∈ Ω do
      trigger <pl, Send | q, [B, 0]>;

upon event <timer, Timeout> and phase = 1 do
  trigger <(2n-2)nbac, Decide | votes>;
```

processes vote 1.

Abort-Validity. In every execution, if a process P decides 0, then at time $3+f$, the local variable $votes$ is 0. If $P = P_n$, then P votes 0, or receives a vote of 0 at time 2, does not receive some vote at time 2 or receives a message of [B, 0] (but sends a message of [B, 1] at time 2). The last two imply the crash of some process or the delay of some message. If $P \neq P_n$, then P votes 0, receives a message of [B, 0] from P_n at time 3, or does not receive any message from P_n at time 3, or receives a message of [B, 0] from some process (but receives a message of [B, 1] from P_n at time 3). The last two imply the crash of P_n or the delay of some message from P_n. Therefore, in every execution, if a process decides 0, then some process proposes 0 or a failure occurs.

Agreement. By contradiction. Suppose that E is a crash-failure execution such that two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. Then by the *commit-validity* property, every process votes 1. If $P = P_n$, then P has received all votes from all processes; since P decides at time $3+f$, P manages to send [B, 1] to every process and then Q should decide 1, which leads to a contradiction. If $P \neq P_n$, then P does not receive any message [B, 0] and moreover, P receives [B, 1] at time 3 from P_n. Clearly, For Q to decide 0, P_n must have crashed while sending [B,1] at time 2. (Otherwise, all have received [B,1], none sends message [B, 0] and then all decide 1. A contradiction.) Now that P_n crashes, Q must have received message [B, 0] later than time $2+f$. (Otherwise, P would receive a message of [B, 0] from Q earlier than or at time $3+f$ and thus decides 0, which is a contradiction.) Then between time 2 and time $f+2$, at least $f+1$ processes manage to send message [B,0] to some process and then crash, which contradicts the fact that at most f processes may crash. □

(2n-2+f)NBAC. We describe our (2n-2+f)NBAC protocol in Algorithm 6. Here if $f-1=n$, then the condition $f-1 \le i \le n-1$ is never fulfilled no matter what i is (and thus the related events are never triggered). As (n-1+f)NBAC, we also change the timer slightly from the other protocols: the timer here starts at time 1 when the first sending event happens.

Proof. (Proof of correctness of (2n-2+f)NBAC.)

Algorithm 6 (2n-2+f)NBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.
  IndulgentUniformConsensus, instance iuc.

upon event <(2n-2+f)nbac, Init> do
  votes := 1;
  received_V := FALSE;
  received_B := FALSE;
  received_Z := FALSE;
  phase := 0;
  decided := FALSE;
  proposed := FALSE;

upon event <(2n-2+f)nbac, Propose | v> do
  votes := votes AND v;
  if $i = 1$ then
    trigger <pl, Send | $P_2$, [V, v]>;
    set timer to time $n+1$;
    phase := 1;
  else
    set timer to time $i$;

upon event <pl, Deliver | p, [V, v]> and phase = 0 do
  votes := votes AND v;
  received_V := TRUE;

upon event <timer, Timeout> and phase = 0 do
  if received_V = TRUE then
    if $i = n$ then
      trigger <pl, Send | $P_1$, [B, votes]>;
    else
      trigger <pl, Send | $P_{i+1}$, [V, votes]>;
  else
    votes := 0;
    if proposed = FALSE then
      trigger <iuc, Propose | 0>;
      proposed := TRUE;
  set timer to time $n+i$;
  phase := 1;

upon event <pl, Deliver | p, [B, b]> and phase = 1 do
  votes := votes AND b;
  received_B := TRUE;
```

```
upon event <timer, Timeout> and phase = 1 and $i = f$ do
  if received_B = TRUE then
    trigger <pl, Send | $P_{f+1}$, [B, votes]>;
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
  else
    votes := 0;
    if proposed = FALSE then
      trigger <iuc, Propose | 0>;
      proposed := TRUE;
  phase = 2;

upon event <timer, Timeout> and phase = 1 and $i = n$ do
  if received_B = TRUE then
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
    if $f \geq 2$ then
      trigger <pl, Send | $P_1$, [Z, votes]>;
  else
    if proposed = FALSE then
      trigger <iuc, Propose | votes>;
      proposed := TRUE;

upon event <timer, Timeout> and phase = 1 and $1 \leq i \leq f-1$ do
  if received_B = TRUE then
    trigger <pl, Send | $P_{i+1}$, [B, votes]>;
  else
    votes := 0;
    if proposed = FALSE then
      trigger <iuc, Propose | 0>;
      proposed := TRUE;
  set timer to time $2n+i$;
  phase := 2;

upon event <timer, Timeout> and phase = 1 and $f+1 \leq i \leq n-1$ do
  if received_B = TRUE then
    trigger <pl, Send | $P_{i+1}$, [B, votes]>;
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
  else
    forall $q \in \{P_1, P_2, \ldots, P_f, P_n\}$ do
      trigger <pl, Send | q, [HELP]>;
```

```
upon event <pl, Deliver | p, [HELP]> and $i = n$ and
$phase = 1$ do
  trigger <pl, Send | p, [HELPED, votes]>;

upon event <pl, Deliver | p, [HELP]> and $1 \leq i \leq f$
and $phase = 2$ do
  trigger <pl, Send | p, [HELPED, votes]>;

upon event <pl, Deliver | p, [HELPED, v]> and not proposed do
  trigger <iuc, Propose | v>;
  proposed := TRUE;

upon event <pl, Deliver | p, [Z, z]> and phase = 2 do
  votes := votes AND z;
  received_Z := TRUE;

upon event <timer, Timeout> and phase = 2 and $1 \leq i \leq f-1$ do
  if received_Z = TRUE then
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
    if $f - 1 \geq i + 1$ then
      trigger <pl, Send | $P_{i+1}$, [Z, votes]>;
  else
    if proposed = FALSE then
      trigger <iuc, Propose | votes>;
      proposed := TRUE;

upon event <iuc, Decide | d> and not decided do
  trigger <(2n-2+f)nbac, Decide | d>;
  decided := TRUE;
```

Termination. Consider any crash-failure (or network-failure) execution E. In E, every correct process proposes a vote. For a correct process $P_i, i \in \{f, n\}$, the timer eventually timeouts at time $n+i$; at time $n+i$, P_i either decides without invoking iuc in (2n-2+f)NBAC or proposes a value to iuc. For a correct process $P_i, i \in \{1, 2, \ldots, f-1\}$, P_i eventually timeouts at time $2n+i$; at time $2n+i$, P_i decides without invoking iuc or proposes a value to iuc. For a correct process $P_i, i \in \{f+1, f+2 \ldots, n-1\}$, P_i eventually timeouts at time $n+i$; at time $n+i$, P_i either decides at time $n+i$ or queries $P_1, P_2, \ldots, P_f, P_n$ for help. If P_n is correct, then P_n eventually assigns 1 to $phase$; if a process in $\{P_1, P_2, \ldots, P_f\}$ is correct, then the process eventually assigns 2 to $phase$. Since at most f processes may crash, then at least one process in $\{P_1, P_2, \ldots, P_f, P_n\}$ is correct and therefore P_i receives at least one message [HELPED, *] and then proposes a value to iuc. Thus by the termination property of iuc in a crash-failure system (or in a network-failure system), every correct process decides in E.

Commit-Validity. In every execution, if a process P decides 1, then P's decision is either a decision of iuc or not. If P's decision is not a decision of iuc, then P decides its local variable $votes = 1$ and there are four possibilities for P when P decides: (1) $P = P_f$, $phase = 1$, $received\_B$ is TRUE; (2) $P = P_n$, $phase = 1$, $received\_B$ is TRUE; (3) $P \in \{P_{f+1}, P_{f+2}, \ldots, P_{n-1}\}$, $phase = 1$, $received\_B$ is TRUE; (4) $P \in \{P_1, P_2, \ldots, P_{f-1}\}$, $phase = 2$, $received\_Z$ is TRUE. By the protocol, in each of the four possibilities, $votes$ is the logical AND of all n votes and thus every process proposes 1. If P's decision is a decision of iuc, then some process Q proposes $votes = 1$ or $v = 1$ to iuc and therefore there are three possibilities when Q proposes: (1) $Q = P_n$, $phase = 1$, $received\_B$ is FALSE, $received\_V$ is TRUE and Q proposes $votes$; (2) $Q \in \{P_{f+1}, P_{f+2}, \ldots, P_{n-1}\}$, $phase = 1$, $received\_B$ is FALSE, Q delivers message [HELPED, v] from some process $p \in \{P_1, P_2, \ldots, P_f, P_n\}$ and Q proposes v; (3) $Q \in \{P_1, P_2, \ldots, P_{f-1}\}$, $phase = 2$, $received\_Z$ is FALSE, $received\_B$ is TRUE, $received\_V$ is TRUE, and Q proposes $votes$. By the protocol, in each of the three possibilities, $votes$ or v is the logical AND of all n votes and thus every process proposes 1.

Abort-Validity. In every execution, if a process P decides 0, then P's decision is either a decision of iuc or not. If P's decision is not a decision of iuc, then P decides its local variable $votes = 0$; since $votes$ is the logical AND of all n votes and thus some process proposes 0. If P's decision is a decision of iuc, then some process Q proposes 0 to iuc. If Q proposes Q's local variable $votes$, then again $votes$ is the logical AND of all n votes and thus some process proposes 0. If Q proposes v where Q delivers message [HELPED, v] from some process $p \in \{P_1, P_2, \ldots, P_f, P_n\}$, then there are two possibilities when Q proposes to iuc: (1) $p = P_n$, $phase = 1$, $received\_V$ is FALSE; (2) $p \in \{P_1, P_2, \ldots, P_f\}$, $phase = 2$, $received\_B$ is FALSE. We note that by the protocol, in every crash-failure execution where no process crashes, neither (1) nor (2) occurs. As a result, if (1) or (2) occurs, then some process must have crashed or some message must have been delayed. If Q proposes 0 to iuc, then there are three possibilities when Q proposes 0 to iuc: (1) $Q \in \{P_2, P_3, \ldots, P_n\}$, $phase = 0$, $received\_V$ is FALSE; (2) $Q = P_f$, $phase = 1$, $received\_B$ is FALSE; (3) $Q \in \{P_1, P_2, \ldots, P_{f-1}\}$, $phase = 1$, $received\_B$ is FALSE. By the protocol, in every crash-failure execution where no process crashes, none of the three possibilities occurs. As a result, if (1) or (2) occurs, then some process must have crashed or some message must have been delayed.

Agreement. By contradiction. Suppose that E is an execution such that two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. Then by the *agreement* property of uniform consensus, at least one of P and Q's decisions is not a decision of iuc. If P's decision is a decision of iuc, then Q's decision is not a decision of iuc; however, by the proof of the *commit-validity* property above, every process proposes 1 and thus when Q decides, Q's local variable $votes = 1$, which leads to a contradiction. If P's decision is not a decision of iuc, then by the proof of the *commit-validity* property above, every process proposes 1 and moreover,

Q's decision must be a decision of iuc; as a result, some process R proposes 0 or v to iuc. When P decides, if a process in $\{P_1, P_2, \ldots, P_f\}$ has not yet crashed, then its local variables $phase = 2$, $received\_B$ is TRUE (when $phase$ is assigned to 2), $received\_V$ is TRUE (when $phase$ is assigned to 2); if a process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ has not yet crashed, then its local variables $phase = 1$, $received\_V$ is TRUE (when $phase$ is assigned to 1). Therefore, R cannot propose 0 when at R, $phase = 0$ or 1; since $received\_V$ and $received\_B$ can only be assigned to TRUE by the protocol (except for initialization), no process would send message [HELPED, 0] to R and thus R cannot propose $v = 0$. As a result, R does not exist, which gives rise to a contradiction. □

2.5 Indulgent Atomic Commit

In this section, we present our INBAC protocol. INBAC solves indulgent atomic commit as defined below. We believe this protocol to be of practical relevance for it is suited to practical distributed database systems which are synchronous “most of the time”.

Definition 4 (Indulgent atomic commit)**.** A protocol π solves *indulgent atomic commit* if it satisfies the following:

- Every network-failure execution of π solves NBAC.

Indulgent atomic commit is the most robust atomic commit problem in Table 2.1. For this problem, we show that our INBAC protocol is optimal in the number of message delays, as well as in the number of messages given that optimal number of message delays. To give the intuition behind the optimal protocol, we first prove the lower bounds on the number of messages, and then sketch the optimal protocol. For completeness, we also include the full protocol and its proof of correctness.

2.5.1 Lower bounds

Recall that we have proven the lower bound on the number of message delays of indulgent atomic commit in Section 2.3. Here we prove a lower bound on the number of messages exchanged given two message delays (which is optimal as shown in Theorem 1) during any nice execution actually for a less robust problem (than indulgent atomic commit), as stated in the following theorem.

Theorem 5 (Lower bound on messages given fewer than three message delays)**.** *Let π be any protocol that (a) solves NBAC in every crash-failure execution, and (b) satisfies agreement in every network-failure execution. Let E be any nice execution of π where every message is transmitted after an exact message delay U. W.l.o.g., E starts at time 0. If every process decides at or before $2U$ in E, then at least $2fn$ messages are exchanged in E.*

Note that for this less robust problem as well as indulgent atomic commit, $2n-2+f$ $(f \geq 2)$ messages are optimal. Thus Theorem 5 also demonstrates the tradeoff between the number of messages and that of message delays for this less robust problem, indulgent atomic commit and other related problems (in total 4 cases out of 27 ones which we consider). As a result, including our tradeoff results obtained in Section 3, all atomic commit problems with nonzero messages as lower bounds (in total 18 out of 27 problem variants) highlight a tradeoff between time and message complexity.

To prove the lower bound of $2fn$ messages, we count the number of necessary messages for each of the n processes. In particular, we show in any nice execution, for any process P, there are two non-overlapping sets of f messages, Λ_1 and Λ_2, such that every message in Λ_1 precedes some message in Λ_2. To describe the relation between those messages precisely, we again apply the notion of "process reachability" introduced in Definition 3 and complete the terminology.

Definition 5 (Reaching a process: complete terminology)**.** Let E be any execution. Let $\underline{m} = \{m_1, m_2, \ldots, m_l\}$ be a sequence of messages in E such that (a) the source of m_1 is P, (b) the destination of m_l is $Q, Q \neq P$, (c) the source src_i of m_i is the destination of m_{i-1} for $i = 2, 3, \ldots, l$, and (d) m_i leaves from src_i later than or at the time at which m_{i-1} arrives at src_i for $i = 2, 3, \ldots, l$.

Recall that (as defined in Definition 3) if m_l arrives at Q at time t or earlier and $\underline{m}$ is the earliest sequence of messages for P (according to t) to reach Q in E, then we say that P *has reached* Q at time t in E.

For any two processes P and Q, if there are two sequences of messages $\underline{m}^1 = m_1^1, m_2^1, \ldots, m_{l_1}^1$ and $\underline{m}^2 = m_1^2, m_2^2, \ldots, m_{l_2}^2$ such that (a) the source of m_1^1 and the destination of $m_{l_2}^2$ is P, (b) the source of m_1^2 and the destination of $m_{l_1}^1$ is Q, (c) m_1^2 leaves from Q later than or at the time at which $m_{l_1}^1$ arrives at Q, and (d) $m_{l_2}^2$ arrives at some time t or earlier, then we say that P reaches Q and *subsequently* Q reaches P before time t (including t).

More generally, given three processes P, Q and R, if there are two sequences of messages $\underline{m}^1 = m_1^1, m_2^1, \ldots, m_{l_1}^1$ and $\underline{m}^2 = m_1^2, m_2^2, \ldots, m_{l_2}^2$ such that (a) the source of m_1^1 is R, (b) the destination of $m_{l_2}^2$ is P, (c) the source of m_1^2 and the destination of $m_{l_1}^1$ is Q, (d) m_1^2 leaves from Q later than or at the time at which $m_{l_1}^1$ arrives at Q, and (e) $m_{l_2}^2$ arrives at some time t or earlier, then we say that R reaches Q and *subsequently* Q reaches P before time t (including t).[11]

Recall that if a process P reaches another process Q, then it is possible that by a sequence of messages, P backs up P's vote at Q. (Lemma 1 actually captures the intuition of backups.) Similarly, if P reaches Q and subsequently Q reaches P, then it is possible that by a sequence of messages, Q *acknowledges* the backup of P's vote at Q. Then Lemma 5 below essentially

---

[11] The time t mentioned in Definition 5 is only for convenience of our proof: the time is assumed to be an accurate global clock, but no process necessarily has access to the global clock.

says that at least f processes must send acknowledgements that confirm the success of the backup, which is also the intuition for our proof of lower bound.

Lemma 5 (Quick acknowledgements)**.** *Let π be any protocol that (a) solves NBAC in every crash-failure execution, and (b) satisfies agreement in every network-failure execution. Let E be any nice execution of π. Let P decide at some time t_1 in E. Let Θ be such set of processes that $\forall Q \in \Theta$, Q satisfies that before t_1 (including t_1) in E, P reaches Q, and subsequently Q reaches P. Among the messages whose destination is P, let $\mathcal{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathcal{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m \in \mathcal{M}} t_m$.*

If $t_2 \leq 2U$, then $|\Theta| \geq f$.

Proof. By contradiction. Suppose that $|\Theta| \leq f-1$. Denote by Φ the set of P and the processes which P has reached at t_2. According to the definition of Θ and t_2, $\Theta \subseteq \Phi$. For each process $Q \in \Theta$, denote by τ_Q the time at which P reaches Q in E. For each process $Q^- \in \Phi \backslash (\Theta \cup \{P\})$, denote by τ_{Q^-} the time at which P reaches Q^- in E.

We build a crash-failure execution E_0 based on E. In E_0, P crashes before sending any message. For Q, E_0 is the same as E until Q crashes and Q crashes before sending any message that is expected to send upon the message(s) received by Q at τ_Q (i.e., Q crashes at τ_Q). For every other process (i.e., a process not in $\Theta \cup \{P\}$), E_0 is the same as E until some process in $\Theta \cup \{P\}$ timeouts at some process not in $\Theta \cup \{P\}$.

Now we construct E_0 after some process timeouts as follows. First, we consider the earliest timeout. The earliest timeout occurs at a process in $\Phi \backslash (\Theta \cup \{P\})$. (By Lemma 1, $|\Phi \backslash \{P\}| \geq f$. As $|\Theta| \leq f-1$, $\Phi \backslash (\Theta \cup \{P\})$ is non-empty.) Let $Q^-_{timeout} \in \Phi \backslash (\Theta \cup \{P\})$ be the process at which the earliest timeout occurs. Denote by t_3 at which the earliest timeout occurs. Clearly, $t_3 > U$. If $Q^-_{timeout}$ sends any message m_1 upon the timeout event, then we assume that m_1 arrives at its destination at time $t_3 + U$. Second, any other message that is different from E due to the timeout events arrives in a delay similarly, i.e., with the same message delay U. Finally, in E_0, P proposes 0, every other process proposes 1 and no process in $\Omega \backslash (\Theta \cup \{P\})$ crashes. As $|\Theta| \leq f-1$, E_0 is a legitimate crash-failure execution of π. Any process $R \in \Omega \backslash (\Theta \cup \{P\})$ decides 0 in E_0. W.l.o.g., let R be the earliest process that decides. Denote by t_4 the time at which R decides.

Then based on E and E_0, we build a network-failure execution E_{async}. In E_{async}, every process proposes 1 and no process crashes. Therefore, E_{async} starts as E. Then we construct E_{async} such that:

- Every message from P to a process in $\Omega \backslash (\Theta \cup \{P\})$ arrives later than $\max(t_1, t_4)$;
- Every message from Q to a process in $\Omega \backslash (\Theta \cup \{P\})$ sent after or at τ_Q arrives later than $\max(t_1, t_4)$;

- Every message from Q^- to P sent after or at τ_{Q^-} arrives later than t_1.
- Every message sent after t_2 to a process in $\Theta \cup \{P\}$ arrives later than t_1 at the process.

For every process $Q \in \Theta$, $t_Q > \tau_Q$. Thus to every process in $\Omega \backslash (\Theta \cup \{P\})$, any process in $\Theta \cup \{P\}$ seems to crash at the same time as in E_0. The first timeout event occurs at the same time t_3 at the same process $Q^-_{timeout}$ as in E_0. Then to every process in $\Omega \backslash (\Theta \cup \{P\})$, E_{async} is indistinguishable from E_0 before and at the first timeout. We let the messages from/to a process in $\Omega \backslash (\Theta \cup \{P\})$ after the first timeout event be (sent/received) the same as in E_0. Therefore to every process in $\Omega \backslash (\Theta \cup \{P\})$, E_{async} is indistinguishable from E_0 before and at t_4.

To Q^-, E_{async} and E are indistinguishable only before τ_{Q^-}. After τ_{Q^-}, Q^- can distinguish between E_{async} and E. There are two possibilities for any Q^- to help P in distinguishing between E_{async} and E: (1) Q^- sends a message in E_{async} which Q^- does not in E; and (2) Q^- does not send a message in E_{async} which Q^- does in E. For the first possibility, Let m_1 be the message sent in E_{async}. Then m_1 is sent after or at $\max(t_3, \tau_{Q^-})$. The same message m_1 is sent in E_0 according to our construction. If m_1 is sent to P, then by our construction, m_1 arrives later than t_1; if m_1 is sent to any other process, then by our construction of E_0, m_1 arrives after or at $t_3 + U$ and thus the receiver of m_1 can only send a message m_2 after or at $t_3 + U$. As $t_3 > U$, $t_3 + U > 2U \geq t_2$; then m_1 does not help the receiver of m_1 in distinguishing between E_{async} and E before t_2. Hence in the first possibility, Q^- cannot help P in distinguishing between E_{async} and E before and at t_1.

For the second possibility, with an abuse of notations, let m_1 be the message sent and O be the receiver of m_1 in E. If $O = P$, then by the definition of Q^-, m_1 arrives later than t_1 in E and does not belong to $\mathscr{M}$. If O is any oher process, then O can only notice the missing of m_1 in E_{async} after or at $t_3 + U$. As a result, m_1 does not help any process other than P in distinguishing between E_{async} and E before t_2. Therefore, following both possibilities, still the same set of messages as $\mathscr{M}$ are received by P before and at t_1 in E_{async} and P is unable to distinguish between E_{async} and E before and at t_1.

Now that P is unable to distinguish between E_{async} and E at t_1, and R is unable to distinguish between E_{async} and E_0 at t_4, P decides 1 at t_1 and R decides 0 at t_4. As a result, E_{async} is a network-failure execution of π that does not satisfy the *agreement* property. A contradiction to the assumption that π solves indulgent atomic commit. □

As the proof of Lemma 5 shows, the sufficient condition in the lemma is non-trivial. In the proof, it is actually critical that P decides in *two or three message delays* for f acknowledgements to be necessary. Suppose that P decides slowly instead. Then P could expect a message from some process R in order to decide so that some process Q^- might notice the crash detection of P (or Q). Q^- might report it to P via R, and as a result, P may notice the incorrect crash detection of itself and wait for others (instead of taking a decision). This also leave an open question of whether f acknowledgements are necessary if a process decides after more than three message delays.

Given Lemma 5, we can go back to our intuition of Theorem 5. As shown in Lemma 5, certain messages do follow an order in any nice execution and because of the inherent order, there exist two non-overlapping sets of messages, Λ_1 and Λ_2, where intuitively Λ_1 backs up votes and Λ_2 acknowledges the success of backups, in any nice execution of a 2-delay protocol. We now prove our Theorem 5, the lower bound on the number of messages.

Proof. (Proof of Theorem 5.) Consider any process P and let t_1 be the time at which P decides. Among the messages whose destination is P, let $\mathscr{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathscr{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m \in \mathscr{M}} t_m$. Then $t_2 = U$ and $t_1 = 2U$. By Lemma 1, at least f messages leave from P at time 0, and by Lemma 5, at least f messages arrive at P at time $2U$. This, in total, gives at least $2fn$ messages during any nice execution. □

2.5.2 Optimal protocol: overview

We present here a protocol, which we denote INBAC, and which is delay-optimal as well as message-optimal given the optimal number of message delays.

We start by looking at what happens in nice executions of INBAC (which actually follows Lemma 1 and Lemma 5); then we explain in other executions, how INBAC uses an underlying consensus module to solve agreement. The state transition of a process in both executions (nice or not) is illustrated in Figure 2.1. For simplicity, for time $2U$ or earlier in INBAC, every process sends a message or decides at multiples of U, i.e., at time 0, U or $2U$.

Overview of INBAC.

- Nice execution. Every nice execution E of INBAC starts by $P_1, P_2, \ldots, P_n$ sending their votes simultaneously. At time 0, every process P sends P's vote to f processes. We say that those f processes are P's *backup processes*. At time U, each of P's backup processes sends P's vote back to P as an acknowledgement. INBAC chooses the set $\mathscr{B}_P$ of P's backup processes as follows: for $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$, $\mathscr{B}_P = \{P_1, P_2, \ldots, P_f\}$; for $P \in \{P_1, P_2, \ldots, P_f\}$, $\mathscr{B}_P = \{P_1, P_2, \ldots, P_{f+1}\} \setminus \{P\}$. Clearly, a process may backup more than one vote. In fact, at time U, P's backup process sends to P a set $\mathscr{V}$ of the votes received as an acknowledgement of the successful backup of each vote in $\mathscr{V}$. (This is a necessary design, which we summarize later in Lemma 6). Thus at time $2U$, P decides if P receives f correct acknowledgements (from P's f backup processes where a correct acknowledgement from process $B \in \mathscr{B}_P$ includes Q's vote for all Q such that $B \in \mathscr{B}_Q$). Obviously, in a nice execution, or more generally, in an execution where messages arrive in time, at $2U$, P knows every process's vote and is able to decide properly.

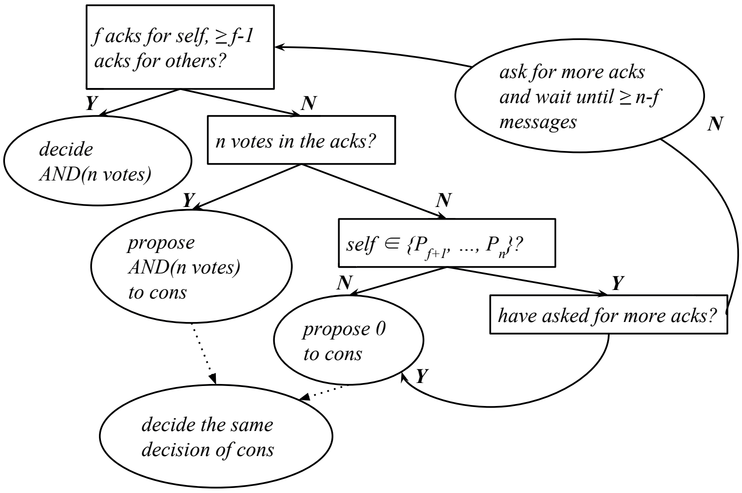


Figure 2.1 – State transition after $2U$

- Consensus to the rescue. Now in an execution E^- in which some process crashes, or some message is delayed, P can propose a value to consensus (we say that P may *cons-propose* a value) and wait for the decision of the consensus. We first explain when P cons-proposes a value and then explain which value P cons-proposes. Now, at $2U$, if P receives at least one acknowledgement from a process in $\{P_1, P_2, \ldots, P_f\}$, then P cons-proposes a value immediately at $2U$. Otherwise, P asks $P_{f+1}, P_{f+2}, \ldots, P_n$ for the acknowledgements which $P_{f+1}, P_{f+2}, \ldots, P_n$ have received. I.e., processes ask for help for the missing acknowledgements and their corresponding votes. To be more specific, if P is a process in $\{P_1, P_2, \ldots, P_f\}$, then P can always cons-proposes a value at $2U$ in E^-. If not and if at $2U$, P indeed receives no acknowledgement from any process in $\{P_1, P_2, \ldots, P_f\}$, then P eventually receives acknowledgement messages from $n - f$ out of n processes and then may cons-propose a value. At the point when P is ready to cons-propose a value, P looks for every process's vote in the acknowledgement messages which P has received so far. If P finds that every process's vote is 1, then P cons-proposes 1; otherwise, P cons-proposes 0.

The state transition of P in E and in E^- is illustrated in Figure 2.1. We use there the following notations: AND denotes the logical AND of those 0's and 1's as votes; Y and N are the abbreviated for *yes* and *no* respectively; $self$ denotes P, the process in question; ack denotes an acknowledgement; $cons$ denotes consensus (which is not invoked if no process crashes and every message arrives in time).

Some remarks on the protocol are in order. Clearly, the strategy of decisions of our INBAC protocol is independent of the underlying consensus algorithm. In addition, INBAC does not necessarily decide 0 when a failure occurs. When a process succeeds in collecting all votes by helping (while, for example, another process may have crashed), INBAC encourages it to propose 1 to consensus by looking at every process's vote in the acknowledgements received. Hence INBAC may decide 1 when a failure occurs.

Best-case complexity. We now count the number of messages and that of message delays. Since in every nice execution every process decides at $2U$, then the number of message delays meets the lower bound (Theorem 1). As for the number of messages in any nice execution, at time 0, for every process P, f messages leave from P; at time $2U$, exactly f messages arrive at the same process P.[12] (This is because in INBAC, a backup process sends the acknowledgement of several votes $\mathcal{V}$ in one message). Therefore, among n processes, $2fn$ messages are exchanged in E, which meets the lower bound on the number of messages in Theorem 5. This optimal result shows that both lower bounds are tight, as summarized in Theorem 6.

In the version as described above, the complexity of INBAC of a failure-free execution in which some process votes 0 is the same as the complexity of any nice execution. We remark that our protocol INBAC may accelerate such failure-free execution by doing the following: if a process P votes 0, then P sends its vote to every process and decides 0 at the very beginning (and in the meantime, a process $Q \neq P$ who receives one vote of 0 decides 0 immediately). Then a failure-free execution in which some process votes 0 can terminate at the end of the *first* message delay, which is faster than any nice execution.

Theorem 6 (Message-optimal indulgent atomic commit given two message delays)**.** *Given any protocol that solves consensus in a network-failure system, INBAC solves indulgent atomic commit, and during every nice execution of INBAC, (a) any process decides after two message delays, and (b) n processes exchange* $2fn$ *messages.*

Finally, as we claimed in the beginning of this section, we note a necessary design for the optimal protocol. We show in Lemma 6 that $f-1$ acknowledgements of other processes' votes are necessary. (Our INBAC adopts this design for optimality; for example, when P_{f+1} decides in a nice execution, P_{f+1} has received exactly $f-1$ acknowledgements of P_1's votes.) As both Lemma 5 and Lemma 6 are necessary in designing message-optimal protocols, e.g., given three message delays, they may be of independent interest and worth mentioning here.

Lemma 6 (Quick acknowledgements of other votes)**.** *Let* π *be any indulgent atomic commit protocol. Let* E *be any nice execution of* π*. Let* P *decide at some time* t_1 *in* E*. Let process* $R \neq P$*. Let* Θ *be such set of processes that* $\forall Q \in \Theta$*,* Q *satisfies that before* t_1 *(including* t_1*) in* E*,* R *reaches* Q*, and subsequently* Q *reaches* P*. Among the messages whose destination is* P*, let* $\mathcal{M}$ *be the set of messages that arrives at* P *before or at* t_1*. For each* $m \in \mathcal{M}$*, let* t_m *be the time at which* m *leaves from its source and let* $t_2 = \max_{m \in \mathcal{M}} t_m$*.*

If $t_2 \leq 2U$*, then* $|\Theta| \geq f-1$*.*

Proof. By contradiction. Suppose that $|\Theta| \leq f-2$. Denote by τ_Q the time at which R reaches

---

[12] A message whose source and destination is the same does not need to be sent over the network; such a message arrives immediately and is not counted in the messages exchanged among the n processes.

Q in E. Denote by τ_P the time at which R reaches P in E. Denote by Φ the set of R and the processes which R has reached at t_2. Denote by τ_{Q^-} the time at which R reaches Q^- for each process $Q^- \in \Phi \setminus (\Theta \cup \{P, R\})$ in E.

We build a crash-failure execution E_0 based on E. In E_0, R crashes before sending any message (i.e., R crashes at time 0). For Q, E_0 is the same as E until Q crashes and Q crashes before sending any message that is expected to send upon the message(s) received by Q at τ_Q (i.e., Q crashes at τ_Q). For P, E_0 is the same as E until P crashes before sending any message that is expected to send upon the message(s) received by P (i.e., P crashes at τ_P). For every other process O, E_0 is the same as E until some process in $\Theta \cup \{P, R\}$ timeouts at some process not in $\Theta \cup \{P, R\}$.

Now we construct E_0 after some process timeouts as follows. First, we consider the earliest timeout. If $\Phi \setminus (\Theta \cup \{P, R\})$ is empty, the earliest timeout occurs at a process later than t_2. If $\Phi \setminus (\Theta \cup \{P, R\})$ is non-empty, the earliest timeout occurs at a process in $\Phi \setminus (\Theta \cup \{P, R\})$. Let Q^- be the process at which the earliest timeout occurs. Denote by t_3 at which the earliest timeout occurs. Certainly, whether $Q^- \in \Phi \setminus (\Theta \cup \{P, R\})$ or not, $t_3 > U$. W.l.o.g., we assume that $Q^- \in \Phi \setminus (\Theta \cup \{P, R\})$. If Q^- sends any message m_1 upon the timeout event, then we assume that m_1 arrives at its destination at time $t_3 + U$. Second, any other message that is different from E due to the timeout events arrives in a delay similarly, i.e., with the same message delay U. Finally, every message that is sent after t_2 arrives later than t_1.

Moreover, in E_0, R proposes 0, every other process proposes 1 and no process in $\Omega \setminus (\Theta \cup \{P, R\})$ crashes. As $|\Theta| \le f - 2$ and $t_1 - t_2 \le U$, E_0 is a legitimate crash-failure execution of π. Any remaining process $O \in \Omega \setminus (\Theta \cup \{P, R\})$ decides 0 in E_0. W.l.o.g., let O be the earliest process that decides. Denote by t_4 at which O decides.

Then based on E and E_0, we build a network-failure execution E_{async}. In E_{async}, every process proposes 1 and no process crashes. Therefore, E_{async} starts as E. Let $\Omega_1 = \Omega \setminus (\Theta \cup \{P, R\})$. Let $\Phi_1 = \Phi \setminus (\Theta \cup \{P, R\})$. Then we construct E_{async} such that:

- Every message from R to a process in Ω_1 arrives later than $\max(t_1, t_4)$;
- Every message from Q to a process in Ω_1 sent after or at τ_Q arrives later than $\max(t_1, t_4)$;
- Every message from P to a process in Ω_1 sent after or τ_P arrives later than $\max(t_1, t_4)$.
- Every message from a process in Φ_1 to R arrives later than $\max(t_1, t_4)$;
- Every message from a process Q^- in Φ_1 to Q sent after or at τ_{Q^-} arrives later than $\max(t_1, t_4)$.
- Every message from a process Q^- in Φ_1 to P sent after or at τ_{Q^-} arrives later than $\max(t_1, t_4)$.
- Every message sent after t_2 arrives later than t_1.

In addition, the rest of the messages which are communicated among $\Omega\backslash(\Theta \cup \{P,R\})$ are (sent/received) the same as in E_0 after the first timeout event. This timeout event occurs at the same time t_3 at Q^- in both E_{async} and E_0.

Process Q^- might send some message m_1 due to the timeout event. If m_1 is sent to P, Q or R, then we assume that m_1 arrives later than t_1; if m_1 is sent to some process O in $\Omega\backslash(\Theta \cup \{P,R\})$, then O can only send some message m_2 after or at $t_3 + U$. As $t_3 > U$, $t_3 + U > 2U \geq t_2$ and therefore, m_2 also arrives later than t_1. Thus any message that is different from E due to the timeout events arrives later than t_1. Then E_{async} and E are indistinguishable for P before and at t_1. As a result, P decides 1 at t_1.

Process O is among $\Omega\backslash(\Theta \cup \{P,R\})$. For O, E_{async} is the same as E_0 before and at t_4. As a result, O decides 0 at t_4.

Clearly, E_{async} is a network-failure execution of π that does not satisfy the *agreement* property. A contradiction to the assumption that π solves indulgent atomic commit. □

2.5.3 Full protocol INBAC

We describe here our INBAC protocol in detail as shown in Algorithm 7. Here the timer starts at time 0 when every process proposes its value as assumed in the beginning of Section 2.4. Each unit of time is set to the known upper bound of the message delay of the given crash-failure system. Sending messages and processing messages are considered negligible in time. In practice, processes may spend different amounts of time in processing a (sub)transaction (which is to be committed or aborted through the protocol); the crash-failure system here imposes that the this amount of time is also known and upper bounded„ and has already been included in the unit of time for the timer (so that the even if different processes start the protocol at different time instants, the use of the timer helps them to incorporate this difference). Thus in a network-failure execution of the protocol, if the timer timeouts and a process has not yet received the message (which it sets the timer to wait for), then a failure (network failure or crash failure) occurs, and vice-versa. Below we also present the proof of correctness of Algorithm 7, i.e., our INBAC protocol.

Proof. (Proof of Correctness of INBAC as well as Theorem 6.) First, we prove that every execution of INBAC satisfies the *agreement* property.

Agreement. By contradiction. Suppose that in some execution E, two different processes P and Q decide differently. Suppose further that P decides 1 and Q decides 0. Given that consensus satisfies the *agreement* property, at least one of P and Q's decisions is *not* a result of the decision of the consensus.

Algorithm 7 INBAC

```
Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.
  IndulgentUniformConsensus, instance iuc.

upon event <inbac, Init> do
  phase := 0;
  proposed := FALSE;
  decided := FALSE;
  collection0 := $\emptyset$;
  collection1 := $\emptyset$;
  collection_help := $\emptyset$;
  wait := FALSE;
  val := $\perp$;
  decision := $\perp$;
  proposal := $\perp$;
  cnt := 0;
  cnt_help := 0;

upon event <inbac, Propose | v> do
  val := v;
  forall $q \in \{P_1, \dots, P_f\}$ do
    trigger <pl, Send | q, [V, v]>;
  if $1 \le i \le f$ then
    trigger <pl, Send | $P_{f+1}$, [V, v]>;
  if $1 \le i \le f+1$ then
    set timer to 1;
  else
    set timer to 2;
    phase := 1;

upon event <pl, Deliver | p, [V, v]> and phase = 0 do
  collection0 := collection0 $\cup$ {(p, v)};

upon event <timer, Timeout> and phase = 0 and $1 \le i \le f$ do
  forall $q \in \Omega$ do
    trigger <pl, Send | q, [C, collection0]>;
  phase := 1;
  set timer to 2;

upon event <timer, Timeout> and phase = 0 and $i = f + 1$ do
  forall $q \in \{P_1, P_2, \dots, P_f\}$ do
    trigger <pl, Send | q, [C, collection0]>;
  phase := 1;
  set timer to 2;
```

```
upon event <pl, Deliver | p, [C, collection]> do
  collection1 := collection1 ∪ {(p, collection)};
  cnt := cnt + 1;

upon event <timer, Timeout> and phase = 1 and not decided and not proposed and $i \geq f + 1$ do
  phase := 2;
  collection_val := $\bigcup_{(p,c) \in collection1} c$;
  collection0 := collection0 ∪ collection_val ∪ {(self, val)};
  if $collection1 = \{(P_j, c_j) \mid 1 \leq j \leq f\}$ where $c_j = \{(P_k, val_k) \mid 1 \leq k \leq n\}$ for every $j$, $1 \leq j \leq f$ (with $val_k$ being the proposal of $P_k$) then
    decision := $\text{AND}_{1 \leq k \leq n} val_k$;
    decided := TRUE;
    trigger <inbac, Decide | decision>;
  else if $cnt \geq 1$ then
    if for every process $P_k$, $1 \leq k \leq n$, $\exists\, val_k$ s.t. $(P_k, val_k) \in \bigcup_{(p,c) \in collection1} c$ then
      proposal := $\text{AND}_{1 \leq k \leq n} val_k$;
      proposed := TRUE;
      trigger <iuc, Propose | proposal>;
    else
      proposed := TRUE;
      trigger <iuc, Propose | 0>;
  else
    wait := TRUE;
    forall $q \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$ do
      trigger < pl, Send | q, [HELP]>;

upon event <pl, Deliver | p, [HELP]> and phase = 2 and $i \geq f + 1$ do
  trigger < pl, Send | p, [HELPED, collection0]>;

upon event <pl, Deliver | p, [HELPED, collection]> and $i \geq f + 1$ do
  collection_help := collection_help ∪ collection;
  cnt_help := cnt_help + 1;

upon cnt + cnt_help ≥ $n - f$ and wait and not proposed and not decided and $i \geq f + 1$ do
  wait := FALSE;
  if $collection1 = \{(P_j, c_j) \mid 1 \leq j \leq f\}$ where $c_j = \{(P_k, val_k) \mid 1 \leq k \leq n\}$ for every $j$, $1 \leq j \leq f$ (with $val_k$ being the proposal of $P_k$) then
    decision := $\text{AND}_{1 \leq k \leq n} val_k$;
    decided := TRUE;
    trigger <inbac, Decide | decision>;
  else if $cnt \geq 1$ then
    if for every process $P_k$, $1 \leq k \leq n$, $\exists\, val_k$ s.t. $(P_k, val_k) \in \bigcup_{(p,c) \in collection1} c$ then
      proposal := $\text{AND}_{1 \leq k \leq n} val_k$;
      proposed := TRUE;
      trigger <iuc, Propose | proposal>;
```

```
    else
      proposed := TRUE;
      trigger <iuc, Propose | 0>;
  else
    if $collection\_help = \{(P_k, val_k) \mid 1 \leq k \leq n\}$ where $val_k$ is the proposal of $P_k$ then
      proposal := $\text{AND}_{1\leq k\leq n} val_k$;
      proposed := TRUE;
      trigger <iuc, Propose | proposal>;
    else
      proposed := TRUE;
      trigger <iuc, Propose |0 >;

upon event <timer, Timeout> and phase = 1 and not decided and not proposed and $1 \leq i \leq f$ do
  if $collection1 = \{(P_j, c_j) \mid 1 \leq j \leq f+1\}$ where $c_j = \{(P_k, val_k) \mid 1 \leq k \leq n\}$ for every $j$, $1 \leq j \leq f$ and $c_{f+1} = \{(P_k, val_k) \mid 1 \leq k \leq f\}$ (with $val_k$ being the proposal of $P_k$) then
    decision := $\text{AND}_{1\leq k\leq n} val_k$;
    decided := TRUE;
    trigger <inbac, Decide | decision>;
    return;
  if for every process $P_k$, $1 \leq k \leq n$, $\exists\, val_k$ s.t. $(P_k, val_k) \in \bigcup_{(p,c)\in collection1} c$ then
    proposal := $\text{AND}_{1\leq k\leq n} val_k$;
    proposed := TRUE;
    trigger <iuc, Propose | proposal>;
  else
    proposed := TRUE;
    trigger <iuc, Propose | 0>;

upon event <iuc, Decide | v> and not decided do
  decided := TRUE;
  trigger <inbac, Decide | v>;
```

If neither of P and Q's decisions is a result of the decision of the consensus, then either process decides the value of its local variable $decision$. Since $decision$ is assigned as the AND of the n processes' votes to $inbac$ at every process, P and Q must agree on their decisions, which contradicts our assumption. If P's decision is a result of the decision of the consensus, then by the *validity* property of consensus, some process R proposes 1 to iuc. Therefore R's local variable $proposal$ is 1, which is equal to the AND of the n processes' votes to $inbac$. Now that Q's decision is equal to its local variable $decision$, which is the AND of the n processes' votes to $inbac$, P and Q must agree on their decisions, which contradicts our assumption.

As a result, Q's decision must be a result of the decision of the consensus while P's decision must not be. Now P's local variable $decision$ is 1. Therefore, every process proposes 1 to $inbac$ and at the same time, if any process assigns a value to its local variable $proposal$

or $decision$, it can only assign a 1. Since Q's decision is a result of the consensus, by the *validity* property of consensus, some process R (not necessarily Q) proposes 0 to consensus. First, we assume that $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$ and examine whether R exists. As P decides 1, variable $collection0$ at every process in $\{P_1, P_2, \ldots, P_f\}$ is $\{(P_k, val_k) | 1 \leq k \leq n\}$. Therefore, $R \notin \{P_1, P_2, \ldots, P_f\}$. I.e., $R \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$. Then variable cnt at R must be 0 and thus for R to propose 0, R must have $cnt\_help = n - f$, i.e., every process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ has sent to R their variable $collection0$. As a result, P has also sent its $collection0$, which is updated to $\{(P_k, val_k) | 1 \leq k \leq n\}$ when $phase = 2$. This leads R to propose 1 to consensus. A contradiction.

Now we assume that $P \in \{P_1, P_2, \ldots, P_f\}$ and examine whether R exists. Similarly, variable $collection0$ at every process in $\{P_1, P_2, \ldots, P_f\}$ is $\{(P_k, val_k) | 1 \leq k \leq n\}$. Moreover, variable $collection0$ at P_{f+1} includes $\{(P_k, val_k) | 1 \leq k \leq f\}$ as a subset. Again, R must belong to $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$. For R to propose 0, R must have $cnt\_help = n - f$, i.e., every process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ has sent to R their updated variable $collection0$, the union of which is equal to $\{(P_k, val_k) | 1 \leq k \leq n\}$. (Variable $collection0$ at every process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ is updated to include its own vote.) This again leads R to propose 1 to consensus. A contradiction.

Next, we prove that every network-failure execution of INBAC satisfies the *validity* property, and the *termination* property.

Validity. Clearly, the *validity* property can be separated into the *commit-validity* property: if a process decides 1, then every process proposes 1; and the *abort-validity* property: if a process decides 0, then some process proposes 0 or a failure occurs. The proof here (and the proofs for the correctness of protocols later) proves that the protocol satisfies the *commit-validity* property and the *abort-validity* property respectively.

Commit-Validity. Suppose that some process P decides 1. If P's decision is a result of the decision of the consensus, then since consensus satisfies the *validity* property, some process R (not necessarily P) must propose 1 to consensus. Since variable $proposal$ at R is equal to the AND of the n votes, every process proposes 1 to $inbac$. If P's decision is not a result, then variable $decision$ at P is equal to the AND of the n votes, which implies that every process proposes 1 to $inbac$.

Abort-Validity. Suppose that process P decides 0. If P's decision is equal to variable $decision$ at P or variable $proposal$ at some other process R, then some process must propose 0 to $inbac$. If not, then some process R (not necessarily P) must have proposed 0 to consensus in the case where some value is missing in variable $collection\_help$ or the collection $\bigcup_{(p,c) \in collection1} c$ at R. This indicates that some message does not arrive before the timer issues a timeout event, which is set to the upper bound of the message delay. Then, in a network-failure system, we can safely conclude that a failure occurs. Thus the *abort-validity* property is satisfied.

Termination. By contradiction. Suppose that some correct process P does not decide. P assigns $phase$ to 1 in finite time. Then P is triggered by the event that the timer issues a timeout and $phase = 1$, when P has not proposed to consensus or decided in $inbac$. If $P \in \{P_1, P_2, \ldots, P_f\}$, then since consensus iuc satisfies the *termination* property in a network-failure system, P eventually decides in $inbac$. A contradiction. If $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$, then P assigns $phase$ to 2 in finite time. In fact, every correct process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ assigns $phase$ to 2 in finite time. Since P does not decide, thus by the *termination* property of iuc in a network-failure system, P must assign $wait$ to TRUE and wait for the condition $cnt + cnt\_help \geq n - f$ to satisfy. If the condition is satisfied and the corresponding event is triggered, then P eventually decides in $inbac$. In other words, for P to not decide, the condition should never be satisfied.

However, when $wait$ is assigned to TRUE, cnt is 0. Only the message of [C, *] increments cnt. Since $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$, then the message of [C, *] that arrives at P can only be from a process in $\{P_1, P_2, \ldots, P_f\}$, each correct process of which must send [C, *] to P. On the other hand, cnt_help at P is incremented if a message from a process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ arrives. Every correct process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ also must send message [HELPED, *] to P. As at most f processes can crash and messages eventually arrive at their destinations respectively, $cnt + cnt\_help$ is eventually equal to or greater than $n - f$. In other words, the condition is eventually satisfied. A contradiction.

Finally, since consensus satisfies the *termination* property in an network-failure system (assuming a majority of correct processes), INBAC also satisfies the *termination* property in an network-failure system (assuming a majority of correct processes).

Therefore, given that consensus can be implemented for a network-failure system, protocol INBAC (i.e., instance *inbac*) solves indulgent atomic commit. □

2.6 Related Work

2.6.1 Complexity of commit protocols

The formal study of atomic commit problems dates back to Skeen [16]. Later, substantial refinement [65, 68, 69] has been made, leading to the properties of Non-Blocking Atomic Commit (NBAC) considered in Chapter 2. A comparison with previous definitions from the literature is now in order. A synchronous NBAC protocol [16, 1] is a protocol which solves NBAC in a crash-failure system (and thus the complexity is covered by our study). In previous impossibility results [68, 83, 84, 75, 76], the definition of validity depended on which failure may occur. *(Strong) validity* was considered in the only case of crash failures, whereas a weak form of validity, *weak validity*, was distinguished if a failure could be a network failure. In fact, weak validity allows processes to abort a transaction (decide 0) even if none of them crashes and all of them vote to commit (propose 1), as long as there is a network failure. Definition 1 unifies validity and weak validity for presentation clarity and consistency with previous impossibility results.

Complexity measures. We consider two measures of complexity: the classical notion of *number of messages*, and the *number of message delays*, following the complexity study by Lamport of consensus [57]. The use of this complexity measure (message delays) is justified by the general context of an arbitrary (asynchronous) system (considering network-failure executions) in [57] and in Chapter 2. Unlike [1, 70], we do not consider the *number of steps* as a measure of time. In [1, 70], steps were defined for synchronous systems and do not fit a general asynchronous setting. (In addition, since steps and message delays measure time differently, even for the special case of synchronous NBAC, the results on number of steps in [1, 70] and our results on message delays are incomparable.)

Complexity results. The most closely related works to our results are (a) Dwork and Skeen's lower bound on the number of messages [1, 25, 26] and (b) Charron-Bost and Schiper's bound on the number of *rounds* [86] (of which the tightness was shown by Dutta et al. [90]). Both works focus on synchronous NBAC, while our study is for an arbitrary (asynchronous) system as well as an arbitrary combination of properties of NBAC. For the special case of synchronous NBAC, we are the first to present a tight lower bound on both the number of messages and that of message delays.

Compared with previous work, we generalize Dwork and Skeen's necessary and sufficient number of messages when at most $n-1$ processes may crash among n processes to an arbitrary number of crashes. Still for the special case of synchronous NBAC, we make Charron-Bost and Schiper's lower bound on time complexity more precise. They showed a lower bound of two rounds. In their model, one round consists of one send phase and one receive phase [86, 91]. Thus a lower bound of two rounds only says that the number of send phases *or* receive phases is at least two: it does not articulate which one. Combined with our tight lower bound of one message delay, we get a clear picture of the time complexity of synchronous NBAC protocols: a process can decide at the earliest by the end of the first message delay, and if so, it has to send messages before its decision. In other words, for any synchronous NBAC protocol, before any process decides, two send phases and one receive phase are necessary. (The tight two-round protocol of [90] needs at least two message delays and thus does not help to get such a picture.) Based on Charron-Bost and Schiper's two-round lower bound, Gray and Lamport [73] informally argued that two message delays should be optimal for indulgent atomic commit. However, by the model of rounds [86, 91], two rounds only imply a bound of one message delay.

2.6.2 Commit protocols

Two-phase commit (2PC) [22] distinguishes one process as the leader, which is a single point of failure in the sense that if it crashes, every other process is blocking in the fear of disagreement [16]. To circumvent this, Skeen [16] proposed three-phase commit (3PC), which adds one message delay and $2n-2$ messages over 2PC, along with a termination protocol. However, as several papers [71, 73] pointed out, 3PC (as well as many of its variants) does not solve the potential conflict between two backup leaders at the same time given by the termination protocol in crash-failure executions. Gray and Lamport [73] proposed *PaxosCommit* based on Paxos consensus [24] to solve the disagreement of non-unique leaders in network-failure executions. They also proposed *faster PaxosCommit* [73], an optimization of PaxosCommit, removing one message delay.[13] Both PaxosCommit [73] and faster PaxosCommit [73] solve indulgent atomic commit.

Faster PaxosCommit and one of our protocols INBAC solve the same problem yet differ significantly in how they achieve two message delays on a technical level. Faster PaxosCommit uses Paxos consensus in a non-black-box way in every execution. However, the design of INBAC follows immediately the proof of our lower bound results (Lemma 1 and Lemma 5 in Chapter 2) and hence does not invoke consensus in any nice execution.

2.6.3 Low-latency commit protocols with weak semantics

As observed in [92], 1-delay commit protocols proposed in [93, 94] assumes that all processes propose 1 before an execution starts. Jiménez-Peris et al. proposed a commit service which has the same latency as 2PC but allows a process to decide twice and differently. MDCC [95] proposed a variant of Paxos to coordinate transactions assuming all processes vote the same. Replicated Commit [96] executed also the Paxos protocol to commit transactions, assuming here that the votes from a majority of processes are already sufficient to commit. All these protocols solve different (and weaker) problems than classical atomic commit.

Calvin [97] eliminated the explicit commit protocol by using a deterministic locking scheme, using only one message to notify the decision; in fact, NBAC is only solved in failure-free executions where one message delay is (not surprisingly) sufficient. Helios [3] commits a distributed transaction if no conflict involving the transaction is detected across datacenters. Helios considers both failure-free and network-failure executions. In failure-free executions, optimal commit latency is achieved. In network-failure executions, the scheme proposed is far from the optimal in terms of complexity. Our INBAC protocol may be adapted to the needs of Helios with better complexity.

---

[13] Gray and Lamport [73] pointed out a possible optimization (without details) for an atomic commit protocol, MD3PC, proposed in [72]. Then MD3PC achieves the same number of message delays and messages as faster PaxosCommit. As MD3PC and faster PaxosCommit are equally efficient in nice executions, MD3PC is omitted from the discussion.

Table 2.4 – Complexity of Indulgent Atomic Commit, and Synchronous NBAC with f Crashes

	Indulgent atomic commit (our result)	Sync. NBAC (our result)	Sync. NBAC [1, 25, 26, 86, 90]
#delays	2	1	-
#messages	$2n-2+f$ (for $f \geq 2$)	$n-1+f$	$2n-2$ (when $f=n-1$) [1, 25, 26]

Table 2.5 – Complexity of INBAC, (n-1+f)NBAC, 1NBAC, 2PC, PaxosCommit and faster PaxosCommit

	1NBAC	(n-1+f)-NBAC	INBAC	2PC [22]	Paxos-Commit [73]	Faster Paxos-Commit [73]
#delays	1	$2f+n-1$	2	2	3	2
#messages	n^2-n	$f+n-1$	$2fn$	$2n-2$	$nf+2n-2$	$2fn+2n$ $-2f-2$
Atomic commit	Sync. NBAC	Sync. NBAC	Indulgent	Blocking	Indulgent	Indulgent

2.7 Concluding Remarks

We present the first systematic study of the (time and message) complexity of atomic commit. Table 2.4 summarizes the complexity results of previous work and our result. The number of message delays for previous work is left blank. We give a collection of lower bounds and matching protocols, by which we also close many questions on atomic commit. For indulgent atomic commit, the most robust among atomic commit problems we study, no (non-trivial) lower bound on the number of message delays or the number of messages was known until our work. Table 2.5 summarizes the time and message complexity of our INBAC, our two optimal synchronous NBAC protocols: (n-1+f)NBAC and 1NBAC, 2PC, PaxosCommit, and faster PaxosCommit.[14] Clearly, our (n-1+f)NBAC and 1NBAC protocols are the best regarding messages and message delays respectively. Among indulgent atomic commit protocols, in the special case of $f=1$, INBAC performs the best regarding both messages and message delays (for $n \geq 2$), and performs almost as efficiently as 2PC. Still among indulgent atomic commit protocols, PaxosCommit and our INBAC protocol show a tradeoff between time and message complexity: for $f \geq 2, n \geq 3$, PaxosCommit is better in messages while our INBAC protocol is better in message delays. On satisfaction of properties, our (n-1+f)NBAC and 1NBAC protocols and 2PC show a tradeoff between agreement and termination. 2PC guarantees agreement in an arbitrary (asynchronous) system (considering a network-failure execution) but not termination even if only crash failures are possible. On the other hand, (n-1+f)NBAC and 1NBAC terminate despite f crashes but an execution in an arbitrary (asynchronous) system may violate *agreement* (due to the use of no-ops for (n-1+f)NBAC and due to the optimal delay

---

[14] To enable a fair comparison, we assume that each protocol involves only the n processes which vote and decide, and each protocol starts when n processes send messages spontaneously. Thus 1 delay from 2PC and 2 delays from PaxosCommit and faster PaxosCommit are removed respectively, while $n-1$ messages are removed from the three protocols respectively from their original counting.

for 1NBAC respectively).

Some questions remain open. For example, for the tradeoff between time and message complexity, the optimal number of messages given greater than two message delays for indulgent atomic commit is not yet clear (although we close the question for two message delays).

3 The Complexity of Causal Transactions[1]

3.1 Introduction

Transactional distributed storage systems have proliferated in the last decade: Amazon's Dynamo [34], Facebook's Cassandra [99], Linkedin's Espresso [100], Google's Megastore [101], Walter [31] and Lynx [102] are seminal examples, to name a few. A lot of effort has been devoted to optimizing their performance for their success heavily relies on their ability to execute transactions in a fast manner [103]. Given the difficulty of the task, two major "strategic" decisions have been made. The first is to prioritize *read-only transactions*, which allow clients to read multiple items at once from a consistent view of the data store. Because many workloads are read-dominated, optimizing the performance of read-only transactions has been considered of primary importance. The second is the departure from strong consistency models [104, 105] towards weaker ones [106, 107, 44, 108, 109, 110]. Among such weaker consistency models, *causal consistency* has garnered a lot of attention for it avoids heavy synchronization inherent to strong consistency and can be implemented in an always-available fashion in geo-replicated settings (i.e., despite partitions), while providing sufficient semantics for many applications [35, 36, 111, 38, 39, 40, 41].

Despite the observation that two-round causal transactions double latency and halve throughput compared with an even weaker consistency model, eventual consistency [36], causal read-only transactions in most transactional storage [35, 36, 37, 38, 40, 41] can induce more than one-round communication. Even the performance of highly optimized state-of-the-art causally consistent transactional storage systems has revealed disappointing. The recent COPS-SNOW system [44] implements "fast" read-only transactions, i.e., transactions that complete in one round of interaction between a client seeking to read the value of an object and the server storing it. This design makes the assumption that write operations are supported only outside the scope of a transaction.[2] COPS-SNOW is designed to outperform COPS [35] and its successor Eiger [36]. Both COPS and Eiger design non-fast read-only transactions yet the

---

[1] Preprint version of an article under submission: Diego Didona, Rachid Guerraoui, Jingjing Wang and Willy Zwaenepoel. *"Distributed Transactions: Dissecting the Nightmare"* [98]

[2] Under this assumption, a single-object write and a transaction that only writes to one object are equivalent.

evaluation of COPS-SNOW reveals that the latency of COPS-SNOW is sometimes higher than that of COPS/Eiger [44]. In fact, the benefits and implications of many designs are unclear, and their overheads with respect to systems that provide no consistency are not well understood.

In this chapter, we investigate the overheads from a theoretical perspective with the aim of identifying possible and impossible causal consistency designs in order to ultimately understand their implications. We prove two impossibility results.

- First, we prove that no causally consistent system can support read-write transactions and implement fast read-only transactions. This result unveils a fundamental tradeoff between semantics (support for read-write transactions) and performance (latency of read-only transactions).
- Second, we prove that fast read-only transactions must be "visible", i.e., their execution updates the states of the involved servers. The resulting overhead increases resource utilization, which sheds light on the inherent overhead of fast read-only transactions and explains the surprising result in the evaluation of COPS-SNOW.

The main idea behind our first impossibility result is the following. One round-trip message exchange disallows multiple servers to synchronize their responses to a client. Servers need to be conservative and return possibly stale values to the client in order to preserve causality, with the risk of jeopardizing progress. Servers have no choice but communicate outside read-only transactions (i.e., helping each other) to make progress on the freshness of values. We show that such message exchange can cause an infinite loop and delay fresh values forever. The intuition behind our second result is different. We show that a fast read-only transaction has to "write" to some server for otherwise, a server can miss the information that a stale value has been returned for some object by the transaction (which reads multiple objects), and can then return a fresh value for some other object, violating causal consistency.

At the heart of our results lies essentially a fundamental tradeoff between causality and (eventual) freshness of values.[3] Understanding this tradeoff is key to paving the path towards a new generation of transactional storage systems. Indeed, the relevance of our results goes beyond the scope of causal consistency. They apply to any consistency model stronger than causal consistency, e.g., linearizability [104, 105] and strict serializability [112, 113], and are relevant also for systems that implement hybrid consistency models that include causal consistency, e.g., Gemini [114] and Indigo [115].

The rest of this chapter is organized as follows. Section 3.2 presents our model and definitions. Section 3.3 presents the impossibility of fast read-only transactions. Section 3.4 presents the impossibility of fast invisible read-only transactions (with restricted semantics, where writes are outside the scope of a transaction). Section 3.6 extends the two impossibilities to partially

---

[3] This tradeoff is different from the traditional one in distributed computing between ensuring linearizability (i.e., finding a linearization point) and ensuring wait-freedom, which refer to both rather strong properties.

replicated storage systems. Section 3.5 discusses alternative protocols that circumvent the impossibility results. Section 3.7 discusses related work. Section 3.8 discusses the implications of our impossibility resutls to some existing causal consistency designs and concludes the chapter.

3.2 Model and Definitions

3.2.1 Model

We assume an arbitrarily large number of *clients* $C_1, C_2, C_3, \ldots$ (sometimes also denoted by C), and at least two *servers* P_X, P_Y (sometimes also denoted by P). Clients and servers interact by exchanging messages. We consider an *asynchronous* system where the delay on message transmission is finite but arbitrarily large, and there is no global clock accessible to any process. Clients and servers have access to their local clocks; however, there can be arbitrary clock drift between any two local clocks. Communication channels do not lose, modify, inject, or duplicate messages, but messages could be reordered.

A *storage* is a finite set of objects. Clients read and/or write objects in the storage via *transactions*. Any transaction T consists of a read set R_T and a write set W_T on an arbitrary number of objects (R_T or W_T could be empty). We denote T by (R_T, W_T). If T is read-only or write-only, we denote T simply by R_T or W_T respectively. For the purpose of establishing results on fast transactions (which are defined later), we focus on such transactions that can issue all operations simultaneously, as illustrated in Figure 3.1. For example, we do not consider the transaction model where a transaction must first read and then write upon the result of the read, which intuitively falls out of the scope of fast transactions. Clearly, the transactions which we focus on do not repeatedly read or write as well; therefore, the objects read by R_T are mutually different; so are the objects written by W_T. Thus a client *starts* a transaction by issuing all operations of the transaction to the storage. When a client returns from transaction T, the client returns a value for each read in R_T and ok for each write in W_T. We say that a client *ends* a transaction when the client returns from the transaction. Every transaction ends.

Here we note that when we later refer to the construction of an execution (of a few specified transactions), we mean a sequence of message exchange events between clients and servers in the asynchronous system (by which the transactions are executed). If we say some event eventually occurs given a prefix of message exchange events, then in every suffix, there is some finite time when the event occurs. This finite time instant can depend on the sequence of events and is not assumed to be known a priori (although for convenience, we might give it a notation).

The storage is implemented by servers. For simplicity of presentation, we first assume that each server stores a different set of objects and the set is disjoint between servers and then we show in Section 3.6 how our results apply to the non-disjoint case, or partially replicated storage systems in general. Every server receiving a request from a client responds. A server

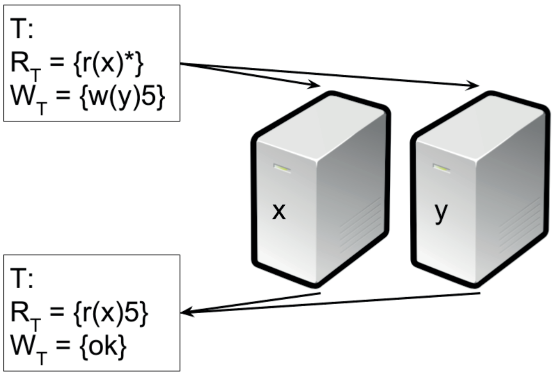


Figure 3.1 – An example transaction

sends a message to a client only if the client requests the server via a transaction and has not returned yet from the transaction; no server receives requests for objects not stored on that server. Naturally, a server that does not store an object stores no information on values written to that object; due to arbitrary clock drift, we consider a client request oblivious to the client's local clock, i.e., without any knowledge of the local clock. Moreover, we assume an implementation where to respond to a read request, a server returns one and only one value which has been written to the object in question.

3.2.2 Causality

We consider a transactional storage that ensures *causality* in the classical sense of [42, 43], which we recall below.

The *local history* of client C_i, denoted L_i, is a sequence of start and end events of the transactions which C_i requests. We assume, w.l.o.g., that any client starts a new transaction after the client has ended all previous transactions, i.e., any client is sequential. Hence any local history L_i can be viewed as a sequence of transactions as well.

We denote by $r(x)v$ a read on object x which returns v, by $r(x)*$ a read on object x for an unknown return value (with symbol $*$ as a place-holder), and by $w(x)v$ a write of v to object x. For simplicity, we assume that every value written is unique. (Our results hold even when the same values can be written.) Definition 6 captures the program-order and read-from causality relation [42]. Assume that each object is initialized with a special symbol $\perp$. (Thus a read can be $r(x)\perp$.)

Definition 6 (Causality [42, 43])**.** Given local histories $L_1, L_2, L_3 \ldots$, for any two transactions T_a, T_b, we say that T_a causally precedes T_b, which we denote by $T_a \rightsquigarrow T_b$, if (1) $\exists i$ such that T_a is before T_b in L_i; or (2) $\exists v, x$ such that $\alpha = w(x)v \in W_{T_a}$ and $\beta = r(x)v \in R_{T_b}$; (3) $\exists T_c$ such that $T_a \rightsquigarrow T_c$ and $T_c \rightsquigarrow T_b$.

Definition 7 ((Causally) legal transactions [42, 43])**.** Given local histories $H = L_1, L_2, L_3, \ldots$, we say that client C_i's history is legal and respects causality if we can totally order all transactions that contain a write in H and all transactions in L_i, such that

1. For every transaction $T \in L_i$, for every read $r = r(x)* \in R_T$,
    (a) If r returns a non-$\perp$ value v and if T_x is the last transaction that contains a write on object x and precedes T, then the write on x in T_x is $w(x)v$;
    (b) If r returns $\perp$, then no transaction that precedes T contains $w(x)*$;
2. For any T_a, T_b such that $T_a \rightsquigarrow T_b$, T_a is ordered before T_b.

Definition 8 (Causal consistency [42, 43])**.** We say that storage cc is causally consistent if for any execution of clients with cc, each client's local history is legal and respects causality.

As noted by Raynal et al. [43], when every transaction contains a single read or a single write, then the definition of causal consistency is identical to the definition of causal memory in [42]. For two writes α, β in two transactions T_a, T_b respectively, if $T_a \rightsquigarrow T_b$, then we also say that $\alpha \rightsquigarrow \beta$ and α causally precedes β.

3.2.3 Progress

Progress is necessary to make any storage useful. Without progress, we may devise a trivial implementation which returns $\perp$ for a read if a client has not written to the object in question, and the most recent value written by C otherwise. The implementation trivially satisfies causal consistency.

To ensure progress, we require any value written to be eventually *visible*. While rather weak, this definition is strong enough for our impossibility results, which apply to stronger definitions. If compared with the definitions of eventual consistency [111, 116], the definition of eventual visibility below is not conditioned on the absence of new writes or based on the occurrence of underlying message exchange events, but focuses on clients' progress in reads. Different from *convergence* property [35] which focuses on transactions that are not causally related, the definition of progress here is decoupled from the definition of causal transactions. In addition, time $\tau_{x,v}$ in Definition 9 only notates eventually when a write or a value is visible rather than imply its exact clock-time a priori.

As assumed before, Definition 9 is based on the setting where each server stores a different set of objects and the set is disjoint between servers. In this setting, all writes of the same object thus happen on the same server; thus Definition 9 also assumes that the last writer of the same object wins, which is the most natural rule here, when deciding progress. For example, if (the transaction that includes) $w(x)a$ ends before (the transaction that includes) $w(x)b$ starts and for any arbitrary time T, some read which starts after T returns a, then Definition 9 is violated. We adapt the definition later to cover the case where multiple servers may store the same object and the writes of the same object can happen on different servers.

Definition 9 (Eventual visibility)**.** If we say a write $w = w(x)v$ of transaction T is eventually visible (or v is eventually visible as unique written values are assumed), then there exists some finite time $\tau_{x,v}$ such that for any transaction T_{rx} which starts no earlier than $\tau_{x,v}$ and has $r(x)v_{new} \in R_{T_{rx}}$, then either $v_{new} = v$ or $w(x)v_{new} \in W_{T_{wx}}$ where transaction T_{wx} returns no earlier than T starts.

Definition 10 (Progress)**.** A (causally consistent) storage guarantees progress if every write is eventually visible.

3.3 The Impossibility of Fast Transactions

In this section, we present and prove our first theoretical result, Theorem 7. We first define formally the notion of *fast* transactions. In short, a fast transaction is one of which each operation executes in (at most) one communication round between a client and a server.

3.3.1 Definitions

Fast transactions

Definition 11 below focuses on the message exchange in the presence of arbitrary, indefinitely long message delay between servers. Clearly, if in the presence of arbitrary, indefinitely long message delay between servers, every transaction can be fast (by a protocol that finishes communication in at most one round-trip), then every transaction can be fast when message delay is known or upper bounded by a known value.[4]

Definition 11 (Fast transaction)**.** We say that a transactional storage provides fast transaction T if for any client C, C's invocation I of T is fast.

If C's invocation I of T is fast, then no matter what execution precedes I, the following execution of T is allowed:

- C sends at most one message to any server P and receives at most one message from any server P;
- If C sends a message to server P, then after the reception of that message, any message which P sends to a server is delayed arbitrarily; moreover, after the reception of that message, P receives no message from any server;
- Eventually C still returns I.

---

[4] A protocol can be designed to communicate more among servers when the servers are confident about an upper bound on the message delay in order to, for example, return fresher values for transactional reads. Such protocol still satisfies Definition 11 if it falls back to finish in one communication round when the servers find the upper bound on message delay is violated.

In the last condition of Definition 11, the eventual return of a client refers to two possibilities: either the client needs not to receive a message from some server to return, or the server eventually replies to the client. Thus Definition 11 excludes implementations where a server waits for the reception of messages from another server (whether the server is one which C sends a message to or not) to reply to a client. Definition 11 allows multiple clients to request the same server so that the duration of two transactions (at least one of which is fast) invoked by different clients can overlap. A final remark is that in Definition 11, if client C sends a message to server P, then no matter what execution precedes the reception of that message at P, the execution of transaction T above should be allowed.

One version

As mentioned in Section 3.2, we assume that a server returns one and only one value for a transactional read, a property which we formally define below as one-version. In this chapter, a version of an object is one value written to the object. When we mention two or more versions, these versions are written to the same object if we do not state otherwise. To ensure that every implementation with one-version property cannot work around the limit on the number of versions, the formal definition considers the implementation as a curious "adversary" whose goal is to output some version other than the allowed one version.

Consider all possible implementations. If some implementation instructs some process P to calculate some version at some point, then w.l.o.g., the version is the result of a certain algorithm which takes the messages and events at P before this point. W.l.o.g., any execution can be considered as a set of events (with their corresponding messages). Thus we may model the curious "adversary" by an algorithm which takes a subset of messages and events in a given execution (as these messages and events appear only at the process in question) and outputs a set of versions. To model that the curious "adversary" indeed outputs versions (rather than arbitrary values), we must bind each version in the output to the write in the given execution. We name such "adversary" as successful algorithms and define it in Definition 12. As the transactional storage considered here is independent from a specific application, we assume those implementations to be independent from the specific values written. Hence the binding in Definition 12 covers all possible implementations.

Definition 12 (Successful algorithms)**.** Consider any algorithm, denoted by $\mathscr{A}$, whose input is some information i_E (events and messages) of execution E. The output of $\mathscr{A}$ is denoted by $\mathscr{A}(i_E)$. We say that $\mathscr{A}$ is *successful*

- If $v \in \mathscr{A}(i_{E^v})$, then in E^v, $\exists a$, $w(a)v$ occurs; and
- For any value u, let E^u be the resulting execution from E^v where $w(a)v$ is replaced by $w(a)u$ (and the corresponding messages are replaced accordingly). Then $u \in \mathscr{A}(i_{E^u})$.

Since any local computation of a client (server) is based on all message exchange events so far at the client (server),[5] Definition 12 and the definitions that follow represent the potential return value of a transactional read as an output of a client's local computation based on all message exchange events at the client until the read (inclusive). Therefore with more messages received at the client side, the client is able to infer more values written to any object. However, one-version property focuses on the messages sent by servers during a transaction. This leads to Definition 13, which counts the increment of versions brought by the increment of messages received.

Definition 13 (Versions revealed)**.** Consider execution E, client C and C's invocation I of some transaction. Denote by M any non-empty subset of message receiving events that occur at C (including message contents) during I. We say that M *reveals* $(n_2 - n_1)$ version(s) of an object a if

- Among all successful algorithms whose input is $v_{C,I}$, n_1 is the maximum number of values in the output that are also values written to a before the start of I;
- Among all successful algorithms whose input is $v_{C,I}$ and M, n_2 is the maximum number of values in the output that are also values written to a before the end of I;

where $v_{C,I}$ is C's *view*, or all events that have occurred at C (including the message content if an event is message receiving), before the start of I.

Finally, Definition 14 combines Definition 12 and Definition 13 and defines formally one-version property. As Definition 13 shows, we let the curious "adversary" try its best in outputting versions. Then in Definition 14, we enforce that despite such effort, only one version can be obtained for each object in question for a given transaction. In this sense, one-version property is the property of messages and events, rather than the client-side algorithm that calculates the versions. As a result, by Definition 14, we define the property in a way independent from message formats. For example, if messages m_1 and m_2 are from two different servers P_X and P_Y and $m_1 = (x, \text{first 8 bits of } z \text{ XOR } c)$, $m_2 = (y, \text{other bits of } z \text{ XOR } c)$, where z is a value written to another object Z, then (m_1, m_2) can return more values x, y, z than expected. Such messages should be excluded and are indeed so by Definition 14.

Definition 14 (One-version property)**.** Consider any execution E, any client C and C's invocation I of an arbitrary transaction T with non-empty read set R. For any non-empty set of servers A, let $\Lambda_{I,A} = R \cap \{\text{objects stored on } P | \forall P \in A\}$ and denote by $M_{I,A}$ the events of C receiving messages from any server in A (including message contents) during I. Then an implementation satisfies one-version property if

- $\forall E, \forall I, \forall A$, $M_{I,A}$ reveals at most one version for each object in $\Lambda_{I,A}$, and no version of any object not in $\Lambda_{I,A}$; and

---

[5] A specific protocol can surely take only a subset of these events, but cannot take more as input.

- $\forall E, \forall I$, when A includes all servers, then $M_{I,A}$ reveals exactly one version for each object in R, and no version of any object not in R.

(If $M_{I,A}$ reveals exactly one version of an object a, we may also specify the version v and say that $M_{I,A}$ reveals v.)

One final remark is that one-version property is defined in a general way, independent from fast transactions. Consider an implementation of transaction which contains intuitively one round but rather than sending a single message as Definition 11, the server sends several messages to the same client. If each of these messages reveals one version, then our impossibility results can be circumvented. The one-version property here however is defined on all message receiving events during a transaction, and thus covers such intuitively one-round protocol.

3.3.2 Result

Theorem 7 says that it is impossible to implement fast transactions (even if just read-only ones are fast).

Theorem 7. *A causally consistent transactional storage that supports transactions which can read and/or write multiple objects does not provide fast read-only transactions.*

The intuition behind Theorem 7 is the following. Consider a server P_X that stores object X and a server P_Y that stores object Y. Suppose that a transaction writes some new values to X and Y and another transaction reads X and Y. There is a risk of violating causality for P_Y if P_X returns an old value to the read-only transaction; furthermore, in this case, P_Y must return an old value (to the same transaction). The statement is also true if we swap P_X and P_Y. By the definition of fast transactions, P_X and P_Y must be able to avoid the risk without help from other servers and thus have to be conservative, i.e., returning old values if there is a risk. As a result, P_X and P_Y take turns in creating causality violation risks for each other, and preventing each other from returning new values forever, jeopardizing thereby progress. Below we first sketch our proof of Theorem 7 and then present the full proof.

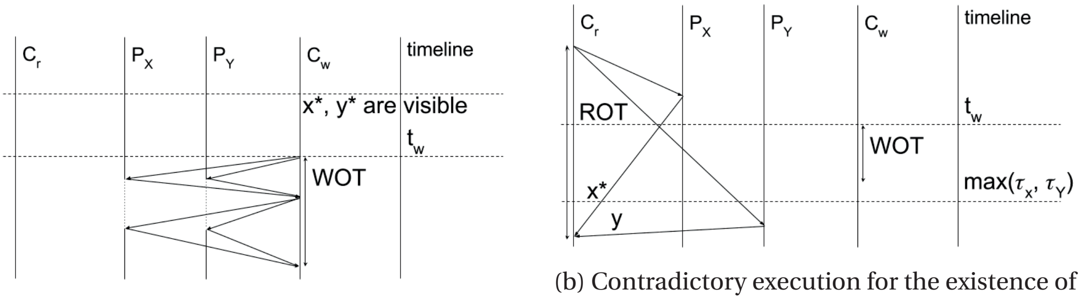
3.3.3 Proof by induction

The proof of Theorem 7 is by construction of a contradictory execution E_{imp} which, to satisfy causality, contains an infinite number of messages the reception of which is necessary for some value to be visible. The reception of an infinite number of messages violates progress. As illustrated in Figure 3.2a, some non-$\perp$ values of X and Y are already visible before our construction of E_{imp}; then client C_w issues transaction $WOT = (w(X)x, w(Y)y)$ which starts at time t_w; since t_w, WOT is the only executing transaction. We make no assumption on the distributed protocol of WOT.

We show the number of messages is infinite by showing that no matter how many k messages have been sent and received, an additional message is necessary for x and y to be visible. Let $m_0, m_1, \ldots, m_{k-1}, m_k$ be the sequence of k messages. Then $\forall k \geq 1$, the $(k+1)$th message m_{k+1} is sent after m_k is received, while m_{k+1} must be received before x and y are visible. Our detailed proof proves the statement for each natural number k and thus shows the number of messages goes to infinity. As every message is sent after previous messages are received and messages are not received instantaneously, the delay to return x or y accumulates and progress (Definition 10) is violated.

As we make no assumption on the underlying distributed protocol of transactions, the communication between P_X and P_Y can be via a third server or not. Definition 15 on the precedence relation of two messages unifies the description of the two types of communication above. Following Definition 15, we simply say that P_X (P_Y) sends a message which *precedes* some message that arrives at P_Y (P_X) in the proofs hereafter.

Definition 15. Message m_1 *precedes* message m_2 if (1) $m_1 = m_2$, or (2) a process sends m_2 after it receives m_1 or (3) there exists message m such that m_1 precedes m and m precedes m_2.



(a) Construction of E_{imp}

(b) Contradictory execution for the existence of the first two messages

Figure 3.2 – Illustration of E_{imp} and the base case

3.3.4 Construction of E_{imp}

The construction of E_{imp} is based on the following notations and execution E_{prefix}. Recall that we denote by P_X the server which stores object X, and P_Y the server which stores object Y. Let E_{prefix} be any execution where X and Y have been written at least once and some non-$\perp$ values of X and Y are visible. Let x^* and y^* be the visible values respectively. Suppose that at time t_{start}, x^* and y^* are visible in E_{prefix}.

Starting from t_{start}, we construct execution E_{imp}. In E_{imp}, client C_w does transaction $WOT = (w(X)x, w(Y)y)$ which starts at some time $t_w > t_{start}$, while all other clients do no transaction. For E_{imp}, since t_w, WOT is the only transaction. The construction continues as long as at least one between x and y is not visible.

As mentioned in Section 3.3.3, the construction adds one message at a time (except for the first two messages). For any positive number k, we construct E_{imp} such that k specific messages are sent and received after t_w , we prove that (S) before x and y are visible, another message, the $(k+1)$th message must be sent and received (after the reception of previous k messages) and therefore, the construction of E_{imp} must continue. If we consider statement (S) as a property $P(k)$, then we essentially prove that $P(k)$ holds for all natural numbers $0,1,2,3,\ldots$.

Our proof naturally goes by induction. Proposition 1 presents the base case and Proposition 2 presents the inductive step on case k. As the base case shows two messages are sent, we index the sequence of messages starting from 0: $m_0, m_1, \ldots, m_{k-1}, m_k$ and then the first inductive step is from case 1 to case 2. As shown in Proposition 1, let P_X and P_Y send m_X and m_Y after t_w that precede some message which arrive at P_Y and P_X respectively. We define m_0 and m_1 as follows: one server between P_X and P_Y sends m_0 before receiving any message which is preceded by m_1 for $\{m_0, m_1\} = \{m_X, m_Y\}$.

It is easy to see that these k messages for any positive number k are not k arbitrary messages but specifically defined by the proof. Therefore, the proof of Proposition 1 and that of Proposition 2 actually belong to the construction of E_{imp} (at least partially). The proofs are, however, deferred to later sections after some helper proposition and helper lemmas for a better presentation of the complete proof.

Proposition 1 (Additional message in the base case)**.** *After t_w, any $P \in \{P_X, P_Y\}$ must send at least one message that precedes some message which arrives at Q for $\{P, Q\} = \{P_X, P_Y\}$.*

Proposition 2 (Additional message in case k)**.** *In E_{imp}, $m_0, m_1, \ldots, m_{k-1}$ have been sent. Let D_{k-1} be the source of m_{k-1}. Let $\{D_{k-1}, D_k\} = \{P_X, P_Y\}$. Let T_{k-1} be the time when the first message preceded by m_{k-1} arrives at D_k. After T_{k-1}, D_k must send at least one message m_k that precedes some message which arrives at D_{k-1}.*

3.3.5 Proof of Theorem 7

Our proof consists of three steps. First, we note that to prove Proposition 2 for each positive number $k+1$, we do not only need the correctness of Proposition 2 but also the correctness of another proposition (Proposition 3) for each k. The latter proposition is a property of the construction of E_{imp} in case k and does not add any message to the construction.

Then we prove two helper lemmas, Lemma 7 and Lemma 8, in order to prove all propositions. Lemma 8 is helpful for the proof of both the base case and case k, and thus proven additionally to avoid repetition, while Lemma 7 shows a property of write-only transactions. As Lemma 8 is based on Lemma 7, we prove the latter first.

Finally, we prove Theorem 7. Our complete proof necessarily shows the construction of an infinite number of necessary messages by induction (through our proof of the base case Proposition 1, the inductive step from case k to case $k+1$ Proposition 2 and Proposition 3), and relates the reception of the sequence of this infinite number of messages to the violation of progress property.

Another proposition in case k

To help prove Proposition 2 for case $k+1$, Proposition 3 shows that in case k, if at some point, some client reads X and Y in one transaction, then the client cannot return the values x and y written by WOT. Proposition 3 is intuitively necessary, as it relates our induction to the eventual visibility of x and y.

Proposition 3 shares the same notations as Proposition 2, for the sequence of messages $m_0, m_1, \ldots, m_{k-1}, m_k$, time T_k, and the source of message D_{k-1}. The client C_r and the read-only transaction ROT issued by C_r are explained below.

Proposition 3 (Case k)**.** *In E_{imp}, $m_0, m_1, \ldots, m_{k-1}, m_k$ have been sent. Then for any t in $[T_{k-1}, T_k)$, if C_r starts ROT at some time in $[T_{k-1}, t)$ and $t_{D_{k-1}} = t$, then ROT may not return x or y.*

Client C_r is a client that requests no transaction if C_r does not request ROT. We note that in the construction of E_{imp}, C_r indeed requests no transaction. Let $ROT = (r(X)*, r(Y)*)$. By Definition 11, for ROT, we schedule messages such that every message which C_r sends to either $P \in \{P_X, P_Y\}$ during ROT arrives at the same time t_P at P. After t_P and before P has sent one message to C_r (during ROT), P receives no message and any message sent by P to a process other than C_r is delayed to arrive after ROT ends. For either P, we denote these messages which P sends to C_r after t_P (during ROT) by $m_{resp,P}$.

In fact, in the later statements and proofs (especially for Lemma 8 and its proof), ROT refers to a read-only transaction that reads X and Y in general and C_r is its client which does not request any other transaction if we do not explicitly say so. The message schedule of ROT (as well as the notations that follow) is the same as mentioned above to take advantage of the property of fast transactions.

Helper lemmas

Lemma 7. *In E_{imp}, no write (including writes in a transaction) occurs other than WOT since t_w. If some client C_r requests ROT, then ROT returns x if and only if ROT returns y.*

Proof of Lemma 7. By contradiction. Suppose that for some execution E_{imp} and some read-only transaction ROT, ROT returns (x^*, y), or (x, y^*). By symmetry, we need only to a contradiction for the former.

As ROT returns (x^*, y), by causal consistency, for C_r, there is serialization $\mathscr{S}$ that orders C_r's transaction ROT and all transactions including a write such that the last preceding writes of

X and Y before ROT in $\mathscr{S}$ are $w(X)x^*$ and $w(Y)y$ respectively. Therefore any $\mathscr{S}$ must order WOT before $w(X)x^*$. By progress, x and y are eventually visible. W.l.o.g., let $\tau_{(X,Y),(x,y)}$ be some time (possibly in the future) when x and y are visible. If C_r requests another read-only transaction $ROT_2 = (r(X)*, r(Y)*)$ after $\tau_{(X,Y),(x,y)}$, then as no write occurs other than WOT since t_w, ROT_2 returns (x, y).

Now that C_r requests two read-only transactions, ROT_2 after ROT, $\mathscr{S}$ must include both transactions and order ROT_2 after ROT. As a result, the last preceding writes of X and Y before ROT_2 in $\mathscr{S}$ cannot be $w(X)x$ and $w(Y)y$ respectively, contradictory to the property of causal consistency. □

Lemma 8 (Communication prevents latest values)**.** *Suppose that E_{imp} has been extended to some time A and there is no other write than contained in WOT since t_{start}. Let $\{P, Q\} = \{P_X, P_Y\}$ where P can be either P_X or P_Y. Denote by time $B > A$ when one specific event*[6] *occurs in E_{imp}. Given P, assume that if C_r starts ROT at some time in $[A, t_P)$, then for any $t_P \in [A, B)$, ROT may not return x or y (no matter how messages are scheduled after time A).*[7] *We have:*

1. *After B, P must send at least one message which precedes some message that arrives at Q;*
2. *Let t be the time when Q receives the first message which is preceded by some message which P sends after B. For any $\tau \in [B, t)$, if C_r starts ROT at some time in $[B, \tau)$ and $t_Q = \tau$,*[8] *then ROT may not return x or y (no matter how messages are scheduled after time B except for time t as well as its precedence).*

Proof of Lemma 8. We prove both statements by contradiction. Let us start with the proof of the first statement by contradiction. Suppose that after B, P sends no message that precedes any message that arrives at Q. In the proof by contradiction of the first statement, we construct two executions: E_1 and E_2 where E_2 is first a mere copy of E_{imp} to time A and ensures the same event to occur at time B, and continues without any transaction until both x and y are eventually visible. Suppose that x and y are visible after time t_{ev} in E_2. Then based on the assumption in Lemma 8, $t_{ev} \geq B$. We continue the construction of E_2 by C_r requesting ROT after t_{ev}. By progress, no matter how the messages of ROT are scheduled, C_r returns (x, y) to ROT. Recall notations $m_{resp,P}$ and $m_{resp,Q}$ previously defined. The client-side algorithm $\mathscr{A}$ of C_r to output the return value of ROT is a successful algorithm. In E_2, given $m_{resp,P}$ and $m_{resp,Q}$ (no matter when they are received and what are their contents), $\mathscr{A}$ outputs (x, y). Then by one-version property, $m_{resp,Q}$ reveals one and only one between x and y. (Otherwise, if $m_{resp,Q}$ can reveal another value v other than x and y, then we can obtain a successful algorithm which outputs x, y, v given $m_{resp,P}$ and $m_{resp,Q}$, violating one-version property.)

---

[6] For the presentation of this lemma, it is not necessary to know the exact event.

[7] Recall notation t_P and the message schedule of ROT previously defined. The message schedule in the assumption can be arbitrary after A as long as the message schedule of ROT is respected.

[8] If needed, by the asynchronous communication, we may delay t after ROT ends to respect the message schedule of ROT that Q receives no message during ROT.

Let t_s be the latest time before B such that P sends a message that precedes some message which arrives at Q in E_1. If $t_s < A$ or t_s does not exist, then we take $t_s = A$. We now turn to construct E_1 (and we resume the construction of E_2 later, which needs not to be complete for this proof). We construct E_1 based on E_2 starting from t_s. We delay any message which P sends after t_s in E_1. If S is a server which receives a message preceded by any message sent from P after t_s in E_2, then we let t_S be the time when S first receive such message in E_2 and delay any message sent from S after t_S in E_1. In E_1, C_r starts ROT after t_s and before B. Recall notations t_P and t_Q in the message schedule of ROT. In E_1, we schedule message events of ROT such that $t_P \in [A, B)$. Furthermore, in E_1 and E_2, we schedule message events of ROT such that t_Q takes the same value greater than t_{ev}.

According to our definition of t_s, after t_s, P does not send any message which precedes some message that arrives at Q in E_2. As we delay the messages which P sends after t_s in E_1, thus before t_Q, Q is unable to distinguish between E_1 and E_2. After t_Q (inclusive), according to the message schedule of ROT, by the time when Q sends one message to C_r during ROT, Q is still unable to distinguish between E_1 and E_2. By Q's indistinguishability between E_1 and E_2, in E_1, $m_{resp,Q}$ is the same content as in E_2 and reveals one and only one between x and y. W.l.o.g., let $m_{resp,Q}$ reveal x.

By the definition of E_{prefix}, the return value of ROT in E_1 cannot include $\perp$. As C_r has not requested any transaction before, then in E_1, the return value depends solely on $m_{resp,P}$ and $m_{resp,Q}$. Therefore, by one-version property, $\mathscr{A}$ cannot output a value other than x for object X. As a result, ROT returns x in E_1. A contradiction to the assumption that if $t_P \in [A, B)$ (which matches E_1), then ROT may not return x or y.

We now prove the second statement by contradiction. The proof by contradiction is similar to that of the first statement. Suppose that in some E_{imp}, for some $\tau \in [B, t)$, some ROT such that $t_Q = \tau$ returns x or y. By Lemma 7, ROT returns (x, y). With an abuse of notations, let t_s be the latest time before B such that P sends a message that precedes some message which arrives at Q in E_{imp}. If $t_s < B$ or t_s does not exist, then we take $t_s = B$.

We construct E_{old} based on E_{imp} by C_r requesting ROT at earlier time. Furthermore, E_{old} is the same as E_{imp} until C_r starts ROT. In E_{old}, the message schedule of ROT satisfies $t_P \in (t_s, B)$ and $t_Q = \tau$. All messages sent by P after t_s are delayed. If S is a server which receives a message preceded by any message sent from P after t_s in E_{imp}, then we let t_S be the time when S first receive such message in E_{imp} and delay any message sent from S after t_S in E_{old}. Thus Q is unable to distinguish between E_{old} and E_{imp} by the time when Q sends one message to C_r (for ROT). Since ROT returns (x, y) in E_{imp}, then $m_{resp,Q}$ reveals x or y in E_{old}. By the definition of E_{prefix}, the return value of ROT in E_{old} cannot include $\perp$. As C_r has not requested any transaction before, then in E_{old}, the return value depends solely on $m_{resp,P}$ and $m_{resp,Q}$, which must include x or y. A contradiction to the assumption in the statement of the lemma. □

Figure 3.3 – Timeline in Lemma 8

As illustrated in Figure 3.3, Lemma 8 is based on an assumption that before B, old versions are returned to ROT and shows that B can be prolonged to time t. However, Lemma 8 makes no assumption on the underlying distributed protocol of WOT and the detailed schedule of message events except for some explicit references in the statement of Lemma 8.

Full proof

What remains is the complete proof of Theorem 7, which proves Proposition 1, Proposition 3 and Proposition 2 by induction, and relates the conclusion of the induction, i.e., the reception of the sequence of this infinite number of messages, to the violation of progress property.

Proof of Theorem 7. By mathematical induction, we start with the base case, i.e., Proposition 1 and Proposition 3 for $k = 1$. Let $A = t_{start}$ and let $B = t_w$. By symmetry, we need only to prove Proposition 1 for $P = P_X$. To start with, we show that given P, for any $t_P \in [A, B)$, if C_r starts ROT before t_P, then ROT may not return x or y, in order to apply Lemma 8 later. As illustrated in Figure 3.2b, at t_P, as WOT has not yet started. Since by the time when P sends one message to C_r during ROT, P receives no message, thus $m_{resp,P}$ cannot reveal x or y. By one-version property, $m_{resp,P}$ reveals at most one version v_1 of X and $\{m_{resp,P}, m_{resp,Q}\}$ also reveals at most one version v_2 of X. Therefore $v_1 = v_2 \neq x$. As C_r has requested no transaction before, the return value of ROT solely depends on $m_{resp,P}$ and $m_{resp,Q}$. As the client-side algorithm of C_r for the return value of ROT is a successful algorithm, ROT returns $v_1 = v_2 \neq x$ for object X. (Due to E_{prefix}, ROT cannot return $\perp$.) Then by Lemma 7, ROT may not return x or y. Figure 3.2b illustrates the execution that contradicts Lemma 7. Thus Lemma 8 applies. By Lemma 8, after $B = t_w$, P must send at least one message that precedes some message that arrives at Q, which concludes that Proposition 1 is true for either $P \in \{P_X, P_Y\}$. Following Proposition 1, recall the definition of m_0 and m_1. We construct E_{imp} by letting m_0 and m_1 be sent. Recall that T_1 is the time when the first message preceded by m_1 arrives at D_0. According to Lemma 8, for any $t \in [B, T_1)$, if C_r starts ROT at some time in $[B, t)$ and $t_{D_0} = t$, then ROT may not return x or y, which proves Proposition 3 for $k = 1$.

We continue with the inductive step from case k to case $k+1$. We assume that Proposition 2 and Proposition 3 are correct for case k and prove that Proposition 2 and Proposition 3 are correct for case $k+1$. Let $A = T_{k-1}$, $B = T_k$, $P = D_{k-1}$ and $Q = D_k$. According to the definition of T_k, T_k is at least the time when m_k is received. By Proposition 2 for case k, m_k is sent at least after T_{k-1}. Therefore, $T_k > T_{k-1}$, or $B > A$. Thus Lemma 8 applies again. By Lemma 8, we thus have: (1) after T_k, $D_{k+1} = D_{k-1}$ must send at least one message m_{k+1} which precedes some message that arrives at D_k; we construct E_{imp} by letting m_{k+1} be sent; and (2) for any

$t \in [T_k, T_{k+1})$, if C_r starts ROT at some time in $[T_k, t)$ and $T_{D_k} = t$, then ROT may not return x or y. I.e., we prove Proposition 2 and Proposition 3 for case $k+1$. Therefore, we conclude that Proposition 2 and Proposition 3 are correct for any positive number k. Clearly, by the proof by induction above, we include an infinitely long sequence of messages $m_0, m_1, m_2, \ldots$ in our construction of E_{imp}.

Next we show that E_{imp} violates progress by contradiction. Suppose that E_{imp} does not violate progress. As there is no other write since the start of WOT, then in E_{imp} there is finite time τ such that any read of object X (or Y) which starts at any time $t \geq \tau$ returns x (or y). We have shown that $T_{k+1} > T_k$ for any positive k. Thus for any finite time τ, there exists K such that for any $k \geq K$, $T_k > \tau$. By Proposition 3, if C_r starts ROT at some time $[T_k, t)$ and $t_{D_k} = t$, then ROT may not return x or y. Since $T_k > \tau$, we reach a contradiction. Therefore we find an execution E_{imp} where two values of the same write-only transaction can never be visible, violating progress. □

3.4 The Impossibility of Fast Invisible Transactions

As we pointed out in the introduction, some systems considered a restricted model where all transactions are read-only and write operations are supported only outside the scope of a transaction. This restricted model also circumvents the impossibility result of Theorem 7. In this model, we present our second theoretical result, Theorem 8, stating that fast read-only transactions (while indeed possible) need to be visible (need to actually write).

We first formally define the notion of (in)visible fast transactions in Definition 16 and then present and prove Theorem 8.

3.4.1 Definitions

For simplicity of presentation as well as our proof, we define invisible transactions based on our definition of fast transactions.

Definition 16 (Invisible fast transactions)**.** We say that fast transaction T is invisible if for no client C, C's invocation I of T is both fast and visible.

Definition 16 is thus based on the visibility of I. Let some C's invocation I be fast. For any execution E that includes I, we schedule I according to Definition 11 and let M be the message exchange events between C and all servers to which C sends a message according to Definition 11. Then Definition 17 shows the visibility of I.

Definition 17. If for some E which schedules I as above, in addition to M, every execution E^- where C does not invoke I is still different from E, then we say I is visible.

Definition 17 defines the visibility of a transaction from the point of view of message exchange events. The intuition behind Definition 17 is that if no matter whether a client requests a transaction or not, in addition to the message exchange events required by the distributed protocol of the transaction, every message exchange event remains the same, then the transaction is indeed invisible to the storage system.

Here the definition of visible transactions covers two possibilities: (1) a server writes locally, which affects the messages sent later by the server; and (2) upon the transaction request, a server sends messages to other servers, for example, to notify them of the transaction. For the latter, even if a server sends empty messages, the transaction is considered visible (if these empty messages would not be sent without the occurrence of this transaction), as these messages add complexity to the storage system. From our proof of Theorem 8, however, we show that fast transactions send more than empty messages.

3.4.2 Result

Theorem 8. *A causally consistent transactional storage that supports fast read-only transactions does not provide invisible fast read-only transactions.*

The intuition of Theorem 8 goes back to that of Theorem 7. Consider a server P_X that stores object X and a server P_Y that stores object Y. Suppose that some client writes some new value to X and then to Y, while another client requests a read-only transaction that reads X and Y. There is a risk of violating causality for P_Y if P_X returns an old value to the read-only transaction. By the definition of fast transactions, P_Y must be able to avoid the risk without help from other servers and thus have to conservative, i.e., returning an old value as well. To ensure progress, P_X surely needs to notify P_Y of when P_Y can stop being conservative. However, due to asynchronous communication, P_X's notification can arrive earlier at P_Y than some transaction T where P_X has already returned an old value. This leads to the fact that P_X must send more than empty messages: P_X's notification needs to include some identifier of T in order for P_Y to satisfy causal consistency.

3.4.3 Proof by contradiction

Here we formalize our intuition and introduce the organization of the full proof.

We prove Theorem 8 by contradiction. I.e., suppose that some causally consistent transactional storage provides invisible fast read-only transactions. Then the assumption for contradiction is equivalent to say that for any C and C's invocation that is fast, for any E which schedules I by fast transactions, some execution E^- where C does not invoke I is the same as E except that E includes additional message exchange events between a client and severs of I. To prove Theorem 8, we choose executions where (1) for each object, some non-$\perp$ value is visible; (2) client C which invokes I has not done any transaction (including single-object write transactions and read-only transactions) before I.

As mentioned in the intuition of our proof, P_X's notification needs to include some identifier of a transaction to satisfy causal consistency. It is counter-intuitive that P_X only notifies the existence of one transaction rather than its identifier. We formalize the necessity of the identifier as follows.

Let $\mathscr{D}$ be some sets of clients which has not done any transaction before a read-only transaction. Let $\mathscr{D}_1$ be any subset of $\mathscr{D}$. Let S_1 be a set of invocations (1) which are fast, (2) each of which is issued by a different client in $\mathscr{D}_1$, and (3) which start at the same time t_0 and end at the same time T_2. We schedule each invocation of S_1 according to Definition 11. Let M_1 be the message exchange events between a client in $\mathscr{D}_1$ and all servers to which a client in $\mathscr{D}_1$ sends a message according to the first entry of Definition 11. We denote by $\mathscr{D}_2$ a different subset from $\mathscr{D}_1$, and S_2, M_2, the invocations and message exchange events that follow.

Proposition 4 (Assumption for contradiction)**.** *For any execution E_1 which schedules S_1 by fast transactions, for some $\mathscr{D}_2$, some execution E_2 where (1) $\mathscr{D}_1$ does not invoke S_1 but $\mathscr{D}_2$ invokes S_2 is the same as E_1 except for the message exchange events M_1 and M_2.*

Proposition 4 captures our intuition on the identifier in that if P_X's notification does not identify an invocation, then some other invocation can be an substitute and as a result, the message exchange events that follow are the same after the substitution.

Proposition 4 is a necessary condition for the assumption that no fast I is visible. To see this, we start with the assumption that no fast I is visible. Then given $\mathscr{D}_1$ and any E_1, we apply the assumption that no fast I is visible to clients in $\mathscr{D}_1$ one by one. After $|\mathscr{D}_1|$ times, all clients and their invocations are removed from E_1, the resulting execution is E_2 for an empty set of clients $\mathscr{D}_2$, which proves Proposition 4.

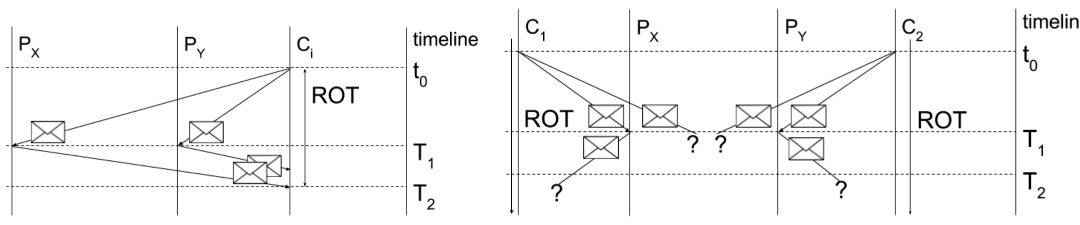
As a result, our proof of contradiction is organized as follows. First, we assume Proposition 4 for contradiction so that if Proposition 4 is violated, then the assumption that no fast I is visible is also violated. Then we present more details of the two executions E_1 and E_2 in the assumption for contradiction. Next, we construct another execution $E_{1,2}$ based on E_1 and E_2 which takes advantage of the same message exchange events in the assumption. Finally, we show that $E_{1,2}$ violates causal consistency. As Proposition 4 is a strictly weaker assumption than the assumption that no fast I is visible, by the contradictory execution $E_{1,2}$, we conclude that Theorem 8 is true.

3.4.4 Construction of executions

We consider a specific read-only transaction $ROT = (r(X)*, r(Y)*)$. Let S_1 be a set of invocations of ROT. Let E_1, $\mathscr{D}_1$ and M_1 follow the definitions in Proposition 4. Then in E_1, every invocation in S_1 starts at the same time t_0 and ends at the same time T_2. Both t_0 and T_2 are notations rather than take specific values. For S_1, w.l.o.g., we further schedule every message which a client in $\mathscr{D}_1$ sends (to a server) to arrive at the same time T_1. According to Definition 11, each client in $\mathscr{D}_1$ receives at most one message. If any, we say that message is a critical message. After T_1 and as long as P_X (P_Y) is still about to send a critical message to some client in $\mathscr{D}_1$, P_X (P_Y) receives no message from any other server. Each client in $\mathscr{D}_1$ receives at most one message from each of P_X and P_Y and returns ROT at time T_2.

By Proposition 4, for E_1, $\exists \mathscr{D}_2$, such that some E_2 where $\mathscr{D}_2$ invokes S_2 instead is the same as E_1 except for message exchange between $\mathscr{D}$ and $\{P_X, P_Y\}$. Then we can schedule every invocation in S_2 in a similar way as S_1. Since for each server $P \in \{P_X, P_Y\}$, P receives no message from any other server after T_1 (before P is still about to send a critical message to some client in $\mathscr{D}_1$) in E_1, we let every invocation in S_2 start at the same time t_0, and every message which a client in $\mathscr{D}_2$ sends (to a server) be received at the same time T_1 in E_2. After T_1, although in E_2, P_X (P_Y) can delay or advance the time when P_X (P_Y) replies to a client in $\mathscr{D}_2$, the time period when P_X (P_Y) receives no message from any other server is the same as in E_1 by Proposition 4. Therefore, w.l.o.g., we assume that the time when P_X (P_Y) sends the last critical message to a client in $\mathscr{D}$ is the same in E_1 and E_2. By the property of fast transactions, each client in $\mathscr{D}_2$ receives at most one message from each of P_X and P_Y and returns ROT, w.l.o.g., at the same time T_2.

The two executions E_1 and E_2 are illustrated in Figure 3.4a. Since after T_2, by Proposition 4, E_1 and E_2 are the same, then we construct both executions as follows. We let another client $C \notin \mathscr{D}$ perform two writes $w(X)x$ and $w(Y)y$ after T_2 to establish $w(X)x \rightsquigarrow w(Y)y$ according to Definition 6. As we assume an arbitrarily large number of clients, C exists. By the schedule of fast read-only transactions, i.e., Definition 11, in E_1 and E_2, some messages may be delayed but need not to be delayed indefinitely. (Moreover, if the delayed message is between two servers, then it is received at the same time in E_1 and E_2 by Proposition 4; if the delayed message is from a server to a client, which is not a critical message, then it is received after the client returns, i.e., time T_2 by Definition 11.) In both executions, no message is delayed indefinitely and therefore y is eventually visible. We denote by τ the time instant after which y is visible in both executions.



(a) Message schedule of E_i (b) Message schedule of $E_{1,2}$

Figure 3.4 – Construction and extension of E_i

As promised, we now construct execution $E_{1,2}$ based on E_1 and E_2. The goal is to let $E_{1,2} = E_1 = E_2$ except for the communication with $\mathscr{D}$ until τ. For $i \in \{1,2\}$, let C_i be any client in $\mathscr{D}_i$. W.l.o.g., we assume that $\mathscr{D}_1 \backslash \mathscr{D}_2 \neq \emptyset$. In $E_{1,2}$, every client in $\mathscr{D}_1 \cup \mathscr{D}_2$ invokes ROT at t_0. As illustrated in Figure 3.4b, while every client $C_1 \in \mathscr{D}_1$ invokes ROT, P_X receives the same message from C_1 at the same time T_1 and no message from a server after T_1 in a same way as in E_1, and sends the same message to C_1 at the same time as in E_1. Similarly, while every client $C_2 \in \mathcal{D}_2$ invokes ROT, P_Y receives the same message from C_2 at the same time T_1 and no message from a server after T_1 in a same way as in E_2, and sends the same message to C_2 at the same time as in E_2. The construction so far only completes the message schedule of invocations of $\mathcal{D}_1 \cap \mathcal{D}_2$.

Let us now consider clients in $\mathcal{D}_1 \backslash \mathcal{D}_2$ and $\mathcal{D}_2 \backslash \mathcal{D}_1$ respectively. While every client in $\mathcal{D}_1 \backslash \mathcal{D}_2$ invokes ROT, P_Y does not receive the message from the client. Similarly, while every client in $\mathcal{D}_2 \backslash \mathcal{D}_1$ invokes ROT, P_X does not receive the message from the client. Due to asynchronous communication, the reception of these messages may be delayed by a finite but unbounded amount of time. We explain later the exact amount. The construction so far is illustrated in Figure 3.4b. Based on the construction so far, by T_2, P_X is unable to distinguish between E_1 and $E_{1,2}$ while P_Y is unable to distinguish between E_2 and $E_{1,2}$.

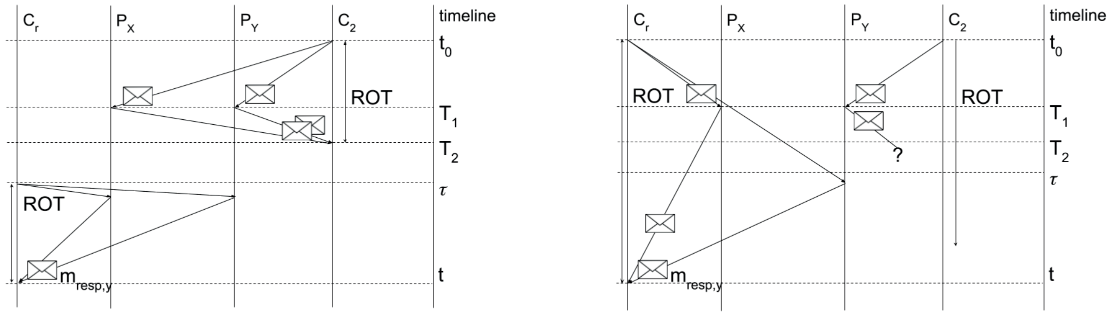
In our previous construction of E_1 and E_2, after T_2, E_1 and E_2 are the same. By the indistinguishability of P_X and P_Y here, we are allowed to continue the construction of $E_{1,2}$ so that after T_2, E_1, E_2 and $E_{1,2}$ are the same. In particular, in $E_{1,2}$, after T_2, the same client $C \notin \mathcal{D}$ performs two writes $w(X)x$ and $w(Y)y$ after T_2 to establish $w(X)x \rightsquigarrow w(Y)y$ in the same way as E_1 and E_2. To keep the respective indistinguishability of P_X and P_Y, these messages that are delayed in the construction so far are received after time τ. We explain later the exact time regarding reception of some of these delayed messages. We recall that τ takes a value determined by our previous construction of E_1 and E_2. Then as a result, we achieve our goal of construction that $E_{1,2} = E_1 = E_2$ except for the communication with $\mathcal{D}$ until τ.

3.4.5 Proof of Theorem 8

The main idea of our proof is as follows. We continue to construct the two executions E_2 and $E_{1,2}$ starting from time τ so that P_Y continues to be unable to distinguish between E_2 and $E_{1,2}$, and then replies to a client a value in $E_{1,2}$ that breaks causal consistency. As we reach a contradiction, we show that our assumption for contradiction, namely, Proposition 4 is violated. Thus we conclude that Theorem 8 is correct. After we prove $E_{1,2}$ is a contradictory execution below, we do not repeat this conclusion.

Our proof by contradiction surely relies on the indistinguishability of servers (P_X, P_Y) between executions $(E_1, E_2, E_{1,2})$. Hence to circumvent the impossibility result of Theorem 8, one has to break the indistinguishability for servers in the construction above, implying the necessity of some write to some server (i.e., writing to a client without the client forwarding the write to any server is not an option). This is consistent with our expectation of what Theorem 8 shows at the beginning of Section 3.4, i.e., fast read-only transactions need to actually write to the storage system.

Proposition 5 (Contradictory execution)**.** *Execution* $E_{1,2}$ *can violate causal consistency.*



(a) Extension of E_2

(b) Extension of $E_{1,2}$.

Figure 3.5 – Extension of two executions

Proof of Proposition 5. We first extend E_2 and $E_{1,2}$ after τ, as illustrated in Figure 3.5 and we present the details below. To start with, we let any client C_r in $\mathscr{D}_1 \backslash \mathscr{D}_2$ start ROT immediately after τ in E_2. Thus in E_2, every C_r sends a message to P_Y.

Recall that in our previous construction of $E_{1,2}$, every C_r sends a message to P_Y as well yet the reception is delayed. Here we construct E_2 and $E_{1,2}$ together for the communication between every C_r and P_Y as follows. For each C_r, we schedule the message which C_r sends to P_Y to arrive at some same time (which is after τ) in both E_2 and $E_{1,2}$. W.l.o.g., we schedule the time to be the same for all clients C_r in $\mathscr{D}_1 \backslash \mathscr{D}_2$. We also schedule P_Y to receive no message from any other server after receiving a message from each C_r in both E_2 and $E_{1,2}$. Then by fast read-only transactions, P_Y still eventually replies to each C_r in both E_2 and $E_{1,2}$.

For the completeness of the construction of E_2, we include the schedule of every C_r's communication with P_X below. In E_2, every C_r sends a message to P_X. For each C_r, we schedule the message which C_r sends to P_X to arrive at the same time as the message which C_r sends to P_Y. We also schedule P_X to receive no message from any other server after receiving a message from each C_r, and P_X to eventually reply to each C_r in E_2.

Now w.l.o.g., we assume that P_X, P_Y in E_2 and P_Y in $E_{1,2}$ send their reply (as defined in Definition 11) to each C_r at the same time. By fast read-only transactions, each C_r receives at most one message from each of P_X and P_Y before C_r returns. As illustrated in Figure 3.5, w.l.o.g., we assume these messages arrive at each C_r at some same time, C_r receives at most one message from each of P_X and P_Y, and every C_r returns to ROT at the same time t in both E_2 and $E_{1,2}$.

Now that we have constructed E_2 and $E_{1,2}$, we compute the return value of ROT in E_2 and $E_{1,2}$ below. Denote the message which C_r receives from P_Y at t by $m_{resp,Y}$. Denote by $m_{resp,X}$, the message which C_r receives from P_X at t. Therefore based on our extension of E_2 and $E_{1,2}$, since by the time when P_Y sends a message to each C_r, P_Y is unable to distinguish between E_2 and $E_{1,2}$, $m_{resp,Y}$ takes the same content in E_2 and $E_{1,2}$ (yet $m_{resp,X}$ can take different content in E_2 and $E_{1,2}$).

We focus on $m_{resp,Y}$. By progress, in E_2, C_r returns y for $r(Y)*$ in ROT. By one-version property, $m_{resp,Y}$ reveals exactly one version of Y, and $m_{resp,X}$ reveals no version of Y. Since $m_{resp,X}$ reveals no version of Y, $m_{resp,Y}$ cannot reveal a version of Y different from y. In other words, $m_{resp,Y}$ must reveal y. In $E_{1,2}$, $m_{resp,X}$ cannot reveal x as $w(X)x$ starts after T_2. Then $m_{resp,X}$ must reveal some value $x^* \neq x$ and $x^* \neq \perp$. As $m_{resp,Y}$ has already revealed y, messages $\{m_{resp,X}, m_{resp,Y}\}$ cannot reveal other versions of X or Y. In $E_{1,2}$, since every C_r does not issue any other transaction before ROT, the return value of ROT solely depends on $\{m_{resp,X}, m_{resp,Y}\}$, which is then (x^*, y).

Finally, we show that the return value (x^*, y) in $E_{1,2}$ violates causal consistency by contradiction. Suppose that $E_{1,2}$ satisfies causal consistency. Then by Definition 8, for any C_r, we can totally order all C_r's transactions and all write operations such that the last preceding writes of X and Y before C_r's ROT are $w(X)x^*$ and $w(Y)y$ respectively. Since $w(X)x \rightsquigarrow w(Y)y$, then $w(X)x$ must be ordered before $w(Y)y$. This leads $w(X)x^*$ to be ordered after $w(X)x$. We now extend $E_{1,2}$ so that (1) every previously delayed message is received after time t, and no other message is delayed; (2) x is thus visible; and (3) C_r invokes $ROT_1 = (r(X)*, r(Y)*)$ after x is visible. In $E_{1,2}$, ROT_1 returns (x, y) by Definition 10. According to Definition 8, the last preceding write of X before ROT_1 must be $w(X)x$. However, $w(X)x^*$ has already been ordered after $w(X)x$ and thus the last preceding write of X before ROT_1 is $w(X)x^*$. A contradiction. We thus conclude that $E_{1,2}$ indeed violates causal consistency. □

3.5 Alternative Protocols

To complement our theorems, we here present two alternative protocols which provides fast causal transactions. To show the feasibility of fast read-only transactions, we describe a protocol which makes fast read-only transactions visible by asynchronous propagation of transaction identifiers. (Recall that in the proof of Theorem 8, we mention the intuition that some kind of transaction identifier is necessary.) To discuss the impossibility results under different assumptions on the underlying system (asynchronous or not) and the global clock (accessible or not), we present a timestamp-based implementation of causally consistent transactional storage. As we consider an accessible global clock, we remove the assumption of oblivious algorithms previously in the timestamp-based implementation to take advantage of the clock.

3.5.1 Visible fast read-only transactions

We present below a suite of algorithms, $\mathscr{A}$, for fast read-only transactions. To comply with our Theorem 7, we restrict all transactions to be read-only and updates to be outside transactions (or equivalently be considered as single-object write transactions). The goal of $\mathscr{A}$ is to better understand our Theorem 8. Theorem 8 shows that fast read-only transactions are visible. The intuition of Theorem 8 is that after a fast read-only transaction T, servers may need to communicate the information of T among themselves. However, it is not clear when such communication occurs. The COPS-SNOW [44] algorithm shows that the communication can take place during one client request of write. $\mathscr{A}$ below shows that the communication can actually take place outside any client request of write and asynchronously. Different from COPS-SNOW where a value written is visible immediately after the write, $\mathscr{A}$ guarantees only eventual visibility.

Algorithm 8 Client-side read/write algorithms

```
local variables
    $lc$, logical clock
    $ctx$, context
end local variables
function WRITE($obj$, $val$)
    Identify server $S$ by $obj$
    $ctx_S$, $lc_S \leftarrow S$.write($lc$, $ctx$, $obj$, $val$)
    update_lc($lc_S$)
    update_context($obj$, $lc_S$, $ctx_S$)
    return OK
end function
function READ($objs$)
    $txID \leftarrow$ generate_txID()
    $fixedCtx \leftarrow ctx$
    for $obj$ in $objs$ do
        $val$, $ver$, $ctx_S$, $lc_S \leftarrow S$.read($lc$, $fixedCtx$, $obj$, $txID$)
        save $val$ to $vals$
        update_lc($lc_S$)
        update_context($obj$, $ver$, $ctx_S$)
    end for
    return $vals$
end function
```

Protocol

We describe first the data structure which each process maintains. All processes maintain locally their logical timestamps and update their timestamps whenever they find their local ones lag behind. They also move their logical timestamps forward when some communication with other processes is made. (The function call in $\mathscr{A}$ is update_lc of which the details are omitted for the simplicity of presentation.) Every client additionally maintains the causal dependencies of the current transaction (i.e., the transactions each of which causally precedes the current one). The maintenance of causal dependencies can be done in a similar way as in COPS [35] and COPS-SNOW [44]. (Our algorithm $\mathscr{A}$ maintains causal dependencies in variable ctx by function calls of update_context and ctx.update. The details are the same as COPS [35] and COPS-SNOW [44] and thus omitted.) Every client is able to generate transaction identifiers (by a function call of generate_txID in $\mathscr{A}$). Every server needs to store the causal dependencies which a client passes as an argument during its write. Every server additionally

Algorithm 9 Server-side read/write algorithms

```
local variables
    $lc$, logical clock
    $vis$, visible versions in tuples <$obj$, $ver$>
    $oldTx$ and $currTx$, storage of tuples <$obj$, $ver$, $ctx$, $txID$> for each object
end local variables
function WRITE($lc_C$, $ctx_C$, $obj$, $val$)
    update_lc($lc_C$)
    $ctx \leftarrow$ the context of $obj$ with the highest version in the storage
    $ctx$.update($ctx_C$)
    update_storage($obj$, $val$, $lc$, $ctx$)
    return $ctx$, $lc$
end function
function READ($lc_C$, $ctx_C$, $obj$, $txID$)
    update_lc($lc_C$)
    if $txID \in oldTx$ then
        $ver \leftarrow$ the version identified by $txID$ in $oldTx$
    else
        $v_{vis} \leftarrow$ the highest version of $obj$ in $vis$
        if <$obj$, $v$> is in $ctx_C$ and $v > v_{vis}$ then
            $ver \leftarrow v$
        else
            $ver \leftarrow v_{vis}$
        end if
    end if
    save <$obj$, $ver$, $ctx_C$, $txID$> to $currTx$
    $val \leftarrow$ the value identified by $ver$ of $obj$ in the storage
    $ctx \leftarrow$ the context identified by $ver$ of $obj$ in the storage
    return $val$, $ver$, $ctx$, $lc$
end function
```

maintains a data structure called $oldTx$ for each object stored.

We next sketch how writes and read-only transactions are handled. The full algorithms are shown in Algorithm 8 and Algorithm 9.

- Every client sends its logical timestamp as well as causal dependencies when requesting a write of object obj. A server uses the server's updated logical timestamp as the version ver of the value val written, stores the version and the value along with the causal dependencies ctx (by a function call update_storage(obj, val, ver, ctx) in Algorithm 9), and returns the version number to the client.

- Every client C sends its logical timestamp when requesting a read-only transaction tx. A server first searches tx in $oldTx$, and returns a pre-computed value according to entry tx in $oldTx$ if $tx \in oldTx$. Otherwise, a server returns some value previously observed

Algorithm 10 Server-side asynchronous check

```
local variables
    Same as in Algorithm 9
end local variables
when all versions of $obj$ below $ver$ are in $vis$, invoke async_check
procedure ASYNC_CHECK($obj, ver$)
    identify $ctx$ by $obj, ver$ in the storage
    for $obj_d, ver_d$ in $ctx$ do
        identify server $D$ by $obj_d$
        $oldTx_D, lc_D \leftarrow D$.async_checkVis($obj_d, ver_d, lc$)
        update_lc($lc_D$)
        save $oldTx_D$ to $oldTx$ as follows:
        for $txID$ in $oldTx_D$ do
            if $txID \notin oldTx$ then
                get tuple <$obj_d, *, ctx_d, txID$> from $oldTx_D$
                identify version $v_{prev}$ as the highest version below $ver$ of $obj$ in the storage
                if <$obj, v$> is in $ctx_d$ and $v > v_{prev}$ then
                    save tuple <$obj, v, -, txID$> into $oldTx$
                else
                    save tuple <$obj, v_{prev}, -, txID$> into $oldTx$
                end if
            end if
        end for
    end for
    for $txID$ in $currTx$ do
        if <$obj, v, *, txID$> is in $currTx$ and $v < ver$ then
            move the tuple identified by $txID$ from $currTx$ to $oldTx$
        end if
    end for
    save $< obj, ver >$ into $vis$
end procedure
function ASYNC_CHECKVIS($obj_d, ver_d, lc_S$)
    update_lc($lc_S$)
    when $< obj_d, ver_d >$ is in $vis$, return $oldTx, lc$
end function
```

by C or some value marked as “visible”.

We here sketch how $oldTx$ is maintained and communicated (during asynchronous propagation). The full algorithm is shown in Algorithm 10.

- After a server S responds to a client’s write request of value w for some object o, S sends a request to every server which stores some value v such that $w(o)v \rightsquigarrow w(o)w$. Any server responds such request with its local $oldTx$ when v is marked as “visible”.

- After S receives a response from all servers which store some value that causally precedes w, S stores their $oldTx$s into S's local one, chooses a value w^* which is written before w[9]

Any read-only transaction is stored and marked as "current" during its execution at any server. A "current" transaction T is put in $oldTx$ when some value w is "visible" and T has returned a value written before w of the same object.

Proof of correctness

Our suite of algorithms $\mathscr{A}$ above provides fast read-only transactions. As every message eventually arrives at its destination (and therefore asynchronous propagation eventually ends), $\mathscr{A}$ satisfies progress. As asynchronous propagation carries transaction identifiers, $\mathscr{A}$ is visible. In what follows, we show that $\mathscr{A}$ satisfies causal consistency.

In Algorithm 9, when a server stores a value, the server chooses a version number strictly greater than all values of the same object previously written. Therefore in addition to relation $\rightsquigarrow$, we also enforce an ordering on all writes of the same object by their version numbers. In what follows, we say that two writes $w_1 \rightarrow w_2$, if (1) w_1 is of a lower version number than w_2 and w_1, w_2 write the same object; or (2) $w_1 \rightsquigarrow w_2$; or (3) $\exists$ some write w_3 such that $w_1 \rightarrow w_3$ and $w_3 \rightarrow w_2$. We first show a property for any read-only transaction in Lemma 9. We then prove the correctness of $\mathscr{A}$ based on Lemma 9.

Lemma 9 (A correct snapshot for visible fast read-only transactions)**.** *Let T be any transaction that contains at least two reads. Given any two reads $r(a)u, r(o)v^* \in R_T$, if $\exists w(a)u^*$ such that $w(a)u$ is of a lower version number than $w(a)u^*$, then $w(a)u^* \rightarrow w(o)v^*$ does not hold.*

Proof of Lemma 9. By contradiction. Suppose that $r(a)u, r(o)v^* \in R_T$ and $w(a)u^* \rightarrow w(o)v^*$ holds. According to Algorithm 9, there are three possibilities when the server P_o that stores object o returns $val = v^*$ at $txID = T$:

1. $txID \in oldTx$;
2. $txID \notin oldTx$ but for object o, ctx_C specifies a version v, higher than the highest version v_{vis} in vis of the same object;
3. $txID \notin oldTx$; and for object o, ctx_C does not specify a version or any specified version v is lower than v_{vis}.

Let us examine each possibility. First, we look at the second possibility. Then $< o, v > \in ctx_C$, v corresponds to $val = v^*$ at P_o, and $v > v_{vis}$. The maintenance of variable ctx maintains

---

[9] In order to choose a value correctly, in the algorithm, S actually sends a request after all values written before w (of the same object) are marked as "visible". Also, S does not choose a value for some tx which S has chosen before.

the precedences of a transaction (a single-object write transaction or a read-only transaction) according to relation →. We sometimes also say a write is in ctx if the pair of the corresponding object and version number is in ctx. By the maintenance of ctx_C, since $w(a)u^* \rightarrow w(o)v^*$, then $w(a)u^* \in ctx_C$. However, according to Line 16 of Algorithm 10 and Line 19 of Algorithm 9, if $w(a)u^* \in ctx_C$, then the server P_a which stores object a is unable to return $val = u$ of which the version is lower than that of u^* at $txID = T$.

Next, we look at the third possibility. Then $txID \notin oldTx$. In addition, for object o, ctx_C does not specify a version or any specified version v is lower than v_{vis}; in either case, v_{vis} corresponds to $val = v^*$ at P_o. According to Line 4 and Line 33 of Algorithm 10, when T reads o at P_o, u and u^* are visible (i.e., in vis) at P_a. Clearly, if T reads a at P_a before P_a replies to $async\_checkVis(a, u^*, *)$, then P_a sends T to P_o during $async\_checkVis(a, u^*, *)$ and P_o could have $T \in oldTx$ when T reads o at P_o, which gives a contradiction. Therefore, T must read a after P_a replies to $async\_checkVis(a, u^*, *)$, i.e., after u^* is visible. Thus according to Line 19 of Algorithm 9, P_a must find $T \in oldTx$ when T reads a. Similarly, due to P_a's reply to P_o's call of $async\_checkVis(a, u^*, *)$, the first time when P_a receives T must be also after u^* is visible (while P_a invokes $async\_check(a, u_1)$ for some version u_1 after the version of u^*). Then according to Line 16 of Algorithm 10, P_a pre-determines a version no smaller than the version of u^* for T, which contradicts the return value $val = u$ of P_a.

Finally, we look at the first possibility. $txID \in oldTx$. Since P_o pre-determines $val = v^*$ for T, then either ctx_C specifies v^* for object o or v^* is visible the first time when P_o receives T. The two cases are similar to the second and third possibilities, leading to contradictions against the return value of P_a. As a result, we conclude that if $w(a)u^* \rightarrow w(o)v^*$ holds, then T cannot have both $r(a)u$ and $r(o)v^*$, which is equivalent to Lemma 9. □

Proof of causal consistency. By contradiction. Suppose that some execution E violates causal consistency. Then in E, some client C's local history cannot be totally ordered to satisfy Definition 7. Clearly, without any read-only transaction, we can order all writes in a way that respects relation → defined previously (which includes the relation of causality ⇝ between any two writes). Therefore C does at least one read-only transaction. In order to incorporate C's read-only transactions, we extend the relation → defined previously. Consider the set TX of transactions that consist of all writes in E and all C's read-only transaction. For any two transactions tx_1 and tx_2, we say that $tx_1 \rightarrow tx_2$, if (1) tx_1 and tx_2 are two writes, tx_1 is of a lower version number than tx_2 and tx_1, tx_2 write the same object; or (2) $tx_1 \rightsquigarrow tx_2$; or (3) $\exists$ some $tx_3 \in TX$ such that $tx_1 \rightarrow tx_3$ and $tx_3 \rightarrow tx_2$.

Let to_w be any ordering that respects relation →. We then add C's read-only transactions in to_w one by one. Since we suppose that E violates causal consistency, we let T be the first read-only transaction such that some to_w exists which can include C's read-only transactions before T but for any to_w, C's read-only transactions up to and including T cannot be placed in to_w to satisfy Definition 7.

Let A be the set of such ordering to_w that can include C's read-only transactions before T and let to_1 be any ordering in A. We first show that T must read at least two objects, the proof of which is by contradiction. Suppose otherwise that $R_T = \{r(a)u\}$. Let the last transaction (which can be a read-only transaction or a write) done by C before T is α. Let the first write done by C after T is β. Then in any to_1 where all C's read-only transactions before T are included, either (1) $w(a)u$ is before α, or (2) β is before $w(a)u$, or (3) $w(a)u$ is between α and β. In the third case, we put T immediately after $w(a)u$. In the second case, $\beta \rightarrow w(a)u$ does not hold. (Suppose otherwise that $\beta \rightarrow w(a)u$ holds. Then the logical timestamp l_1 which the client of $w(a)u$ receives from P_a during $w(a)u$ is higher than the logical timestamp l_2 which C receives from the server that stores the object written by β during β. However, when T reads a, the logical timestamp which C receives from P_a is at least l_1, and as a result, the value of $l_2 \geq l_1$, a contradiction.) We move β and its successors of relation $\rightarrow$ after $w(a)u$. The resulting ordering is still in A. We then put T immediately after $w(a)u$. In the first case, there are two possibilities: (i) between $w(a)u$ and α, there is some write $w(a)u^*$; (ii) between $w(a)u$ and α, there is no write $w(a)u^*$. For the latter, we put T immediately after α. For the former, let $w(a)u^*$ be the first write of object a after $w(a)u$ in to_1. Then $w(a)u^* \rightarrow \alpha$ does not hold. (Suppose otherwise that $w(a)u^* \rightarrow \alpha$ holds. Then $w(a)u^*$ is in the variable ctx maintained by C before T starts. As a result, when T reads a, P_a sees $w(a)u^* \in ctx_C$ and thus returns a value with a version number no smaller than that u^*, a contradiction.) We move $w(a)u^*$ and its successors of relation $\rightarrow$ after α. The resulting ordering is still in A. We then put T immediately after α.

Now we continue in the case where T reads at least two different objects. We consider Lemma 9 as a property of any read-transaction. Based on Lemma 9 and to_1, we construct another ordering $to_2 \in A$ as follows. For any $r(a)u \in R_T$, consider W_u be the set of such write $w(o)v^*$ that (1) in to_1, some write $w(a)u^*$ is after $w(a)u$ and $w(o)v^*$ is after $w(a)u^*$ and (2) $r(o)v^* \in R_T$. If $W_u = \emptyset$, then we do nothing for $r(a)u$; otherwise, we let $w(a)u^*$ be the first write of a after $w(a)u$ in to_1. We then augment W_u by adding the precedence of each element according to relation $\rightarrow$, and we do this until no more write after $w(a)u^*$ in to_1 can be added. Let ss be the subsequence of to_1 which contains all writes in W_u. We move ss immediately before $w(a)u^*$.

Below we verify that the resulting ordering to_u (after the construction for $r(a)u$) falls in A. By the construction based on relation $\rightarrow$, to_u still respects relation $\rightarrow$. Thus we only need to verify that C's read-only transactions before T can be placed in to_u. We know that in to_1, all C's read-only transactions before T can be placed. Then while moving ss, we may move some of C's read-only transactions as well. Namely, for any to_1, given a way to put all C's read-only transactions before T so that they are legal, we include in W_u the last read-only transaction rtx_{last} done by C before T that is put after $w(a)u^*$; then we still augment W_u by adding the precedence of each element according to relation $\rightarrow$ and stop the addition when no more write or read-only transaction after $w(a)u^*$ in to_1 can be added. Now consider ss as the subsequence of to_1 which contains all writes and read-only transactions in W_u. Since $w(a)u^* rightarrowrtx_{last}$ does not hold, we still move ss immediately before $w(a)u^*$ and the resulting to_u respects relation →. Thus if ss includes any read-only transaction, then in to_u, the position of the read-only transaction is still legal. In addition, C's read-only transactions that are put before $w(a)u^*$ remain unchanged. Therefore, to_u finds a way to place all C's read-only transactions before T and falls in A.

Since ss is only a subsequence of to_1, the move of ss creates no new pair $w(a)u$ and $w(o)v^*$ such that $r(a)u, r(o)v^* \in R_T$ and $w(o)v^*$ is after $w(a)u^*$ and $w(a)u^*$ is after $w(a)u$ for some $w(a)u^*$. Then after a finite number of moves, we can construct an ordering $to_2 \in A$ such that for any $r(a)u \in R_T$, $W_u = \emptyset$. We now turn to the placement of T in to_2. Let α be C's last transaction before T. Let β be C's first write after T. Let w_{last} be the last write in to_2 that corresponds to some read in T. Since during the construction of to_2, we move the positions of some read-only transactions as well, after the construction of to_2, we have also constructed a way to place all C's read-only transactions before T in to_2. For this placement, there are three possibilities: (1) w_{last} is between α and β, (2) w_{last} is before α, and (3) w_{last} is after β. We show that in all these possibilities, we can place T possibly after some rearrangement so that all C's transactions up to and including T are legal, which gives a contradiction. In the first possibility, we place T after w_{last} and we find all preceding writes of T correct, a contradiction. In the second possibility, if there is any $w(a)u^*$ between $w(a)u$ and α, then according to Line 19 and Line 16 of Algorithm 10, no $w(a)u^*$ exists such that (1) $w(a)u^* \rightarrow \alpha$ and (2) $r(a)u \in R_T$; therefore we can move $w(a)u^*$ and its successors of relation → after α in to_2; after the possible rearrangement, we place T after α and find all preceding writes of T correct, a contradiction. In the third possibility, for any $r(a)u \in R_T$, $\beta \rightarrow w(a)u$ does not hold. We thus move β and its successors of relation → after w_{last} in to_2; after the rearrangement, we place T after w_{last} and find all preceding writes of T correct, a contradiction. As T is able to be placed in some ordering in A, we reach a contradiction against our assumption, and we must therefore conclude that our algorithm $\mathscr{A}$ satisfies causal consistency. □

3.5.2 Timestamp-based implementation

We present here some timestamp-based implementation of causally consistent transactional storage to show that our impossibility results (Theorem 7 and Theorem 8) can be circumvented under different assumptions on the underlying system. As we show later, the timestamp-based implementation is invisible and the complexity of a server processing a client request is low, w.l.o.g., we assume that the local computation at any server takes negligible time (compared with communication delay) in our timestamp-based implementation.

Invisible fast read-only transactions

The algorithm $\mathscr{B}$ here relies on the assumption that all processes can access a global accurate clock. The algorithm considered here is non-oblivious and thus takes advantage of accurate timestamps. The description of $\mathscr{B}$ is as follows.

- Before any client starts a transaction, the client accesses the clock and stamps the transaction with the current time;
- Every client sends the accurate timestamp while requesting a transaction;
- If an operation writes a value to an object, then the server that stores the object attaches the timestamp to the value;
- If an operation reads a value from an object, then the server that stores the object returns the value with the highest timestamp which is still smaller than the timestamp of the transaction. (If two or more values are attached with the same highest timestamp, then we break the tie by returning the value written with the highest client ID.)

Each transaction induces one communication round and is thus fast. Each read-only transaction is also invisible. Algorithm $\mathscr{B}$ guarantees progress as the global accurate clock makes progress.

Given the accurate global clock, $\mathscr{B}$ is correct when a transaction is not allowed to write more than one object. Below is its proof of correctness (which is actually similar to the proof of correctness of $\mathscr{A}$).

First, we construct an acyclic graph of all writes according to causality. If two writes are on the same object, then we add a directed edge from the write with the lower timestamp to the higher one. If two writes can happen at the same timestamp, then we augment timestamps by breaking the tie using client IDs. After the addition, the graph is still acyclic. We consider all possible topological sorts of the graph. We also define the relation $\to$ between any two writes w_1, w_2 as follows. If $w_1 \to w_2$, then either $w_1 \rightsquigarrow w_2$, or w_1 and w_2 are on the same object while w_1 is of a lower timestamp, or there exists w_3 such that $w_1 \to w_3$ and $w_3 \to w_2$. Clearly, relation $\to$ captures the order between two writes in topological sorts.

Second, for each client C, if we add C's read-only transactions one by one, then either we succeed in one topological sort, or we find the first transaction T such that all topological sorts are incorrect. To examine C's read-only transactions, we augment relation $\to$ by adding (tx_1, tx_2) if at least one transaction between tx_1 and tx_2 is done by C and $tx_1 \rightsquigarrow tx_2$, and by transitivity. The relation is still acyclic. Suppose that for some client C, all topological sorts are incorrect. Let T be the first transaction such that all topological sorts are incorrect. There are two possibilities: (1) T reads a single object; (2) T reads multiple objects. In the first possibility, let $R_T = \{r(a)u\}$. In any topological sort where C's read-only transactions before T are put legally, either some of C's transaction (a single-object write transaction or a read-only transaction) done before T is put after $w(a)u^*$, or some of C's write done after T is put before $w(a)u$. For the former, for any transaction tx done by C before T that is ordered after $w(a)u^*$ in a topological sort, $w(a)u^* \to tx$ does not hold. (Suppose otherwise that $w(a)u^* \to tx$ holds. If tx writes a, by the definition of $\to$, tx is of a higher timestamp than a. If tx reads a, still by the definition of $\to$, there exists some write $w(a)u^{*}*$ of a timestamp no smaller than $w(a)u^*$

such that tx returns u^{**}. If tx neither writes nor reads a, again by the definition of →, either $w(a)u^*$ ends before tx starts (by program-order causality) or u^* is readable at the server before tx starts (by read-from causality). In any of the cases above, T which follows tx should return a value of a with a higher timestamp than that of u. A contradiction.) As a result, we can move tx as well as its precedence according to relation → before the first $w(a)u^*$ after $w(a)u$ in a given topological sort, and reach a contradiction: the resulting sort respects → and T can be put immediately before $w(a)u^*$ to be legal. For the latter, for any write w done by C after T that is ordered before $w(a)u$ in a given topological sort, $w \rightarrow w(a)u$ does not hold. (Suppose otherwise that $w \rightarrow w(a)u$ holds. Then the timestamp of w is lower than that of $w(a)u$. Therefore the timestamp of T is also lower than that of $w(a)u$, which leads T to be unable to return u. A contradiction.) As a result, we can move w as well as its successors according to relation → after $w(a)u$, and reach a contradiction: the resulting sort respects → and T can be put immediately after $w(a)u$ to be legal.

Thus we exclude the first possibility. In the second possibility, we first prove that every transaction T satisfies Lemma 10. Then given a topological sort, given any read $r(a)u \in R_T$, we collect set W_u of such write $w(o)v^*$ that (1) $w(a)u$ is of a smaller timestamp than $w(a)u^*$ and $w(a)u^*$ is ordered before $w(o)v^*$ in the sort, and (2) $r(o)v^* \in R_T$. We also collect in W_u the read-only transactions of C done before T which are put after any $w(a)u^*$ of a higher timestamp than $w(a)u$ in the sort. Any such read-only transaction tx satisfies $w(a)u^* \rightarrow tx$ does not hold for any $w(a)u^*$. We move W_u as well as its precedence according to relation → before the first $w(a)u^*$ after $w(a)u$ in the sort. In the resulting sort, all C's read-only transactions before T are put legally.

Lemma 10 (A correct snapshot for the timestamp-based implementation)**.** *Let T be any transaction that contains at least two reads. Given any two reads $r(a)u, r(o)v^* \in R_T$, if $\exists w(a)u^*$ such that $w(a)u$ is of a smaller timestamp than $w(a)u^*$, then $w(a)u^* \rightarrow w(o)v^*$ does not hold.*

Proof of Lemma 10. By contradiction. Suppose that $w(a)u^* \rightarrow w(o)v^*$ holds. Then before $w(o)v^*$ starts, a write on object a with a timestamp no lower than the timestamp of $w(a)u^*$ has ended. Therefore when T starts, since the global accurate clock is accessible to every process, the server which stores object a has a value u^{**} with a timestamp no lower than the timestamp of $w(a)u$ yet lower than the timestamp of T. This leads T to return u^{**} rather than u, a contradiction. □

Given a topological sort, we repeat the procedure above from the first write which corresponds to a read in R_T to the last one. We then obtain a topological sort which respects →, where every read-only transaction done by C before T is legal and no write $w(a)u^*$ of a higher timestamp than $w(a)u$ is before $w(o)v^*$ for any $r(a)u, r(o)v^* \in R_T$. Let α be C's last transaction before T. Let β be C's first write after T. There are three possibilities in the resulting sort for the position of the last write w_{last} which corresponds to a read in R_T: (1) w_{last} is before α; (2) w_{last} is between α and β; and (3) w_{last} is after β. In the first case, if there is any write w_e of the same object as w_{last} between w_{last} and α, then we further move the first w_e (which is between w_{last} and α) as well as its successors according to relation $\rightarrow$ after α; then we put T immediately after α, a contradiction. In the second case, we simply put T after w_{last}, a contradiction. In the third case, by the use of accurate global clock, $\beta \rightarrow w_{last}$ does not hold; we can move β as well as its successors according to relation $\rightarrow$ after w_{last}; then we put T immediately after w_{last}, a contradiction. Therefore, $\mathscr{B}$ is correct given the access to a global accurate clock.

As a result, $\mathscr{B}$ shows that Theorem 8 can be circumvented given the access to a global clock and the use of non-oblivious algorithms. In addition, as shown by $\mathscr{B}$, the access to the global clock guaranteed for clients is sufficient for the circumvention of Theorem 8. On the other hand, it is also necessary: the proof of Theorem 8 holds if only servers can access the global clock (while a client request is still oblivious to its local clock). Although the access to a global accurate clock circumvents the impossibility result of Theorem 8, the proof of Theorem 8 still holds even if the global accurate clock is accessible to all processes.

Invisible fast read-write transactions

We next show that given the access to a global accurate clock and an upper bound u on the communication delay, we can adapt our algorithm $\mathscr{B}$ above to work for read-write transactions. In other words, given the access to a global accurate clock and an upper bound u on the communication delay, Theorem 7 can be circumvented. Let us call the modified algorithm by $\mathscr{B}^+$. Now given the upper bound u, a client imposes that every transaction is executed for a time period of $2u$; when returning a value to some read of an object o, instead of comparing with the timestamp ts of the transaction in question, the server compares the timestamp of each value of o with $ts - 2u$.

Now to prove the correctness of $\mathscr{B}^+$, we define relation $\rightarrow$ between any two transactions tx_1, tx_2 which contain a non-empty set of writes as follows. If $tx_1 \rightarrow tx_2$, then either $tx_1 \rightsquigarrow tx_2$, or tx_1 and tx_2 have their write set overlap on the same object while the timestamp ts_1 of tx_1 and ts_2 of tx_2 satisfy $ts_1 < ts_2 - 2u$, or there exists tx_3 such that $tx_1 \rightarrow tx_3$ and $tx_3 \rightarrow tx_2$. The proof starts with all topological sorts of a graph that represents the relation $\rightarrow$ defined here and continues with the examination of each transaction that includes a non-empty set of reads done by a client. The proof of $\mathscr{B}^+$ is similar to that of $\mathscr{B}$ and therefore omitted.

3.6 Storage Assumptions

For presentation simplicity, we made an assumption that servers store disjoint sets of objects. In this section, we show how our results apply to the non-disjoint case. A general model of servers' storing objects can be defined as follows. Each server still stores a set of objects, but no server stores all objects. For any server S, there exists object o such that S does not store o. In this general model, when a client reads or writes some object o, the client can possibly request multiple servers all of which store o. Below we first adapt progress property to the general model in a way that is decoupled from the underlying distributed protocols of the storage system. Then w.l.o.g., we may assume that when client C accesses o, C requests all servers that store o.

3.6.1 Weak progress property

As promised previously, we adapt our previous definition of progress property (Definition 9 and Definition 10) to the general model of servers' storing objects. As shown in Definition 18 and Definition 19, weak progress only guarantees that if one write causally precedes the other, then the former one is eventually overwritten. Such weakness in the definition is to avoid any specific assumption on the underlying distributed protocol of the storage system in the general model. In the general model, a distributed protocol may choose an arbitrary rule in deciding which value is visible depending on the application (different from the application of the natural rule that the last writer wins under the previous assumption), especially for causally related writes. Hence it is necessary to define weak progress to cover all such rules. As a result of weaker progress guarantee in the general model, as we show later, the constructions of executions for the proofs of two impossibility results are more specific than those for our previous proofs.

Definition 18 (Weak eventual visibility)**.** If we say a write w in transaction T is weakly eventually visible, then there exists some finite time $\tau_{x,v}$ such that for any transaction T_{rx} which starts no earlier than $\tau_{x,v}$ and has $r(x)v_{new} \in R_{T_{rx}}$, $v_{new} \neq \perp$ and any transaction T_{wx} such that $w(x)v_{new} \in W_{T_{wx}}$ does not satisfy $T_{wx} \rightsquigarrow T$. (Here T_{wx} can be T, and if T is the only transaction so far, then $T_{wx} = T$.)

Definition 19 (Weak progress)**.** A causally consistent storage guarantees weak progress if every write is weakly eventually visible.

3.6.2 Impossibility of fast transactions

We sketch here the correctness of Theorem 7 in the general model. In the general model, the proof of Theorem 7 still constructs a contradictory execution E_{imp}. The construction goes by induction as shown in Proposition 6. To satisfy causality, by induction, there is a sequence of an infinite number of messages in the construction E_{imp}, which thus violates progress and shows the correctness of this impossibility result. We sketch below only the construction of E_{imp}. The proof of the violation of progress is the same as the previous proof of Theorem 7 and is then omitted.

Different from the previous construction, the construction of E_{imp} starts with no transaction. Client C_w writes a first transaction that writes to all objects and reads no object. Suppose that at time t_{start}, the writes of the first transaction are all visible. After t_{start}, C_w does a second transaction WOT that writes to all objects (to have the transaction span multiple servers) and reads no object at time t_w. All other clients do no transaction. Similarly with the previous construction, the construction of E_{imp} goes as long as at least one write of WOT is not visible.

Proposition 6 (Induction under the general storage assumption)**.** *After* t_w*, at least one server must send one message. Let* M_0 *be the set of messages which a server sends after* t_w*. For the first server which receives a message in* M_0*, denote the message by* m_0*. Thus we construct* E_{imp} *to send* m_0 *after* t_w*.*

For any positive number k*, assume that in* E_{imp}*,* $m_0, m_1, \ldots, m_{k-1}$ *have been sent. Then after the reception of* m_{k-1}*, at least one server must send one message. Let* M_k *be the set of messages which a server sends after the reception of* m_{k-1}*. For the first server which receives a message in* M_k*, denote the message by* m_k*. Thus we construct* E_{imp} *to send* m_k *after the reception of* m_{k-1}*.*

Proof of Proposition 6. Proposition 6 clearly consists of the base case and the inductive step. The proof of the base case is by contradiction. Suppose that after t_w, no server sends any message and eventually all writes of WOT are visible. Then we can construct an execution E^+ based on E_{imp}. In E^+, we let client C_r do a read-only transaction ROT that reads all objects, send a message to each server, and receive at most one message from each server. We schedule the message exchange between C_r and all servers according to Definition 11. By weak eventual visibility, in some execution E^+, ROT returns all values written by WOT. We call this execution by E. In E, for each server S, let $m_{resp,S}$ be the message which C_r receives from S. Let ss be the set of such server S that $m_{resp,S}$ reveals some version written by WOT. Let R be a server in ss. We then construct an execution E_{new} based on E. In E_{new}, we let client C_r do a read-only transaction ROT that reads all objects while we change the schedule of message exchange between C_r and all servers. More specifically, we let R receive C_r's message at the same time as in E but different from E, we let all servers except for R receive C_r's message at some same time before t_w, and the rest of the schedule follows Definition 11. Therefore, R still replies to C_r the same message as in$E$, which reveals some version written by WOT; however, all servers except for R reply with messages that reveal some version written by the first transaction of C_w. As a result, the return value of ROT in E_{new} breaks causal consistency, which leads to a contradiction. We then conclude the correctness of the base case; i.e., after t_w, at least one server sends some message before eventually all writes of WOT are visible, and we construct E_{imp} to send m_0 after t_w.

The proof of the inductive step is similar and also by contradiction. Suppose that after the reception of m_{k-1}, no server sends any message and eventually all writes of WOT are visible. We construct an execution E^+ based on E_{imp}, of which the construction is similar to that in the base case. By weak eventual visibility, in some execution E^+, ROT returns all values written by WOT. We still call this execution by E, define $m_{resp,S}$ for each server S, and use the notation ss. Let P be the sender of m_{k-1}. If $P \in ss$, then we let $R = P$; otherwise, R is a server in ss. We then construct an execution E_{new} based on E. In E_{new}, we let client C_r do a read-only transaction ROT that reads all objects while we change the schedule of message exchange between C_r and all servers. More specifically, we let $\{R, P\}$ receive C_r's message at the same time as in E but different from E, we let all servers except for $\{R, P\}$ receive C_r's message at some same time between the reception of m_{k-2} (if there is one) and the reception of m_{k-1}, and the rest of the schedule follows Definition 11. In addition, if m_{k-1} is not sent to R, then we delay m_{k-1} from being received in E_{new}; otherwise, we allow m_{k-1} to be received at the same time as in E. Then $\{R,P\}$ still replies to C_r the same message as in$E$, which reveals some version written by WOT. However, according to the correctness of case $k-1$ (i.e., the assumption for the correctness of case k), all servers except for $\{R,P\}$ are unable to distinguish between whether message m_{k-1} is sent or not. As a result, all servers except for $\{R,P\}$ reply with messages that reveal some version written by the first transaction of C_w. The return value of ROT in E_{new} breaks causal consistency, which leads to a contradiction. We thus conclude that after the reception of m_{k-1}, at least one server sends some message before eventually all writes of WOT are visible, and we construct E_{imp} to send m_k after the reception of m_{k-1}. □

3.6.3 Impossibility of fast invisible transactions

We sketch here the correctness of Theorem 8 in the general model. In the general model, the proof of Theorem 8 still goes by contradiction and we use the same assumption for contradiction, Proposition 4, recalled in Proposition 7. The notations $\mathcal{D}$, $\mathcal{D}_1$, S_1, M_1, t_0, T_2 and $\mathcal{D}_2$, S_2, M_2 follow the same definitions. The main steps remain the same: (1) we construct two executions E_1 and E_2 following Proposition 7; (2) we construct execution $E_{1,2}$ based on E_1 and E_2; and (3) we show that $E_{1,2}$ violates causal consistency. Our sketch below focuses on the construction of E_1, E_2 and $E_{1,2}$.

Proposition 7 (Assumption for contradiction)**.** *For any execution E_1 which schedules S_1 by fast transactions, for some $\mathcal{D}_2$, some execution E_2 where (1) $\mathcal{D}_1$ does not invoke S_1 but $\mathcal{D}_2$ invokes S_2 is the same as E_1 except for the message exchange events M_1 and M_2.*

Different from the previous construction, the construction of E_1 (E_2) starts with no transaction. Some client $C \notin \mathcal{D}$ writes, firstly, each object once. Suppose that at time t_{start}, these writes are all visible. Then we construct E_1 (E_2) starting from t_{start}. We consider a read-only transaction ROT which reads all objects. For $i \in \{1,2\}$, in E_i, every invocation in S_i of ROT starts at the same time t_0 and ends at the same time T_2. Every message which a client in $\mathcal{D}_i$ sends (to a server) arrives at some same time T_1. For each server, after T_1 and as long as this server is still about to send a message to a client in $\mathcal{D}_i$, the server receives no message from any other server. This time period for each server P while P receives no message from any other server is the same in E_1 and E_2. Each client in $\mathcal{D}_i$ receives at most one message from each server and returns ROT at time T_2.

After T_2, C writes again all objects. Let $o_1, o_2, \ldots, o_{n_{obj}}$ be the set of all objects. Then C executes writes $\underline{w} = w(o_i)v_i, i = 1,2,\ldots,n_{obj}$ sequentially, which establishes $\forall k \in \mathbb{Z}$, $2 \leq k \leq n_{obj}$, $w(o_{k-1})v_{k-1} \rightsquigarrow w(o_k)v_k$. All writes in $\underline{w}$ are eventually visible. Let τ be the time when $v_1, v_2, \ldots, v_{n_{obj}}$ are visible in both E_1 and E_2.

W.l.o.g., assume that $\mathcal{D}_1 \backslash \mathcal{D}_2 \neq \emptyset$. After τ, in both E_1 and E_2, one same client C_r in $\mathcal{D}_1 \backslash \mathcal{D}_2 \neq \emptyset$

requests the same read-only transaction ROT that reads all objects. (In E_1, C_r has requested ROT once, while in E_2, C_r has not.) We schedule these transactions according to Definition 11 and moreover, any message which C_r send to a server arrives at the same time. In either E_1 or E_2, for each server S, let $m_{resp,S}$ be the message which C_r receives from S. Let ss be the set of such server S that $m_{resp,S}$ reveals some version written by some write in $\underline{w}$. Let R be a server in ss. Note that $ss \backslash \{R\} \neq \emptyset$. Let Π be the set of all servers. The construction of $E_{1,2}$ is the same as that in the previous proof of Theorem 8 by substituting $\{R\}$ for P_Y and $\Pi \backslash \{R\}$ for P_X, which we sketch below.

The execution $E_{1,2}$ is based on E_1 and E_2 starting from t_0. Every client in $\mathscr{D}_1 \cup \mathscr{D}_2$ requests ROT that reads all objects at time t_0, $\Pi \backslash \{R\}$ receives the messages which $\mathscr{D}_1$ sends at the same time T_1 as in E_1, and R receives the messages which $\mathscr{D}_2$ sends at the same time as in E_2. (Clearly, those messages which $\mathscr{D}_1 \backslash \mathscr{D}_2$ sends to R are delayed as well as those messages which $\mathscr{D}_2 \backslash \mathscr{D}_1$ sends to $\Pi \backslash \{R\}$.) Therefore, by Proposition 7, $\Pi \backslash \{R\}$ and R reply to a client in $\mathscr{D}$ in the same way as in E_1 and E_2 respectively. The critical messages which $\Pi \backslash \{R\}$ sends to $C_r \in \mathscr{D}_1 \backslash \mathscr{D}_2$ are received at the same time as in E_1. If $\Pi \backslash \{R\}$ sends a non-critical message to C_r, then the non-critical message is delayed after the construction of $E_{1,2}$ completes. The rest of the schedule regarding messages between servers is the same as E_1 (E_2). Furthermore, after T_2, C issues $\underline{w}$ sequentially which is the same as E_1 (E_2).

After τ, R receives a message from C_r (a message previously delayed) at the same time as in E_2 while C_r requests ROT. By τ, R is unable to distinguish between $E_{1,2}$ and E_2 and thus by the time when R sends a critical message to C_r, R is still unable to distinguish between $E_{1,2}$ and E_2. As a result, R sends the same $m_{resp,R}$ to C_r in $E_{1,2}$ as E_2, which reveals some version written by some write in $\underline{w}$. We schedule C_r to receive $m_{resp,R}$ at the same time as in E_2 as well. Since C_r has not requested ROT before, the return value of ROT solely depends on these critical messages from Π. However, the critical messages received from $\Pi \backslash \{R\}$ are sent before $\underline{w}$ occurs and therefore may only reveal versions written by C in the first pass of writes to all objects. The return value of C_r's ROT then violates causal consistency. This gives a contradiction, showing that Proposition 7 is incorrect and therefore Theorem 8 is correct in the general model.

3.7 Related Work

3.7.1 Causal consistency

Ahamad et al. [42] were the first to propose *causal consistency* for a memory accessed by read/write operations. Raynal et al. [43] formally defined *causal transactions*. Bouajjani et al. [117] formalized the verification of causal consistency. A large number of systems [35, 118, 37, 36, 38, 41] implemented transactional causal consistency, while some [35] defined formally and strengthened causal consistency to include *convergence* property, which concerns the conflict resolution of two updates that are not causally related. Our results also hold for this strengthened causal consistency.

In the literature, extended notions of causal consistency are also proposed, considering non-transactional systems. These notions include *real time causal consistency* [119], which additionally respects the real-time order of any two operations. To formalize the consistency model of replication schemes (an issue orthogonal to the problem considered in this chapter), Attiya et al. [120] and Xiang and Vaidya [121] proposed related notions of causal consistency based on the events that are executed at the servers (rather than the histories of operations issued by the clients). More specifically, Attiya et al. [120] defined *observable causal consistency* for servers where (1) the program-order causality relation is tracked between clients' operations at the same server, but (2) there is no read-from causality relation defined and (3) the concurrent writes to the same object at different servers are resolved. Xiang and Vaidya [121] introduced *replica-centric causal consistency* where the causality relation is between the following two types of events at the servers: (1) the update issued by a server (meaning that the server receives the update from a client and then starts to propagate the update to other servers) and (2) the update applied by a server (meaning that the server receives the propagation of the update from another server).

3.7.2 Causal read-only transactions

Most implementations do not provide fast (read-only) transactions. COPS [35] and Eiger [36] provide a two-round protocol for read-only transactions. Read-only transactions in Orbe [37], GentleRain [38], Cure [40] and Occult [41] can induce more than one-round communication. Read-only transactions in ChainReaction [118] can induce more than one-round communication as well as abort and retry, resulting in more communication. Eiger-PS [44] provides fast transactions and satisfies *process-ordered serializability* [44], stronger than causal consistency; yet in addition to the request-response of a transaction, each client periodically communicates with every server. Our Theorem 7 explains Eiger-PS's additional communication. COPS-SNOW [44] provides fast read-only transactions but writes can only be performed outside a transaction; moreover, any read-only transaction in COPS-SNOW is visible, complying with our Theorem 7 and Theorem 8. If each data center is modelled as a process which stores a copy of all objects, then a transactional store, SwiftCloud [39], can provide fast read-only transactions (between a data center and a client). However, in addition to the request-response of a transaction, a data center can send a client a stream of update notifications [39]. Our Theorem 7 explains at least one of the two designs (the full copy and out-of-scope communication) is necessary.

3.7.3 Impossibility results

Existing impossibility results on storage systems have typically considered stronger consistency properties than causality or stronger progress conditions than eventual visibility. Brewer [106] conjectured the CAP theorem that no implementation guarantees *consistency*, and *availability* despite *network partitions*. Gilbert and Lynch [107] formalized and proved Brewer's conjecture in *partially synchronous* systems. They formalized consistency by *atomic* objects

[105] (which satisfy *linearizability* [104], stronger than causal consistency). Considering a storage implemented by *data centers* (clusters of servers), if any value written is *immediately* visible to the reads at the same data center (to which the write request is sent), and some client can access two objects at two data centers respectively, then Roohitavaf et al. [122] proved the impossibility of ensuring causal consistency and availability despite network partitions. Their proof (as well as the proof of the CAP Theorem) relies on message loss in face of network partition. On the contrary, our impossibility results do not assume message loss, and thus are not implied by their proof.[10] Lu et al. [44] proved the SNOW theorem, saying that fast *strict serializable* transactions [112, 113] (satisfying stronger consistency than causal consistency) are impossible. As strict serializability is stronger than causal consistency, the SNOW theorem does not imply our results.

The impossibility result, the CAC theorem [119] states that no implementation guarantees *one-way convergence*, availability, and any consistency stronger than real time causal consistency assuming infinite local clock events and arbitrary message loss, in the model where each pair of processes can communicate. (By contrast, in our model, we assume two clients do not communicate.) Here one-way convergence [119] is a progress property conditioned on the communication between each pair of processes (rather than a progress property of a client's read, different from our definition of eventual visibility). This turns the CAC theorem an impossibility result for replication schemes (an issue orthogonal to the problem considered in this chapter). As mentioned earlier, Attiya et al. [120] and Xiang and Vaidya [121] formalized some related notions of causal consistency in the context of replication schemes. According to their notions, Attiya et al. [120] proved that a replicated store implementing multi-valued registers cannot satisfy any consistency strictly stronger than observable causal consistency, while Xiang and Vaidya [121] proved that for replica-centric causal consistency, it is necessary to track down writes.

3.7.4 Transactional memory

In the context of transactional memory, if the implementation of a read-only operation (in a transaction) writes a base shared object, then the read-only operation is said to be *visible* and *invisible* otherwise [123]. Known impossibility results on invisible reads of TM assume stronger consistency than causal consistency. Attiya et al. [124] showed that no TM implementation ensures strict serializability, *disjoint-access parallelism* [124][11] and uses invisible reads, the proof of which shows that if writes are frequent, then a read-only transaction can not terminate in a finite number of steps. Peluso et al. [125] considered any consistency that respects the real-time order of transactions (which causal consistency does not necessarily respect), and proved a similar impossibility result. Perelman et al. [126] proved an impossibility result for a

---

[10] Although the CAP theorem can be considered as an impossibility result of a strongly consistent replication system, as we do not assume message loss, even in our extended model of replicated storage systems, our impossibility results are not implied by the CAP theorem or its proof.

[11] Disjoint-access parallelism [124] requires two transactions accessing different application objects to also access different base objects.

multi-version TM implementation which provides invisible read-only transactions, ensures strict serializability and maintains only a necessary number of versions; the proof of this impossibility result focuses on garbage collection of versions [126]. None of the results or proofs above imply our impossibility results.

3.8 Concluding Remarks

Our impossibility results establish fundamental limitations on the performance on transactional storage systems. The first impossibility basically says that fast read-only transactions are impossible in a general setting where writes can also be performed within transactions. The second impossibility says that in a setting where all transactions are read-only, they can be fast, but they need to visible. A system like COPS-SNOW [44] implements such visible read-only transactions that leave traces when they execute, and these traces are propagated on the servers during writes. Recall that we provide in Section 3.5 a variant algorithm where these traces are propagated outside writes, demonstrating that the complexity of these traces does not arise due to writes.

Clearly, our impossibilities apply to causal consistency and hence to any stronger consistency criteria. They hold without assuming any message or node failures and hence hold for failure-prone systems. In Section 3.3 and Section 3.4, for presentation simplicity, we assumed that servers store disjoint sets of objects, but our impossibility results hold without this assumption as shown in Section 3.6. Some design choices could circumvent these impossibilities like imposing a full copy of all objects on each server (as in SwiftCloud [39]), or periodic communication between servers and clients (as in Eiger-PS [44]). Each of these choices clearly hampers scalability.

We considered an asynchronous system where messages can be delayed arbitrarily and there is no global clock. One might ask what happens with synchrony assumptions. If we assume a fully synchronous system where message delays are bounded and all processes can access a global accurate clock, then our impossibility results can both be circumvented. We give such a timestamp-based algorithm in Section 3.5. If we consider however a system where communication delays are unbounded and all processes can access a global accurate clock, then only our Theorem 7 holds (while our timestamp-based algorithm can still circumvent Theorem 8). In this sense, message delay is key to the impossibility of fast read-only transactions, but not to the requirement that they need to be visible in the restricted model where all transactions are read-only (and writes are outside the scope of a transaction).

4 The Complexity of Optimistic Secure Transactions[1]

4.1 Introduction

In *fair* computation (of a deterministic function) [48, 6], n parties possess n pieces of information and need to output the function of these n pieces of information (the inputs) *atomically*. Namely, a party obtains the output of the function if and only if the other $n-1$ parties obtain the same output. A prominent example is auctions: after n parties offer a price for some item, they wish to determine the highest price and the winner without ambiguity, e.g., when more than one party claims to win the item. A solution is the fair computation of the n bids (prices).

The difficulty of fair computation stems from the fact that a party might be *malicious* (dishonest) and try to obtain other parties' inputs, twist other parties' output, or arbitrarily delay other parties from obtaining an output. Still, honest parties should eventually obtain an output in a fair manner: they should all obtain the function of the n inputs, or all obtain a specific value $\perp$ (denoted *abort* in [48]). In an asynchronous context, rather than waiting forever for some message, any party may decide to *stop* the computation. Such ability of a party to stop at any time without jeopardizing *fairness* has been called *timely termination* [48]. As a matter of fact, fair computation is in general impossible without a trusted third party [50]. Yet, this third party is not needed in every execution of a fair computation protocol.

Optimistic fair computation stipulates that the third party does not need to be invoked if all n parties are honest [48, 6, 128]. An execution where n honest parties output without invoking the third party is called an *optimistic* execution [48, 128]. Given that cheating is seldom and the third party is considered a bottleneck, optimism is practically appealing. To claim true practicality, however, optimistic executions should be efficient. To be specific, the number of messages exchanged among n honest parties (which compute the function without resorting to the third party) should not be prohibitive. Until our work (presented in this chapter), the optimal number of messages was unknown.

---

[1] Postprint version of the article published in DISC 2016: Rachid Guerraoui and Jingjing Wang. *"Optimal Fair Computation"* [127]

We prove in this chapter that $\ell + 2n - 3$ is the optimal number of messages that an optimistic execution of optimistic fair computation may achieve in the presence of $n-1$ potentially malicious parties in an asynchronous network, where ℓ is the length of the shortest sequence that contains all permutations of n symbols as subsequences [129]. Given recent results in combinatorics [58, 59, 60, 130], the optimal number of messages for optimistic fair computation is 4 for $n = 2$, $n^2 + 1$ for $3 \leq n \leq 7$, and asymptotically $\Theta(n^2)$ for $n \geq 8$.[2]

The main idea behind our proof of the $\ell + 2n - 3$ lower bound is the identification of a *decision propagation* pattern according to which an honest party reaches an agreement with the others. The decision propagation occurs when some party decides to stop the computation. The pattern can be between any two parties P and Q. To get an intuition, consider an optimistic execution E, let event E_P ="P receives message m_P" and let event E_Q ="Q receives message m_Q". Let $\bar{e}$ be the complement of an event e. To ensure timely termination in an asynchronous network, an honest party P (Q)'s stop could result from $\bar{E}_P$ ($\bar{E}_Q$). However, a malicious P's stop can impose an honest Q's stop by pretending $\bar{E}_P$. If when P and Q complete E, E_P occurs before E_Q and Q does not receive any message between E_P and E_Q, then before E_Q happens, Q is unable to distinguish whether E_P or $\bar{E}_P$ occurs. As a result, malicious P's decision may *propagate* to honest Q here. To prevent *fairness* from being jeopardized by malicious propagation, in the context of possibly $n-1$ malicious parties, every party should participate in this propagation so that none has a chance to pretend being honest.

This yields a subsequence of n events E_P (one for each party P) and n messages (whose destinations are the n parties) in E. Clearly, the order of the parties does not matter and therefore, any *permutation* of the n events must occur as a subsequence in E. Hence we establish a relation between the least number of messages of an optimistic execution and ℓ, the length of the shortest sequence that contains all permutations of n symbols as subsequences.

Our lower bound on the number of messages is tight in the following sense. We present an $(\ell + 2n - 3)$-message optimistic fair computation scheme of some function f given a shortest permutation sequence $\underline{s}$. Our protocol, where the n parties are honest and compute without the third party, consists of three phases: (a) the n parties send *verifiable encryption* [133] of their n inputs respectively, in case they recover those inputs (if needed) in the *non-optimistic* execution, which defines the first n messages; (b) the n parties exchange $\ell - 2$ messages defined by $\underline{s}$; and (c) the n parties exchange the concatenation of the n inputs, which defines the last $n-1$ messages. The $\ell - 2$ messages $m_1 m_2 \ldots m_{\ell-2}$ in phase (b) have their sources and destinations defined by the sequence $\underline{s} = s_1 s_2 \ldots s_\ell$ as follows. The party represented by symbol s_j is the source of m_{j-1} for $j = 2, \ldots, \ell - 1$, and the destination of m_{j-2} for $j = 3, 4, \ldots, \ell$. (s_1 is the source of the last message m_0 of phase (a) and s_2 is the destination of m_0.) When a party resorts to T in a non-optimistic execution, T follows the idea of decision propagation to

---

[2] Newey [58] (and then many others [59, 60, 130, 131, 132]) studied the length ℓ of the shortest permutation sequence. Although Newey [58] showed that $\ell = 3$ for $n = 2$, and $\ell = n^2 - 2n + 4$ for $3 \leq n \leq 7$, the exact ℓ for $n \geq 8$ is still considered as an open problem [59, 60]. Up until now, the best upper bound is $\lceil n^2 - \frac{7}{3}n + \frac{19}{3} \rceil$ for $n \geq 7$ [60], while a lower bound of ℓ is of the form $n^2 - cn^{7/4} + \epsilon$ for some constant c and some $\epsilon > 0$ [130].

decide an output. The pattern is the same as shown in our proof of the lower bound so that the number of messages in every optimistic execution is minimal.

As we will explain in Section 5, many results have been published on problems related to fair computation [51, 52, 53, 54, 55, 56]. None implies our lower bound. On the other hand, our $(\ell + 2n - 3)$-message optimistic fair computation scheme can be used to implement fair exchange of certain digital signatures (including Schnorr signatures [134], DSS signatures [135], Fiat-Shamir signatures [136], Ong-Schnorr signatures [137], GQ signatures [138]). Thus, our scheme is also a message-optimal optimistic fair exchange scheme [48]. Moreover, combined with our proof of the lower bound, this optimistic fair exchange scheme of digital signatures also implies that $\ell + 2n - 3$ is the optimal number of messages for optimistic fair contract signing [54].

The rest of this chapter is organized as follows. Section 4.2 presents our general model and defines optimistic fair computation. Section 4.3 presents our lower bound on the number of messages. Section 4.4 presents our $(\ell + 2n - 3)$-message optimistic fair computation scheme. Section 4.5 discusses related work.

4.2 Model and Definitions

4.2.1 The parties

We consider a set Ω of n parties $P_1, P_2, \ldots, P_n$ (sometimes also denoted by P, Q). These parties are all *interactive* in the sense that they can communicate with each other by exchanging messages. All parties are *computationally-bounded* [139] in the sense that they run in time polynomial in some security parameter s.[3]

In addition to the n parties, we also assume a trusted third party T. T follows the protocol assigned to it. The communication with T is such that when T is communicating with P_i, P_j needs to wait for P_j's turn to communicate with T for any two parties $P_i, P_j \in \Omega, i, j \in \{1, 2, \ldots, n\}$. We assume that T is also computationally bounded.

At most $n - 1$ parties can be *malicious*. A malicious party could deviate arbitrarily from the protocol assigned to it. The malicious party could interact arbitrarily with the others as well as T. For example, a malicious party may drop certain messages. A party that *crashes* at some point in time is considered as a malicious party that drops all the messages from that point. Malicious parties may also collude. (The goal of malicious parties and their collusion can be breaking *fairness*, e.g., to obtain an output for themselves and to prevent an output to an honest party. Fairness is defined formally later in Definition 22.)

Communication channels do not modify, inject, duplicate or lose messages. Every message

---

[3] Hereafter, when we say that a probability is negligible, we mean that the probability is a *negligible* function $g(s)$ of the security parameter s; i.e., $\forall c \in \mathbb{N}$, $\exists C \in \mathbb{N}$ such that $\forall s > C$, $g(s) < \frac{1}{s^c}$. The definition of negligible function is later repeated in Definition 24.

sent eventually reaches its destination. Any modified, injected, duplicate, or lost message is considered to be due to malicious parties. The delay on message transmission is finite but unbounded. Messages could be reordered. Communication channels are authenticated and made secure by Transport Layer Security [140]. No party can be masqueraded and no message can be eavesdropped.

4.2.2 Fair computation

We consider the problem of optimistic fair computation in the classical sense of [6, 48]. The problem involves a deterministic function f to be computed by the n parties. Function f is agreed upon by the n parties in advance. We assume that f takes n strings $x_1 \in \{0,1\}^{\ell_1}, x_2 \in \{0,1\}^{\ell_2}, \ldots, x_n \in \{0,1\}^{\ell_n}$ as inputs and returns $z \in \{0,1\}^{\ell_z}$ as its output.

Definition 20 (Computation)**.** A *computation* scheme for f is a collection $(P_1, P_2, \ldots, P_n)$ of n algorithms. The algorithms can carry out two interactive protocols:[4]

- *Compute*: $P_i, i = 1, 2, \ldots, n$ is initialized with a local input x_i. If P_i finishes this protocol, P_i returns a local output o_i which can take the following values: $z \in \{0,1\}^{\ell_z}$ or $\perp$. If Compute is interrupted by Stop (which we introduce below), Compute returns the same output as Stop.

- *Stop*: This is the protocol invoked by P_i when P_i wants to stop the computation. P_i can invoke this protocol at any point in time. P_i obtains P_i's *status* of Compute so far (i.e., the sequence of messages that have arrived at P_i so far) as a local input to Stop. P_i makes a local output o_i which can take the following values: $z \in \{0,1\}^{\ell_z}$, or $\perp$.

In the classical definition of fair computation [6], the problem is defined in the *simulatability paradigm* [5], which basically expresses a correct solution to fair computation in terms of a simulation of the *ideal process*. In what follows, we recall the notion of the ideal process (Definition 21), and then fair computation (Definition 22).

Definition 21 (Ideal process [6])**.** The *ideal* process for *fair* computation of f is a collection $(\bar{P}_1, \bar{P}_2, \ldots, \bar{P}_n, U)$ of $n+1$ deterministic algorithms. $\bar{P}_i, i = 1, 2, \ldots, n$ is initialized with a local input x_i. U is parameterized by f. $\bar{P}_i$ sends message $a_i = x_i$ to U. Messages are delivered instantly. U returns a message m_i to P_i according to Equation (4.1) as soon as $a_1, a_2, \ldots, a_n$ have arrived at U or one message of $\perp$ has arrived at U. $\bar{P}_i$ outputs whatever U returns to it.

$$\forall i \in \{1, 2, \ldots, n\}, m_i = \begin{cases} f(a_1, a_2, \ldots, a_n) & \text{if } a_1 \neq \perp, a_2 \neq \perp, \ldots, a_n \neq \perp \\ \perp & \text{if } \perp \in \{a_1, a_2, \ldots, a_n\} \end{cases} \tag{4.1}$$

---

[4] We consider Compute and Stop as such type of protocols that a party does not randomly choose whether to send a message or not, or the party to whom a message is sent. Nevertheless, the contents of messages exchanged are allowed to be randomized.

The process is *ideal* in the sense that among $n+1$ parties, the information of a private input is only exposed to the *universally trusted* U. We explain the meaning of this universal trust when we present Definition 22. In Definition 22, collusion between malicious parties is represented as malicious parties controlled by an adversarial algorithm $\mathscr{A}$. In this case, $\mathscr{A}$ also controls the communication in the sense that $\mathscr{A}$ can delay messages arbitrarily. In addition, Definition 22 distinguishes between the case where all parties are honest, for which we define the *completeness* property, and the case where at least one party is malicious, for which we define the *fairness* property. We remark that the fairness property here encompasses both fairness and *privacy*. As shown by Definition 22, even malicious parties who try to obtain other parties' private inputs do not learn any information beyond whatever an honest party can, i.e., whatever is revealed by the computation result of the function.

Definition 22 (Fair computation[5]). A computation scheme α solves *fair* computation for f [6] if it satisfies the following properties:

- *Fairness*: for any $e \in \mathbb{N}, 1 \leq e \leq n-1$ and any e malicious parties $P_{d_1}, P_{d_2}, \ldots, P_{d_e}$, for any computationally bounded algorithm $\mathscr{A}$ that controls the e malicious parties[6], there exists a computationally bounded algorithm $\mathscr{S}$ that controls $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$[7] such that for any $x_1, x_2, \ldots, x_n$, $O_{P_1, P_2, \ldots, P_n, \mathscr{A}}(x_1, x_2, \ldots, x_n)$ and $O_{\bar{P}_1, \bar{P}_2, \ldots, \bar{P}_n, \mathscr{S}}(x_1, x_2, \ldots, x_n)$ are computationally indistinguishable [141, 142];
- *Termination*: If an honest party P_i invokes Stop, then P_i eventually outputs.
- *Completeness*: $\forall x_1, x_2, \ldots, x_n$, if $P_1, P_2, \ldots, P_n$ are honest and none invokes Stop, then all parties output $z = f(x_1, x_2, \ldots, x_n)$; if $P_1, P_2, \ldots, P_n$ are honest and some invokes Stop, then either all parties output $z = f(x_1, x_2, \ldots, x_n)$, or all parties output $\perp$.
- *Non-triviality*: There is at least one execution in which $P_1, P_2, \ldots, P_n$ are honest and none invokes Stop.

Assumptions and notations:

- w.l.o.g., $P_{d_1}, P_{d_2}, \ldots, P_{d_e}$ output nothing but $\mathscr{A}$ may output arbitrarily[8], and similarly, $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$ output nothing but $\mathscr{S}$ may output arbitrarily; and

---

[5] The original definition in [6] is ambiguous when all parties are honest: (1) if the asynchronous network delays every message, then to ensure termination, every honest party should output $\perp$ at some point in time; however, by the original definition, all honest parties output z, except with negligible probability, which yields a contradiction; and (2) if in a protocol, all parties send no message and only outputs $\perp$, then by the original definition, this protocol also matches the ideal process, which however is a trivial protocol.

[6] $\mathscr{A}$ also plays the role of the asynchronous network as defined in Section 4.2. The probability of the joint output between honest parties and an adversarial algorithm is taken over the randomness of the adversarial algorithm.

[7] In the ideal process, $\mathscr{S}$ sees $x_{d_1}, x_{d_2}, \ldots, x_{d_e}$, may change $a_{d_1}, a_{d_2}, \ldots, a_{d_e}$ and also sees $m_{d_1}, m_{d_2}, \ldots, m_{d_e}$ but $\mathscr{S}$ cannot see other messages from or to U, or U's internal state (which makes U *universally trusted*).

[8] The assumption that a malicious party outputs nothing is for definition only. In practice, a malicious party may output arbitrarily.

- $O_{P_1,P_2,\dots,P_n,\mathscr{A}}(x_1,x_2,\dots,x_n)$ denotes the joint output of $P_1,P_2,\dots,P_n,\mathscr{A}$ when running α for $x_1,x_2,\dots,x_n$, and $O_{\bar{P}_1,\bar{P}_2,\dots,\bar{P}_n,\mathscr{S}}(x_1,x_2,\dots,x_n)$ denotes the joint output of $\bar{P}_1,\bar{P}_2,\dots,\bar{P}_n,\mathscr{S}$ when running the ideal process for $x_1,x_2,\dots,x_n$.

Definition 23 (Optimistic fair computation)**.** A fair computation scheme is *optimistic* [48] if it satisfies the following property.

- *Optimism*: $\forall x_1,x_2,\dots,x_n$, if $P_1,P_2,\dots,P_n$ are honest and none invokes Stop, then all parties output $z = f(x_1,x_2,\dots,x_n)$ without interacting with T.

When $P_1,P_2,\dots,P_n$ are honest and none invokes Stop, $P_1,P_2,\dots,P_n$ carry out Compute only. In this case, an optimistic execution is an execution of Compute, where every party finishes all communication steps of Compute and outputs.

We focus on the class $\mathscr{C}$ of function f such that for any $x_1 \in \{0,1\}^{\ell_1}, x_2 \in \{0,1\}^{\ell_2},\dots,x_n \in \{0,1\}^{\ell_n}$, given any $n-1$ out of n strings, there are at least two possibilities for the evaluation of $f(x_1,x_2,\dots,x_n)$ considering all possibilities of the missing string (e.g., if $x_1,x_2,\dots,x_{n-1}$ are given, then x_n is the missing string). For a function f in the complement of $\mathscr{C}$, a protocol that solves optimistic fair computation can still be vulnerable to the following attack: a subset of parties colludes, leaves with the evaluation of f immediately but an honest party outputs $\perp$. In the literature [143, 144], fair protocols for the complement of $\mathscr{C}$ are considered, but they ensure fairness different from Definition 21 and Definition 22 and are not the focus here. We also assume that T does not have prior knowledge of $x_1,x_2,\dots,x_n$. Therefore no computationally bounded algorithm, even with the help of T, is able to evaluate f from any $n-1$ out of the n inputs of $P_1,P_2,\dots,P_n$ for any missing input with non-negligible probability. We call this assumption the *no prior knowledge of* T.

4.3 Lower Bound

In this section, we prove our lower bound on the number of messages exchanged during an optimistic execution of optimistic fair computation. We first present an overview of our proof and then formally prove our lower bound. Our proof of lower bound starts with preliminaries (including a formal definition of indistinguishability) and follows the main idea presented in the overview.

Recall that we consider those functions that cannot be evaluated by only a subset of n parties, e.g., we do not consider constant functions. In addition, a scheme (or the Compute protocol of a scheme) which sends no message, invokes Stop and outputs $\perp$ only is excluded by the *non-triviality* property (Definition 22). Thus the lower-bound is non-zero.

In Theorem 9, we express our lower bound in terms of n and ℓ, the length of the shortest sequence that contains all permutations of n symbols as subsequences.

Theorem 9 (Message complexity). *For any function $f \in \mathscr{C}$, for any optimistic fair computation scheme for f (for n parties, among which $n-1$ can be malicious), the n parties exchange at least $\ell + 2n - 3$ messages in every optimistic execution.*

4.3.1 Proof overview and intuition

To have a better understanding of our proof of lower bound, we present an overview as well as intuition which covers the main points of our proof. A detailed proof is presented later. To prove Theorem 9, we count the number of messages in every optimistic execution. We view every optimistic execution E as a sequence of messages ordered according to when they reach their destinations respectively. We first pinpoint two necessary messages in E, and then we show that between these two messages, there must exist certain patterns of messages.

Intuitively, when starting E, no party knows anything about other parties' inputs; there is a border-line message m_1^* such that, after m_1^* reaches its destination, one and only one party knows something about all the other parties' inputs. If any honest party $P_i \in \Omega$ stops before m_1^* arrives at its destination, then P_i has no hope of outputting $z = f(x_1, x_2, \dots, x_n)$ even with the help of T, by the *no prior knowledge of* T.

By the end of E, every party receives sufficient messages to compute z (by the *optimism* property); there is another border-line message m_2^* such that, after m_2^* reaches its destination, one and only one party has sufficient messages to compute z. If any honest party P_i stops after m_2^* arrives at its destination, P_i outputs z by the *completeness* property (with or without the help of T). Figure 4.1a illustrates the two messages.

(a) P_i outputs $\perp$ if P_i stops before m_1^*; and z if P_i stops after m_2^*

(b) P_i decides the same value v after P_j if P_i stops between m_1^* and m_2^*.

Figure 4.1 – The output of P_i if P_i stops at some point in execution E

What P_i should output if it stops between m_1^* and m_2^* requires a closer look. Suppose that when P_i wants to stop, P_i has not received some message m_i. (We clarify some terminology here. When we say that P_i has not received or does not receive some message m_i, we mean that P_i has not received m_i but received every message with destination P_i before m_i in E. The terminology applies to any party hereafter.) When P_i wants to stop, either no other party has decided an output (and then P_i can easily decide), or some party $P_j \in \Omega, j \neq i$ has decided. If P_j claims that it has not received message m_j and m_i is the first message with destination P_i after m_j in E, then P_j's decision *propagates* to P_i. Clearly, if P_j is honest, then P_i has to decide the same output as P_j (except with negligible probability) by the *fairness* property. Figure 4.1b illustrates this agreement.

This agreement between two parties induces a *decision propagation* pattern, which gives rise to a certain pattern of messages in E. When P_j is honest and stops due to the missing message m_j, P_j needs to enforce P_i to stop and agree on their decisions. Thus in the sequence of messages (ordered at the beginning of our proof overview), after a message m_j with destination P_j, there must exist a message m_i with destination P_i so that P_j could enforce P_i on the same output if (a) P_j does not receive m_j, (b) P_j invokes Stop and outputs $\perp$, and (c) P_i does not receive m_i and invokes Stop.

Because $n-1$ parties can be malicious, we use this decision propagation pattern to build the following scenario. The scenario also connects a party's output before m_1^* and a party's output after m_2^* to the decision propagation. Suppose one party P_1 stops before m_1^* arrives at its destination and then the other $n-1$ parties stop following the decision propagation pattern above: for $k=1$, we denote by m_1 the message which P_1 has not received when P_1 stops; then for $k=2,3,\ldots,n$, if there is a message m_k in E that is the first message with destination P_k between m_{k-1} and m_2^* , then P_k stops when P_k has not received m_k, and if not, P_k stops after m_2^* arrives at its destination.

Clearly, if the pattern of the n messages whose destinations are $P_1, P_2, \ldots, P_n$ does not exist between m_1^* and m_2^* in E, then P_n would output z by the property of m_2^*. However, P_1, as well as other parties $P_2, P_3, \ldots, P_{k-1}$ for which messages $m_2, m_3, \ldots, m_{k-1}$ exist, would output $\perp$ by the property of m_1^* and decision propagation. As P_{k-1} can be an honest party, this would violate the *fairness* property. Therefore, the pattern of the n messages whose destinations are n parties, or in fact any permutation of the n parties must exist as a subsequence of E between between m_1^* and m_2^*.

Thus, the number of messages between m_1^* and m_2^* (inclusive) of E is at least ℓ. In the meantime, in E, before m_1^*, there are at least $n-1$ messages to meet the definition of m_1^* and after m_2^*, there are at least $n-2$ messages to meet the definition of m_2^*. We add together the minimum numbers of messages before m_1^*, after m_2^* and between m_1^* and m_2^*, and then have $\ell+2n-3$ as the final minimum number of messages during every optimistic execution.

4.3.2 Full proof of Theorem 9

We now give a detailed proof of Theorem 9. The full proof is organized as follows. First we give the *(weak) fairness* property that we use repeatedly in the proof. To show this property, we recall the formal definition of computational indistinguishability to elaborate the definition of fairness in Section 4.2. Second, we present some preliminary assumptions, without the loss of generality, on Stop for the simplicity of the presentation of of our proof. Finally, we show the main part of our proof. The proof overview captures the main idea of our proof. Thus not surprisingly, the main part of our proof starts with two necessary messages m_1^* and m_2^*, and then proceeds to show that between these two messages, there must exist certain patterns of messages. We next count all the necessary messages before m_1^*, after m_2^* and messages in between respectively and complete our proof.

(Weak) fairness

First, we give the *(weak) fairness* property that we use repeatedly in the proof.

Lemma 11 ((Weak) fairness)**.** *If a computation scheme α solves fair computation, then it satisfies the following propety. For any $e \in \mathbb{N}$, any $1 \le e \le n-2$, any e malicious parties and any computationally bounded algorithm $\mathscr{A}$ that controls the e malicious parties, $\forall x_1, x_2, \ldots, x_n$, any two honest parties $P_i, P_j, i, j, \in \{1, 2, \ldots, n\}$ output the same except with negligible probability.*

We show that this property is implied by the *fairness* property in Definition 22. Before proving the property, we recall formal definitions and terminologies used in Definition 22 such as computational indistinguishability and a negligible function from their classical definitions in [141, 142].

Definition 24 (Computational indistinguishability)**.** If function g is a negligible function of variable s, then $\forall c \in \mathbb{N}, \exists C \in \mathbb{N}$ such that $\forall s > C$, $g(s) < \frac{1}{s^c}$.

Let $A = \{A(1^s, a)\}$ be a distribution ensemble, i.e., random variables indexed by 1^s and a. Let $B(1^s, a) = \{B(1^s, a)\}$ be also a distribution ensemble. Then A and B are *computationally indistinguishable*, if for any computationally bounded algorithm $\mathscr{D}(1^s, a, w, D)$ that takes q independently identically distributed random variables following the distribution D,

$$|Pr[\mathscr{D}(1^s, a, w, A(1^s, a)) = 1] - Pr[\mathscr{D}(1^s, a, w, B(1^s, a)) = 1]| = negl(s), \forall a, \forall w$$

where $negl(s)$ is a negligible function of s, $q = q(s)$ is a polynomial of s and the probabilities are taken over the random choices of $\mathscr{D}$ and q random variables of D.

In the context of Definition 22, s is the security parameter of the fair computation scheme. Recall that in Definition 22, we say that the joint outputs $O = O_{P_1, P_2, \ldots, P_n, \mathscr{A}}(x_1, x_2, \ldots, x_n)$ and $\bar{O} = O_{\bar{P}_1, \bar{P}_2, \ldots, \bar{P}_n, \mathscr{S}}(x_1, x_2, \ldots, x_n)$ are computationally indistinguishable. This means that for any computationally bounded algorithm $\mathscr{D}(1^s, a, w, D)$,

$$|Pr[\mathscr{D}(1^s, a, w, O) = 1] - Pr[\mathscr{D}(1^s, a, w, \bar{O}) = 1]| = negl(s), \forall a, \forall w$$

where $a = x_1 || x_2 || \cdots || x_n$, w may be arbitrary auxiliary information which is publicly known and both O and $\bar{O}$ are indexed by 1^s and a.

Proof of Lemma 11. Consider a computationally bounded algorithm $\mathscr{A}$ that does not control P_i or P_j. Let o_i, o_j be the random variables that represent P_i and P_j's outputs in the joint output O respectively. Suppose that $\mathscr{A}$ controls $e, 1 \le e \le n-1$ malicious parties $P_{d_1}, P_{d_2}, \ldots, P_{d_e}$.

By Definition 22, there exists a computationally bounded algorithm $\mathscr{S}$ that controls $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$ such that O and $\bar{O}$ are computationally indistinguishable. Let $\bar{o}_i$ and $\bar{o}_j$ be the random variables that represent $\bar{P}_i$ and $\bar{P}_j$'s outputs in the joint output $\bar{O}$ respectively. Since $\mathscr{S}$ does not control $\bar{P}_i$ or $\bar{P}_j$, $\bar{o}_i = \bar{o}_j$ with probability 1.

Consider a computationally bounded algorithm $\mathscr{D}$ that tries to distinguish O and $\bar{O}$ as follows. $\mathscr{D}$ takes one sample from the given distribution D. If in the sample, the ith element and the jth element are the same, then $\mathscr{D}$ outputs 1; if not, $\mathscr{D}$ outputs 0. Then there exists a negligible function $negl(s)$ such that

$$|Pr[\mathscr{D}(1^s, a, w, O) = 1] - Pr[\mathscr{D}(1^s, a, w, \bar{O}) = 1]| = negl(s), \forall a$$

where $a = x_1||x_2||\cdots||x_n$ and w is an empty string.

Since $\bar{o}_i = \bar{o}_j$ with probability 1, $Pr[\mathscr{D}(1^s, a, w, \bar{O}) = 1] = 1$. Let ρ be the probability such that $o_i = o_j$. Then $Pr[\mathscr{D}(1^s, a, w, O) = 1] = \rho$. Thus $\rho = 1 - negl(s)$. I.e., for any algorithm $\mathscr{A}$, any two honest parties $P_i, P_j, i, j, \in \{1, 2, \ldots, n\}$ output the same except with negligible probability. □

Then, we discuss some essential properties/convention of Stop, which we use later in the proof.

Preliminaries

Here we make some assumption on Stop.

If P invokes Stop several times, Stop returns the same value as the first time.

P may communicate with T in Stop, but P does not communicate with T in Compute. This is consistent to the *optimism* property.

When P invokes Stop, either P does not send messages to any other party including T and simply terminates, or P communicates with T and then terminates. If P communicates with T, P sends only one stop request. T does not ask any party (including P[9]) for additional messages when computing an output for P. This is due to the atomicity of the communication with T and the *termination* property.

When P communicates with T and then terminates, T sends a response only to P. In the asynchronous network, even if T sends messages to parties other than P, they might receive the messages after they complete Compute or Stop in the worst case. Thus we consider that T does not send messages to other parties.

We say that an optimistic execution E is initialized with $x_1, x_2, \ldots, x_n$, if n parties in E are initialized with $x_1, x_2, \ldots, x_n$. When we discuss any optimistic execution E, E must have been initialized with some n strings. Thus the term E initialized with (some) $x_1, x_2, \ldots, x_n$ does not lose generality.

---

[9] Since P communicates only with T, then in this case, P can simply send P's status and P's local input to T and T does not need to ask P for additional messages.

Sometimes we denote a party by O, P, Q, R, with an abuse of notations on O and R (as their meaning is clear in the context).

Full proof

Recall the intuition (Section 4.3.1) that there are two necessary messages (of every optimistic execution). Here we precisely define the two messages, m_1^* and m_2^*, and show their basic properties. Lemma 12 and Corollary 1 define m_1^* and prove a property of m_1^*; Lemma 13 defines m_2^* and Corollary 3 and Lemma 14 show properties of m_2^*. Corollary 2 confirms the intuition of the order between two events: the arrival of m_1^* and the arrival of m_2^*.

Lemma 12. *For any optimistic execution E, for any two parties P and Q, we say that P contacts Q in E if one of the two properties below holds: (a) P sends m to Q and Q receives m; or (b) there exists a party O such that P contacts O and subsequently O contacts Q.*

Then for any optimistic execution E and any $P \in \Omega$, there exists a message m such that before m arrives at its destination, $\exists Q \in \Omega \backslash \{P\}$ such that Q has not contacted P yet and after m arrives at its destination, $\forall Q \in \Omega \backslash \{P\}$, Q has contacted P.

Thus P is the destination of m. Let t be any status of P before P receives m in E. Then if P invokes Stop with t and no other party has invoked Stop, then Stop returns $\perp$ to P.

Proof. The lemma contains two parts. We first prove the existence of message m. By contradiction. Suppose that for some optimistic execution E initialized with $x_1, x_2, \ldots, x_n$ and some $P \in \Omega$, after E finishes, $\exists Q \in \Omega \backslash \{P\}$ has not contacted P yet. Then by the *optimism* property, P performs a computationally bounded algorithm that computes $f(x_1, x_2, \ldots, x_n)$ given only $\Omega \backslash \{Q\}$'s inputs. A contradiction.

Second, we prove that if P invokes Stop with t and no other party has invoked Stop, then Stop returns $\perp$ to P. Since no other party has invoked Stop, Stop is only able to return to P a value based on t, P's input and T's input. Let E be initialized with $x_1, x_2, \ldots, x_n$. Since t is P's status in E before P receives m, $\exists Q \in \Omega \backslash \{P\}$ has not contacted P yet and thus t can be constructed given only $\Omega \backslash \{Q\}$'s inputs. Since E is an optimistic execution, then by the *completeness* property, if Stop returns a non-$\perp$ value, Stop returns $z = f(x_1, x_2, \ldots, x_n)$. Suppose that Stop returns z to P. Then there is a computationally bounded algorithm that evaluates f given only $\Omega \backslash \{Q\}$'s inputs and T's inputs, which gives a contradiction. □

Corollary 1. *For any optimistic execution E, there exists message m_1^* such that (a) before m_1^* arrives at its destination, $\forall P \in \Omega$, $\exists Q \in \Omega \backslash \{P\}$ such that Q has not contacted P yet and (b) after m_1^* arrives at its destination, there exists the destination R of m_1^* such that $\forall Q \in \Omega \backslash \{R\}$, Q has contacted R.*

Proof. The correctness follows from Lemma 12. □

Lemma 13. *For any optimistic execution E initialized with $x_1, x_2, \ldots, x_n$, there exists message m_2^* such that (a) before m_2^* arrives at its destination R, no P computes $z = f(x_1, x_2, \ldots, x_n)$ from P's status and P's input (according to the protocol underlying E) and (b) after m_2^* arrives at R, R computes z from R's status and R's input (according to the protocol underlying E).*

In E, before R receives m_2^*, $\forall P \in \Omega \backslash \{R\}$, P has been contacted by Q, $\forall Q \in \Omega \backslash \{P\}$.

Proof. The lemma contains two parts. The existence of message m_2^* follows from the *optimism* property.

We prove the second part by contradiction. Suppose that in E, $\exists O \in \Omega \backslash \{R\}, Q \in \Omega \backslash \{O\}$ such that when R receives m_2^*, O has not been contacted by Q. Consider an execution F that is the same as E for the prefix that ends at the event of m_2^* arriving at its destination (inclusive); in F, after R receives m_2^*, O invokes Stop, and Stop returns before any other party invokes Stop. In F, O is honest. By Lemma 12, O outputs $\perp$. However, an honest party R outputs z, which violates the *completeness* property. A contradiction. □

Corollary 2. *For any optimistic execution E, let m_1^* be defined as in Corollary 1 and let m_2^* be defined as in Lemma 13; then the event of m_1^* arriving at its destination precedes the event of m_2^* arriving at its destination.*

Proof. The correctness follows from Lemma 13, the class of function f considered and $n \geq 2$. □

Corollary 3. *For any optimistic execution E, let m_2^* be defined as in Lemma 13 and let R be the destination of m_2^*; then in E, before R receives m_2^*, $\forall P \in \Omega \backslash \{R\}$, P has received at least one message.*

Proof. The correctness follows from Lemma 13 and $n \geq 2$. □

We have now defined the two messages: m_1^* and m_2^*. Here they are defined for any certain optimistic execution E. (If it is clear in the context, we omit the re-definition in the statements of the following lemmas.) Lemma 12 above shows the output of an honest party if it stops before the arrival of m_1^*. Below Lemma 14 shows the output of an honest party if it stops after the arrival of m_2^*.

Lemma 14. *For any optimistic execution E initialized with $x_1, x_2, \ldots, x_n$, let R be the destination of m_2^*; for any $P \in \Omega \backslash \{R\}$, let m be the last message received by P before message m_2^* arrives R in E. By Corollary 3, m exists.*

Let t be the status of P in E right after P receives m. Then for any execution $E(P)$ such that $E(P)$ is the same as E for P until P invokes Stop, and P invokes Stop with t after P receives m (and before P's next receipt of some message), Stop returns $z = f(x_1, x_2, \ldots, x_n)$ to P.

Proof. For any $E(P)$, P's behavior is the same as an honest P to the parties in $\Omega\backslash\{P\}$ and T, w.l.o.g., we say that in $E(P)$, P is honest.

Let $\mathscr{M}_P$ be the set of messages which P sends before m_2^* arrives at R in E. Then the event of P receiving m is the last event in E that might trigger P to send some message in $\mathscr{M}_P$. Due to the arbitrary delay of communication channels and the arbitrary time instant of invoking Stop, there exists such an execution $E(P)$ that P has sent all the messages in $\mathscr{M}_P$ before P's next receipt of some message and before P invokes Stop with t. For any such execution $E(P)$, the parties in $\Omega\backslash\{P\}$ may continue E without noticing P's invocation of Stop up to the point when m_2^* arrives at its destination R, and then an honest party R outputs z. Therefore, Stop should return z to P, for otherwise, as all parties are honest here, the return of $\perp$ violates the *completeness* property.

Now due to the arbitrary time instant of invoking Stop, it is indistinguishable for T whether P, invoking Stop with t, has sent all the messages in $\mathscr{M}_P$ or not. Therefore, for any $E(P)$, Stop has to return z to P. □

Following our proof overview, after the properties of m_1^* and m_2^*, what an honest party should output if it stops between m_1^* and m_2^* is shown in Lemma 15. In Lemma 15, we assume a subsequence of messages in an optimistic execution; roughly speaking, we assume that the honest party stops after this subsequence and investigate its output. We later combine Lemma 15 and the properties of m_1^* and m_2^* into Lemma 16, which relates the sequence of messages ordered by when they are received in an optimistic execution to the permutation sequence.

Lemma 15. *For any optimistic execution E and any* $k, 2 \le k \le n$*, w.l.o.g., let* $m_1, m_2, \ldots, m_k$ *be k messages in E such that (a) the destination of* $m_i, 1 \le i \le k$ *is* P_i*; (b)* $m_{i+1}, 1 \le i \le k-1$ *is the first message received by* P_{i+1} *after* P_i *receives* m_i *in E. Let* $t_i, 1 \le i \le k$ *be the status of* P_i *in E right before* P_i *receives* m_i*.*

For $1 \le i \le k$*, define execution* $E(P_i)$ *such that* $E(P_i)$ *is the same as E for* P_i *until* P_i *invokes Stop; in* $E(P_i)$*,* P_i *invokes Stop with* t_i *right before message* m_i *arrives at* P_i*.*

Assume that for any $E(P_1)$*, if no other party invokes Stop before* P_1*, then Stop returns* $\perp$ *to* P_1*. Then*

- *for* $k = 1$*, for any* $E(P_k)$*, when* P_k *invokes Stop, if no other party has invoked Stop, then Stop returns* $\perp$ *to* P_k*.*
- *for* $k = 2$*, for any* $E(P_k)$*, when* P_k *invokes Stop, if* P_{k-1} *has invoked Stop with* t_{k-1}*, Stop has returned* $\perp$ *to* P_{k-1} *and no other party has invoked Stop, then Stop returns* $\perp$ *to* P_k *except with negligible probability.*
- *for* $3 \le k \le n$*, for any* $E(P_k)$*, when* P_k *invokes Stop if* $P_1, P_2, \ldots, P_{k-1}$ *have invoked Stop with* $t_1, t_2, \ldots, t_{k-1}$ *respectively and for* $2 \le i \le k-1$*,* P_i *invokes Stop after Stop returns to*

P_{i-1}, and Stop has returned $\perp$ to $P_1, \ldots, P_{k-1}$, and no other party has invoked Stop, then Stop returns $\perp$ to P_k except with negligible probability.

Proof. Let E be initialized with $x_1, x_2, \ldots, x_n$. We prove the lemma by induction. The base case, for which $k = 1$, is trivial.

Suppose the statement is true for $k-1, 2 \leq k \leq n$. Assume any $E(P_k)$ as an execution such that when P_k invokes Stop, $P_1, \ldots, P_{k-1}$ have invoked Stop with $t_1, \ldots, t_{k-1}$ respectively according to the statement, Stop has returned $P_1, \ldots, P_{k-1}$ $\perp$ and no other party has invoked Stop, and Stop returns r to P_k, where r is a random variable. (The randomness comes from that of $P_1, \ldots, P_k$ and T.) Figure 4.2a illustrates $E(P_k)$.

For any $E(P_k)$, let $E^*(P_k)$ be an execution that is the same as $E(P_k)$ for $P_1, P_2, \ldots, P_n$ until P_k invokes Stop right before message m_{k-1} arrives at P_{k-1}. If $P_j, \ldots, P_{k-1}$ for some $j, 1 \leq j \leq k-1$ do not invoke Stop before message m_{k-1} arrives at P_{k-1} in $E(P_k)$, let $P_j, \ldots, P_{k-1}$ invoke Stop right before message m_{k-1} arrives at P_{k-1} in the same order with the same status as in $E^*(P_k)$. Also, let P_k invoke Stop after Stop has returned P_{k-1} $\perp$.

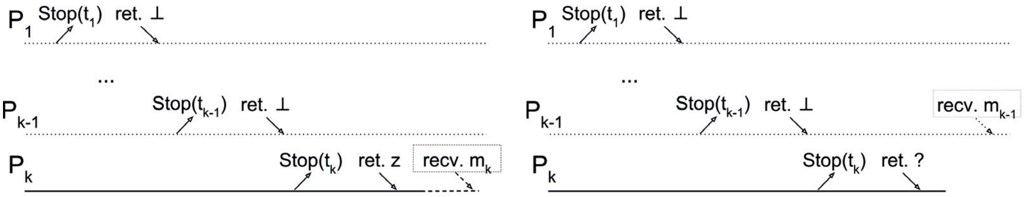
Due to the arbitrary delay of communication channels, in both $E(P_k)$ and $E^*(P_k)$, P_k's behavior is the same as an honest P_k to $\Omega \backslash P_k$ and T. Hereafter we say that P_k is honest. Again due to the arbitrary delay of communication channels, to P_k and T, any $E^*(P_k)$ is indistinguishable from any $E(P_k)$ at the point when P_k invokes Stop. Furthermore, since m_k is the first message received by P_k after P_{k-1} receives m_{k-1} in E, the status of P_k in $E^*(P_k)$ is also t_k. Thus in any $E^*(P_k)$, Stop returns r to P_k (where the distribution of r remains the same). Figure 4.2b illustrates $E(P_k)$.

For any $E(P_{k-1})$ and for any $E^*(P_k)$, we define an execution F such that (a) F is the same as $E(P_{k-1})$ for P_{k-1} until P_{k-1} invokes Stop with t_{k-1} right before m_{k-1} arrives at P_{k-1}; (b) F is the same as $E^*(P_k)$ for P_k until P_k invokes Stop with t_k right before m_{k-1} arrives at P_{k-1}; (c) when P_k invokes Stop, $P_1, \ldots, P_{k-1}$ have invoked Stop with $t_1, \ldots, t_{k-1}$ respectively and $P_i, 2 \leq i \leq k-1$ invokes Stop after Stop returns to P_{i-1}, Stop has returned $\perp$ to $P_1, \ldots, P_{k-2}$ and no other party has invoked Stop. Figure 4.2c illustrates our construction F.

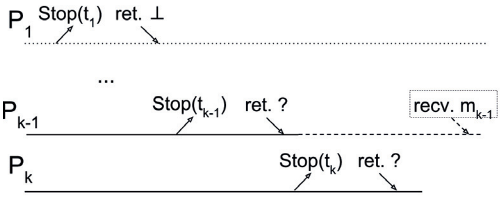
In F, P_{k-1}'s behavior is the same as an honest P_{k-1} to $\Omega \backslash \{P_{k-1}\}$ and T. Hereafter, we say that P_{k-1} is honest in F. According to n, there are two possibilities. First, if $n = 2$, then $k = 2$ and all parties are honest. Since the statement is true for $k-1$, Stop returns $\perp$ to P_{k-1} in F. Then by the *completeness* property, $r = \perp$ with probability 1. Second, if $n > 2$, since the statement is true for $k-1$, then Stop returns $\perp$ to P_{k-1} except with negligible probability. When Stop returns $\perp$ to P_{k-1}, $E^*(P_k)$ and F are indistinguishable to T and P_k due to the arbitrary delay of communication channels. As a result, Stop returns r to P_k (where the distribution of r remains the same).

Then for the second possibility, by the *(weak) fairness* property, $r = \perp$ except with negligible probability. We can show this by contradiction. Suppose that $r \neq \perp$ with non-negligible probability. We build an algorithm $\mathscr{A}$ such that (1) $\mathscr{A}$ controls all parties except for P_{k-1} and P_k, and (2) $\mathscr{A}$ plays the asynchronous network and the roles of the malicious parties so that the resulting execution among $P_1, P_2, \dots, P_n, T$ is F. $\mathscr{A}$ is a computationally bounded algorithm such that two honest parties P_{k-1} and P_k output differently with non-negligible probability. This violates the *(weak) fairness* property. A contradiction.

As a result, if the statement is true for $k-1, 2 \le k \le n$, the statement is true for k. Therefore, the lemma is true for any $k, 2 \le k \le n$. □



(a) Execution $E(P_k)$ (b) Execution $E^*(P_k)$



(c) Execution F

Figure 4.2 – The three key executions in the proof of Lemma 15. A dot line means that any event might occur. A dashed line means that an event does not occur. A solid line means that the same event as in E occurs.

Lemma 16. *For any optimistic execution E, let $\underline{R}$, a sequence of Ω, be the sequence of destinations of the messages ordered by when they are received between the two events: the event of m_1^* arriving at its destination and the event of m_2^* arriving at its destination, inclusive.*

Then $\underline{R}$ contains all the permutations of Ω as subsequences.

Proof. Let E be initialized with $x_1, x_2, \dots, x_n$. We prove by contradiction. Suppose that, w.l.o.g., $\underline{R}$ does not include $P_1, P_2, \dots, P_n$ as a subsequence.

By Corollary 2, $\underline{R}$ starts at the destination of m_1^* and ends at the destination of m_2^*; and $\underline{R}$ includes P_1 as a subsequence, which is also true for $P_2, \dots, P_n$. Then there exists some $k, 2 \le k \le n-1$ such that $\underline{R}$ includes $P_1, P_2, \dots, P_k$ as a subsequence and does not include $P_1, P_2, \dots, P_{k+1}$ as a subsequence.

As a result, there exists a sequence $m_1, m_2, \dots, m_k$ of k messages in E such that (a) the destination of $m_i, 1 \le i \le k$ is P_i; (b) $m_{i+1}, 1 \le i \le k-1$ is the first message received by P_{i+1} after P_i receives m_i and (c) $m_1 = m_1^*$, or m_1 is the first message received by P_1 after m_1^* arrives at its destination; and (d) the event of m_k arriving at P_k precedes the event of m_2^* arriving at its destination. (The event of m_k may also be the event of m_2^*.)

Let t_1 be the status of P_1 right before P_1 receives m_1 in E. Define execution $E(P_1)$ such that $E(P_1)$ is the same as E for P_1 until P_1 invokes Stop with t_1 right before m_1 arrives at P_1. By Lemma 12, for any $E(P_1)$, if no other party invokes Stop before P_1, then Stop returns $\perp$ to P_1.

Let $t_i, 2 \le i \le k$ be the status of P_i right before P_i receives m_i in E. Define execution $E(P_k)$ such that (a) $E(P_k)$ is the same as E for P_k until P_k invokes Stop with t_k right before message m_k arrives at P_k; (b) $P_1, P_2, \ldots, P_{k-1}$ invoke Stop with $t_1, t_2, \ldots, t_{k-1}$ respectively; (c) for $2 \le i \le k$, P_i invokes Stop after Stop returns to P_{i-1}; (d) Stop returns $\perp$ to $P_1, P_2, \ldots, P_{k-1}$; and (5) no other party has invoked Stop. By Lemma 15, Stop returns $\perp$ to P_k in $E(P_k)$ except with negligible probability. Figure 4.3a illustrates $E(P_k)$.

Let m be the last message received by P_{k+1} before message m_2^* arrives at its destination in E (inclusive). By Corollary 3, m exists if P_{k+1} is not the destination of m_2^*. Therefore, if P_{k+1} is not the destination of m_2^*, then the event of m arriving at its destination precedes the event of m_k arriving at P_k in E (for otherwise, we have a subsequence $P_1, P_2, \ldots, P_{k+1}$, which gives a contradiction). Moreover, P_{k+1} can not be the destination of m_2^* (for otherwise, we again have a subsequence $P_1, P_2, \ldots, P_{k+1}$, which gives a contradiction).

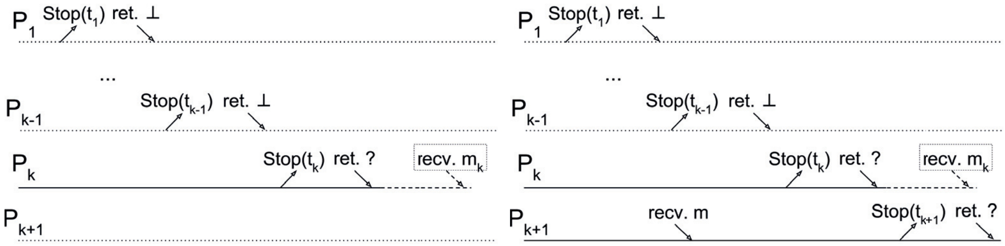
Let t_{k+1} be the status of P_{k+1} right after P_{k+1} receives m in E. Consider an execution $E(P_k, P_{k+1})$ that is the same as $E(P_k)$ for all the parties in $\Omega \backslash \{P_{k+1}\}$ and is the same as E for P_{k+1} until P_{k+1} invokes Stop with t_{k+1}, which is after Stop has returned to P_k. Since the event of m arriving at its destination precedes the event of message m_k arriving at P_k, in $E(P_k, P_{k+1})$, we let P_{k+1} invoke Stop with t_{k+1} also after P_{k+1} receives m. Figure 4.3b illustrates our construction $E(P_k, P_{k+1})$.

In $E(P_k, P_{k+1})$, P_k's behavior is the same as an honest P_k to $\Omega \backslash \{P_k\}$ and T; P_{k+1}'s behavior is the same as an honest P_{k+1} to $\Omega \backslash \{P_{k+1}\}$ and T. Hereafter, we say that P_k and P_{k+1} are honest in $E(P_k, P_{k+1})$. Moreover, until Stop returns to P_k, $E(P_k, P_{k+1})$ and $E(P_k)$ are indistinguishable to P_k and T and therefore Stop returns $\perp$ to P_k except with negligible probability also in $E(P_k, P_{k+1})$. However, by Lemma 14, Stop returns $z = f(x_1, x_2, \ldots, x_n)$ to P_{k+1}.

Now we build an algorithm $\mathscr{A}$ such that (1) $\mathscr{A}$ controls all parties except for P_{k-1} and P_k, and (2) $\mathscr{A}$ plays the asynchronous network and the roles of the malicious parties such that every execution among $P_1, P_2, \ldots, P_n$ satisfies $E(P_k.P_{k+1})$. $\mathscr{A}$ is a computationally bounded algorithm such that two honest parties P_k and P_{k+1} output differently with non-negligible probability. This violates the *(weak) fairness* property. A contradiction. $\square$

Now that we have all the necessary properties of any optimistic execution, we are ready to prove Theorem 9.

Proof of Theorem 9. Let $\underline{R}$ be defined as in Lemma 16. Recall that ℓ is the length of the shortest



(a) Execution $E(P_k)$ (b) Execution $E(P_k, P_{k+1})$

Figure 4.3 – Two key executions in the proof of Lemma 16. A dot line means that any event might occur. A dashed line means that an event does not occur. A solid line means that the same event as in E occurs.

sequence which contains, as subsequences, all permutations of n different symbols. Then by Lemma 16, ℓ lower bounds the length of $\underline{R}$. By the definition of m_1^*, there are at least $n-2$ messages that precede m_1^* in E; otherwise, at least one party has not yet contacted the destination of m_1^*. By the definition of m_2^*, there are at least $n-1$ messages that follow m_2^* in E; otherwise, at least one party P cannot compute z from P's input and P's status.

Therefore, during any optimistic execution E, the number of messages sent is at least $\ell + 2n - 3$. □

Remark 1 (Honest behavior in an execution)**.** Usually without a protocol specification, we cannot define any honest behavior. In the proof of Theorem 9, the honest behavior is relative to an optimistic execution.

4.4 An Optimal Protocol

To prove that $\ell + 2n - 3$ is a tight lower bound, we describe in this section an $(\ell + 2n - 3)$-message optimistic fair computation scheme for the function that implements fair exchange of certain items. This shows that the optimal message complexity can be achieved for some optimistic fair computation scheme.

Our optimal protocol relies on a publicly verifiable *transcript*. I.e., each destination (i.e., each party that receives a certain message) can verify whether the previous messages have arrived at their destinations correctly. This is realized by adding digital signatures [139, 145]. In order to help T recover the n inputs (if necessary) when some party invokes Stop, the n parties exchange *verifiable encryption* [133] of the n inputs in the protocol that computes without the third party. Section 4.1 recalls the basics of digital signatures and verifiable encryption, before describing our optimal protocol.

4.4.1 Preliminaries

We denote a digital signature on message m by $\sigma = Sig_{sk}(m)$, and the verification algorithm of a digital signature by $Ver_{pk}(\sigma, m)$, where pk is a public key and sk is the corresponding secret key. Sometimes we denote the signature of a party $P_i, i \in \{1, 2, \ldots, n\}$ simply by $Sig_i(m)$.

Recall that s is the security parameter of the fair computation scheme. Then roughly speaking, a digital signature scheme is secure if any adversary (of running time polynomial in the security parameter s) is able to forge a valid signature on some new message even after seeing many valid signatures on other messages (chosen by the adversary and of a number polynomial in s), only with negligible probability. (See [139, 145] for a formal definition of digital signature schemes and their security.)

A verifiable encryption scheme is a recovery algorithm D and a two-party protocol between prover P and verifier V [133]. Their common inputs are public key vk, public value x, condition κ[10] and binary relation R. Prover P takes witness w as an extra input. Verifier V rejects and outputs $\perp$ if $(x, w) \notin R$; otherwise V not only accepts but also obtains string α such that $D(sk, \kappa, \alpha) = w$ and $(x, w) \in R$.

We denote an instance of verifiable encryption by $VE(vk, \kappa, w, x, R)$. Roughly speaking, a verifiable encryption scheme is secure, if no malicious verifier is able to learn w without sk and no malicious prover is able to make V accept $\hat{\alpha}$ such that $(D(sk, \kappa, \hat{\alpha}), \hat{w}) \notin R$. The formal definition and definition of security for verifiable encryption schemes are recalled later when we prove the correctness of the optimal protocol. A prominent example of verifiable encryption is Asokan et al.'s non-interactive construction of verifiable encryption so that in the two-party protocol between P and V, only P sends a message to V and this message is considered as the string α if V accepts the message. Asokan et al.'s non-interactive construction of verifiable encryption can be used to verifiably encrypt a list of digital signature schemes, which includes Schnorr signatures, DSS signatures, Fiat-Shamir signatures, Ong-Schnorr signatures and GQ signatures [48].

4.4.2 Protocol description

In this section, we now present an $(\ell + 2n - 3)$-message optimistic fair computation scheme for function f (Equation 4.3) and thereby, prove that the lower bound of $\ell + 2n - 3$ messages is tight (Theorem 10). In other words, we show the tightness in a constructive way.

Theorem 10. *There exists an optimistic fair computation scheme for some function f where n honest parties can evaluate f after they exchange exactly $\ell + 2n - 3$ messages without resorting to T (i.e., in every optimistic execution).*

---

[10] Condition κ usually represents the instance ID of the protocol, the public value and the binary relation to be verified. In our fair computation scheme later, the resulting string of the two-party protocol between P and V can only be decrypted by a trusted party. The trusted party decrypts the string only if the following condition holds: the decrypted witness satisifies the binary relation with the public value.

Algorithm 11 Compute π

```
Require: a sequence $\underline{i}$ of length $l$ that contains all the permutations of $\{1,2,\dots,n\}$
Ensure: $(l+2n-3)$-message Compute $\pi$
Build sequence $\underline{j}$:
    $j_1, j_2, \dots, j_{n-2}, \underline{i}, j_{n+l-1}, j_{n+l}, \dots, j_{l+2n-3}$
    where (a) $j_1, j_2, \dots, j_{n-2}, i_1$ are $n-1$ different symbols; and (b) $i_l, j_{n+l-1}, j_{n+l}, \dots, j_{l+2n-3}$ are $n$ different symbols.
Set $j_0 = \{1,2,\dots,n\} \backslash \{i_1, j_1, j_2, \dots, j_{n-2}\}$
In $\pi$, $P_{j_{k-1}}$ sends a message $m_{k-1}$ to $P_{j_k}$ upon receiving $m_{k-2}$ for $k = 1,2,\dots,l+2n-3$ (except $P_{j_0}$ who sends $m_0 = VE_{j_0}$ upon initialization) where
    $m_{k-1} = \begin{cases} m_{k-2}||VE_{j_{k-1}}||Sig_{j_{k-1}}(m_{k-2}||VE_{j_{k-1}}) & 2 \le k \le n \\ m_{k-2}||Sig_{j_{k-1}}(m_{k-2}) & n+1 \le k \le end(j_{k-1}) \\ m_{k-2}||x_{j_{k-1}}||Sig_{j_{k-1}}(m_{k-2}||x_{j_{k-1}}) & end(j_{k-1})+1 \le k \le l+n-2 \\ (x_1, x_2, \dots, x_n) & l+n-1 \le k \le l+2n-3 \end{cases} \quad (4.2)$
    and
    $VE_{j_{k-1}} = VE(vk_T, \kappa, x_{j_{k-1}}, a_{j_{k-1}}, R_{j_{k-1}});$
    $\kappa = (a_1, R_1), (a_2, R_2), \dots, (a_n, R_n)$, which identifies the intended $x_1, x_2, \dots, x_n$;
    $end(j_{k-1}) = \max_{K \in \{1,2,\dots,l\}} \{K | i_K = j_{k-1}\} + n - 2$
$P_1, P_2, \dots, P_n$ outputs $z = (x_1, x_2, \dots, x_n)$
```

We build our protocol with Compute π (Algorithm 11) and Stop μ (Algorithm 12) given *any* sequence that contains all the permutations of $\{1,2,\dots,n\}$. Let l be the length of the sequence. We then show in Theorem 11 that our protocol is an $(l+2n-3)$-message optimistic fair computation scheme for the following function:

$$f(x_1, x_2, \dots, x_n) = \begin{cases} (x_1, x_2, \dots, x_n) & (a_i, x_i) \in R_i \text{ for } i = 1,2,\dots,n \\ \perp & \text{otherwise} \end{cases} \tag{4.3}$$

where $R_1, R_2, \dots, R_n$ are n relations that allow the non-interactive construction of verifiable encryption and $a_1, a_2, \dots, a_n$ are n public values.[11] $R_1, R_2, \dots, R_n, a_1, a_2, \dots, a_n$ are included in the public description of f.

The one-time setup of the protocol is not included in Algorithm 11 and Algorithm 12. Before π and μ are carried out, a one-time setup (a) distributes necessary keys: T's public key vk_T

---

[11] We also assume that for $i \in \{1,2,\dots,n\}$, given a_i, any computationally bounded algorithm outputs x_i with negligible probability, and given (a_i, x_i) such that $(a_i, x_i) \in R_i$, any computationally bounded algorithm outputs $y_i, y_i \neq x_i$ such that $(a_i, y_i) \in R_i$ with negligible probability.

Algorithm 12 Stop μ

```
Require: sequence $\underline{j}$ of length $l+2n-3$ built for $\pi$
Ensure: Stop $\mu$ that accompanies $\pi$
For any $k \in \{0,1\dots,l+2n-3\}$, $P_{j_k}$ invokes $\mu$ when $P_{j_k}$ wants to stop in $\pi$; otherwise, if $\pi$ has not started, the $n$ parties output $\perp$, or if $\pi$ has finished, the $n$ parties output $(x_1,x_2,\dots,x_n)$.
For $k=0$, when invoking $\mu$, if $P_{j_k}$ has not sent $m_k$, $P_{j_k}$ quietly leaves $\pi$ and $\mu$ and outputs $\perp$.
For $1 \le k \le n-1$, when invoking $\mu$, if $P_{j_k}$ has not received $m_{k-1}$ correctly, $P_{j_k}$ quietly leaves $\pi$ and $\mu$ and outputs $\perp$.
For $n \le k \le l+2n-3$, let $I_k = \{index \mid j_{index} = j_k, index \in \{1,2,\dots,k-1\}\}$, let $last_k = \max I_k$ when $I_k \ne \emptyset$ and let $last_k = 0$ when $I_k = \emptyset$, and define $m_{-1}$ as an empty string. Then, for $n \le k \le l+2n-3$, when invoking $\mu$, if $P_{j_k}$ has not received $m_{k-1}$ correctly and has received $m_{last_k-1}$, then $P_{j_k}$ sends to $T$ message $req_k = m_{last_k}$. By sending $req_k$, $P_{j_k}$ claims that $P_{j_k}$ does not receive $m_{k-1}$.
$T$ verifies that $req_k$ is consistent with $P_{j_k}$'s claim; and $T$ calculates response
    $$resp = \begin{cases} \text{“aborted”} & \text{if } req_k \text{ and } P_{j_k}\text{'s claim are not consistent} \\ & \text{or } P_{j_k} \text{ has sent a request before} \\ z = (x_1,x_2,\dots,x_n) & \text{else if variable } z \text{ (which is initialized to } \perp\text{) is not } \perp \\ \text{“aborted”} & \text{else if } req_k \text{ does not contain } VE_1, VE_2, \dots, VE_n \\ z \leftarrow (x_1,x_2,\dots,x_n) & \text{else if } k > \min_{index \in \{progress+1,\dots,l+2n-3\}}\{index \mid j_{index} = j_k\} \\ & \text{and } x_i \leftarrow D(sk_T, \kappa, VE_i) \text{ for } i = 1,2,\dots,n \\ z \leftarrow (x_1,x_2,\dots,x_n) & \text{else if } k \ge l+n-1 \\ & \text{and } x_i \leftarrow D(sk_T, \kappa, VE_i) \text{ for } i = 1,2,\dots,n \\ \text{“aborted”} & \text{otherwise} \end{cases}$$
    $T$ updates $progress$ (which is initialized to 0) to $k$ if $k > progress$, $req_k$ and $P_{j_k}$'s claim are consistent and $P_{j_k}$ has not sent a request before. $T$ then sends $resp$ to $P_{j_k}$.
$P_{j_k}$ outputs $\perp$ if $resp =$“aborted”; and $P_{j_k}$ outputs $z$ if $resp = z$.
```

and secret key sk_T, n parties' public and secret keys correctly; (b) distributes the public description of f correctly; and (c) executes the one-time setup of the verifiable encryption. (If implemented, a trusted party Certificate Authority [146] can do this one-time setup.)

Some remarks on μ are in order: (a) as each part of the request message is publicly verifiable, T is able to efficiently verify whether a party P's request and P's claim are consistent by following Equation (4.2); and (b) P may invoke Stop at any point in time[12], e.g., when a message received by P in π is incorrect, or when P is impatient while waiting for some message; our protocol allows every party to define their own strategy of invoking Stop, independent of the other $n-1$ parties.

We prove that this protocol (consisting of π and μ), given a shortest permutation sequence, is

---

[12] If messages are delivered instantly, P does not invoke Stop.

an $(\ell + 2n - 3)$-message optimistic fair computation scheme (of which the proof is in Section 4.4.3). This implies Theorem 10. Combined with Theorem 9, $\ell + 2n - 3$ is thus a tight lower-bound on the number of messages for optimistic fair computation.

Theorem 11. *Given a sequence* $\underline{i}$ *of length* l *that contains all the permutations of* $\{1,2,\dots,n\}$*, the protocol consisting of* π *and* μ *is an* $(l + 2n - 3)$*-message optimistic fair computation scheme for function* f *in Equation (4.3) in an asynchronous network with* $n - 1$ *potentially malicious parties.*

In fact, function f implements fair exchange among n parties for items $x_1, x_2, \dots, x_n$ that satisfy relations $R_1, R_2, \dots, R_n$. Then Algorithm 11 and Algorithm 12 form a compiler that can transform a shortest permutation sequence into an $(\ell + 2n - 3)$-message optimistic fair exchange scheme. An application is a message-optimal optimistic fair exchange scheme of digital signatures [48].[13]

4.4.3 Correctness proof of our protocol

We give here a detailed proof of correctness for our optimistic fair computation scheme for function f, and thereby, prove Theorem 11. We note that this is a proof of a stand-alone execution. This is consistent with Definition 22, which considers a single execution of optimistic fair computation in isolation among $n + 1$ parties (including the trusted party T).

Before we present the proof, we recall the formal definition and security guarantee of verifiable encryption from [133].

Definition 25 (Verifiable encryption [133])**.** Let (G, E, D) be the key generation, encryption and decryption algorithms of a semantically secure public-key encryption scheme. Let (vk, sk) be one key pair generated by G where vk is the public key and sk is the secret key. Let R be a relation and let $L_R = \{x | \exists w \text{ such that } (x, w) \in R\}$. Then a *verifiable encryption* scheme for a relation R consists of a two-party protocol (P, V) and a recovery algorithm D. P and V take as common inputs: vk, x, R (and some condition κ to open string α). P takes witness w such that $(x, w) \in R$ as an extra input. V rejects (i.e., outputs $\perp$), or accepts and obtains string α. D takes as inputs: sk, α (and κ). D outputs a witness $\hat{w}$ (if the condition κ holds for $\hat{w}$).

A verifiable encryption scheme is *secure* if it satisfies the following properties:

- *Completeness*: If P and V are honest, then V accepts in the two-party protocol for all (vk, sk) and for all $x \in L_R$.

---

[13] In the application of fair exchange of digital signatures, R_i is some homomorphism θ depending on a given digital signature scheme [48], and each of the first n messages of π is appended with an image of θ such that the pre-image produces a correct signature. See [48] the non-interactive construction of verifiable encryption on digital signatures on the details of how to choose θ and produce a correct signature by a pre-image of θ. We remark here that the non-interactive construction of verifiable encryption on digital signatures in [48] uses part of the correct signature as the pre-image of θ or as the input to the function f (rather than use the signing key of the give digital signature scheme).

- *Validity*: For any computationally bounded algorithm $\hat{P}$, for all (vk, sk), if V accepts and obtains string α in the two-party protocol with $\hat{P}$, then given α and sk, D outputs a witness $\hat{w}$ such that $(x, \hat{w}) \notin R$ with negligible probability.

- *Computational zero-knowledge*: For every algorithm $\hat{V}$, there exists an expected polynomial-time simulator S given vk and x as well as R and κ, and with black-box access to $\hat{V}$ such that for all $x \in L_R$, the output of S is computationally indistinguishable from the output of $\hat{V}$ after the two-party protocol with an honest P (which is given vk, x, R, κ and some witness w such that $(x, w) \in R$).

Furthermore, for the simplicity of the proof, we consider the particular verifiable encryption scheme proposed in [133]. In particular, their construction of verifiable encryption includes a three-move protocol (between the prover P and verifier V), where the second move is V sending a random bit string. Hence, as [133] pointed out, this protocol can be made non-interactive via Fiat-Shamir heuristic [147]: P uses a hash function to generate the random bit string. Therefore the resulting non-interactive variant is one message sent by P considered as the string α, and secure in the random oracle model [148]. For the non-interactive variant, it is easy to see that the algorithm V in the scheme can be deterministic; i.e., given the one message sent by $\hat{P}$, either V rejects (with probability 1) or V accepts (with probability 1); and the recovery algorithm D in the scheme is also deterministic; i.e., given sk, κ and α, either D rejects (with probability 1) or D outputs a witness (with probability 1).

Proof of Theorem 11. As shown in Algorithm 11, the number of messages is equal to the length of sequence $\underline{j}$ which is $l + 2n - 3$. Thus the n parties exchange exactly $l + 2n - 3$ messages in π. In what follows, we verify that our protocol satisfies Definition 22 and Definition 23.

Optimism. If $P_1, P_2, \dots, P_n$ are honest and none invokes Stop, then all parties follow π in which all parties output $z = f(x_1, x_2, \dots, x_n)$ without interacting with T.

Non-triviality. As shown in Algorithm 11, if messages are delivered instantly, then $P_1, P_2, \dots, P_n$ do not invoke Stop; therefore, we find one execution of π that $P_1, P_2, \dots, P_n$ are honest and none invokes Stop.

Completeness. If $P_1, P_2, \dots, P_n$ are honest and none invokes Stop, then all parties follow π and output $z = f(x_1, x_2, \dots, x_n)$. Next, we show by contradiction that if all parties are honest and some invokes Stop, then either all parties output $\perp$ or all parties output $z = f(x_1, x_2, \dots, x_n)$. Suppose that an honest party P outputs $\perp$ and an honest party Q outputs z. Since P outputs $\perp$, then either (1) π has not started, or (2) $P = P_{j_k}$ and $0 \le k \le n - 1$, or (3) $P = P_{j_k}$ and

$n \leq k \leq l+2n-3$ For cases (1) and (2), since by Equation (4.2),

$$m_k = \begin{cases} m_{k-1}||VE_{j_k}||Sig_{j_k}(m_{k-1}||VE_{j_k}) & 1 \leq k \leq n-1 \\ VE_{j_0} & k=0, \end{cases}$$

P has not sent VE_{j_k}. Again by Equation (4.2), $m_{end(j_k)} = m_{end(j_k)-1}||x_{j_k}||Sig_{j_k}(m_{end(j_k)-1}||x_{j_k})$. Since $end(j_k) > n-1$, P has not sent x_{j_k}. Since all parties are honest, Q does not output z from running π or μ in cases (1) and (2).

In case (3), since all parties are honest, by the *completeness* property of verifiable encryption, and the definition of digital signatures, T accepts that req_k and $P = P_{j_k}$'s claim are consistent. As P is honest, P has not sent a request before. In case (3), we consider two disjoint cases: (a) $\exists i \in \{1,2,\ldots,n\}$, VE_i is not in req_k, and (b) $\forall i \in \{1,2,\ldots,n\}$, VE_i is in req_k.

Consider case (3.a). By Equation (4.2), $\forall index \geq n-1$, m_{index} contains $VE_1, VE_2, \ldots, VE_n$. Then $0 \leq last_k \leq n-2$. Since $j_0, j_1, \ldots, j_{n-1}$ are different from each other, $j_k \neq j_{n-1}$. Moreover, $k = \min_{index \in \{n,n+1,\ldots,l+2n-3\}}\{index | j_{index} = j_k\} \leq end(j_k)$. Therefore, P has not sent x_{j_k}, and Q cannot output x_{j_k} following π.

Clearly, if Q does not interact with T, then Q outputs $\perp$, and furthermore, if Q interacts with T before P interacts with T, then Q also outputs $\perp$. If Q interacts with T after P interacts with T, then we assume that Q sends a request req_q to T. Since Q is honest, T accepts that req_q is consistent with Q's claim that Q has not received m_{q-1} (but has received $m_{last_q - 1}$). By the definition of $\underline{i}$, $q \leq end\, j_q$. (Otherwise, we do not have j_k, j_q as a subsequence of $\underline{i}$). In addition, since Q is honest, $q \leq \min_{index \in \{k+1,k+2,\ldots,l+2n-3\}}\{index | j_{index} = j_q\}$. Therefore, T sends "aborted" to Q.

In case (3.b), w.l.o.g., assume that P is the earliest process that sends to T a request and receives "aborted". Then variable $progress$ is 0 at T when P sends req_k. Then we have

$$k \leq \min_{index \in \{1,2,\ldots,l+2n-3\}}\{index | j_{index} = j_k\} \triangleq first(j_k).$$

If $j_k \neq j_0$, $k \leq n-1$, which gives a contradiction. If $j_k = j_0$, then since $k \leq first(j_k)$, $last_k = 0$; thus $req_k = m_{last_k} = VE_{j_0}$, which also gives a contradiction for $n \geq 2$.

Termination. As shown in Algorithm 11 and Algorithm 12, an honest party either follows π and outputs, or wants to stop, follows μ and outputs. Since any message between an honest party and T eventually reaches its destination, an honest party eventually outputs.

Fairness. We prove that for any $e \in \mathbb{N}$, $1 \leq e \leq n-1$, any e malicious parties $P_{d_1}, P_{d_2}, \ldots, P_{d_e}$, and any computationally bounded algorithm $\mathscr{A}$, there exists a computationally bounded algorithm $\mathscr{S}$ such that the joint outputs $O_{P_1,P_2,\ldots,P_n,\mathscr{A}}(x_1, x_2, \ldots, x_n)$ and $O_{\bar{P}_1,\bar{P}_2,\ldots,\bar{P}_n,\mathscr{S}}(x_1, x_2, \ldots, x_n)$ are computationally indistinguishable for any $x_1, x_2, \ldots, x_n$.

We construct $\mathscr{S}$ that runs $\mathscr{A}$ as a black-box as follows.

1. $\mathscr{S}$ generates $n+1$ key pairs $(pk_1, sk_1), (pk_2, sk_2), \ldots, (pk_n, sk_n), (vk_T, sk_T)$; and then $\mathscr{S}$ invokes $\mathscr{A}$ and initializes $\mathscr{A}$ with inputs $x_{d_1}, x_{d_2}, \ldots, x_{d_e}$, $n+1$ parties' public keys $pk_1, pk_2, \ldots, pk_n, pk_T$ and malicious parties' private keys $sk_{d_1}, sk_{d_2}, \ldots, sk_{d_e}$.[14]

2. $\mathscr{S}$ plays the role of the $n-k$ honest parties $P_{h_1}, P_{h_2}, \ldots, P_{h_{n-e}}$ and T, and executes our protocol honestly with $\mathscr{A}$ except that:

    - If by Algorithm 11, $\mathscr{S}$ has to send the $(k-1)$th message for $1 \le k \le n$ on behalf of an honest party, then by the construction of Fiat-Shamir paradigm [147] and the *computational zero-knowledge* property of verifiable encryption, $\mathscr{S}$ can simulate the random oracle [148] and invoke the simulator (defined in the *computational zero-knowledge* property) to compute message $\hat{m}_{k-1}$ (that is computationally indistinguishable from the $(k-1)$th message except with negligible probability).

    - If by Algorithm 11, $\mathscr{S}$ has to send the $(k-1)$th message for $end(j_{k-1}) + 1 \le k \le l + n - 2$ on behalf of an honest party P_{src}, then $\mathscr{S}$ sends $\hat{x}_{d_1}, \hat{x}_{d_2}, \ldots, \hat{x}_{d_e}$ on behalf of $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$ respectively to U. $\mathscr{S}$ obtains a response from U, which contains $x_{h_1}, x_{h_2}, \ldots, x_{h_{n-e}}$. Then $\mathscr{S}$ uses x_{src} to compute message $\hat{m}_{k-1}$. (How to obtain $\hat{x}_{d_1}, \hat{x}_{d_2}, \ldots, \hat{x}_{d_e}$ is explained later.)

    - If by Algorithm 12, $\mathscr{S}$ has to send a response including the $P_{h_1}, P_{h_2}, \ldots, P_{h_{n-e}}$'s inputs on behalf of T, then $\mathscr{S}$ sends $\hat{x}_{d_1}, \hat{x}_{d_2}, \ldots, \hat{x}_{d_e}$ on behalf of $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$ respectively to U. $\mathscr{S}$ obtains a response from U, which contains $x_{h_1}, x_{h_2}, \ldots, x_{h_{n-e}}$. $\mathscr{S}$ uses $x_{h_1}, x_{h_2}, \ldots, x_{h_{n-e}}$ to compute a response. (How to obtain $\hat{x}_{d_1}, \hat{x}_{d_2}, \ldots, \hat{x}_{d_e}$ is explained later.)

    - ($\mathscr{S}$ sends $\hat{x}_{d_1}, \hat{x}_{d_2}, \ldots, \hat{x}_{d_e}$ on behalf of $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$ respectively only once to U.)[15]

3. In addition, $\mathscr{S}$ executes the following.

    - If according to an honest party P's strategy of invoking Stop and μ, at some point in the execution with $\mathscr{A}$, P invokes Stop and outputs $\perp$, then $\mathscr{S}$ sends $\perp$ on behalf of an arbitrary party in $\bar{P}_{d_1}, \bar{P}_{d_2}, \ldots, \bar{P}_{d_e}$ to U. If $\mathscr{S}$ ever sends $\perp$, $\mathscr{S}$ sends $\perp$ only once.

    - $\mathscr{S}$ saves $\hat{x}_{d_1}, \hat{x}_{d_2}, \ldots, \hat{x}_{d_e}$ by decrypting $VE_{d_1}, VE_{d_2}, \ldots, VE_{d_e}$ from the messages exchanged with $\mathscr{A}$ (in π or μ).

---

[14] Both $\mathscr{A}$ and $\mathscr{S}$ are also initialized with public information, including the relations $\kappa = (a_1, R_1)||(a_2, R_2)||\cdots||(a_n, R_n)$, the algorithms of our protocol and in particular, the deterministic strategy of when to invoke Stop for every honest party.

[15] We note that on behalf of T, when $\mathscr{S}$ has to verify the ciphertexts of verifiable encryption in a request, $\mathscr{S}$ only verifies those ciphertexts $VE_{d_1}, VE_{d_2}, \ldots, VE_{d_e}$ (as $\mathscr{S}$ creates the others).

4. Finally, $\mathscr{S}$ outputs whatever $\mathscr{A}$ outputs.

We verify that $\mathscr{S}$ has saved $\hat{x}_{d_1}, \hat{x}_{d_2}, \dots, \hat{x}_{d_e}$ before $\mathscr{S}$ has to send a message that contains at least one honest party's input. If by Algorithm 11, $\mathscr{S}$ has to send the $(k-1)$th message for $end(j_{k-1}) + 1 \le k \le l + n - 2$, then $\mathscr{S}$ has received and verified the $(k-2)$th message. By the definition of sequence $\underline{i}$, if $j_{k-1} = j_{n-1}$, then $end(j_{k-1}) \ge n+1$; if $j_{k-1} \ne j_{n-1}$, then the first symbol of $\underline{i}$ is not j_{k-1} and thus $end(j_{k-1}) \ge n$. In either case, $k \ge n+1$, and therefore the $(k-2)$th message includes $VE_{d_1}, VE_{d_2}, \dots, VE_{d_e}$. If by Algorithm 12, $\mathscr{S}$ has to send a response on behalf of T, then $\mathscr{S}$ has verified the corresponding request, which also includes verified $VE_{d_1}, VE_{d_2}, \dots, VE_{d_e}$. Thus, by the *validity* property of verifiable encryption, $\mathscr{S}$ successfully decrypts $\hat{x}_{d_1}, \hat{x}_{d_2}, \dots, \hat{x}_{d_e}$ such that $\{a_{d_i}, \hat{x}_{d_i}\} \in R_{d_i}, \forall i \in \{1, 2, \dots, e\}$ except negligible probability.

We also verify that $\mathscr{S}$ does not send $\perp$ and $\hat{x}_{d_1}, \hat{x}_{d_2}, \dots, \hat{x}_{d_e}$ to U in the same execution, except with negligible probability in a separate lemma (Lemma 17, which is given and proved later).

To show that the joint outputs $O_{P_1,P_2,\dots,P_n,\mathscr{A}}(x_1, x_2, \dots, x_n)$ and $O_{\bar{P}_1,\bar{P}_2,\dots,\bar{P}_n,\mathscr{S}}(x_1, x_2, \dots, x_n)$ are computationally indistinguishable, we first consider the transcript between $\mathscr{S}$ and $\mathscr{A}$, and the transcript among $P_1, P_2, \dots, P_n$ and $\mathscr{A}$. By the *computational zero-knowledge* property of verifiable encryption and the definition of $\mathscr{S}$, any computationally bounded algorithm $\mathscr{A}$ cannot distinguish the two transcripts except with negligible probability. Let F be any execution between $\mathscr{A}$ and $\mathscr{S}$ in the game above when $\mathscr{S}$ is well-defined[16]. W.l.o.g., in F, honest parties played by $\mathscr{S}$ output according to Algorithm 11. Denote by O_F the joint output of $P_1, P_2, \dots, P_n$ and $\mathscr{A}$ in F. Then O_F and $O_{P_1,P_2,\dots,P_n,\mathscr{A}}(x_1, x_2, \dots, x_n)$ are computationally indistinguishable.

We next consider the execution G among $\bar{P}_1, \bar{P}_2, \dots, \bar{P}_n, \mathscr{S}$ and U when $\mathscr{S}$ runs F. We compare the joint output O_F with the joint output O_G of $\bar{P}_1, \bar{P}_2, \dots, \bar{P}_n, \mathscr{S}$ in G as follows. $\mathscr{S}$'s output is the same as $\mathscr{A}$'s output. For any honest party $P_{h_i}, i \in \{1, 2, \dots, n-e\}$, we show that $\bar{P}_{h_i}$ outputs the same. There are three possibilities for P_{h_i}: P_{h_i} either (1) invokes Stop and outputs $\perp$, or (2) invokes Stop and outputs a non-$\perp$ value, or (3) does not invoke Stop but outputs a non-$\perp$ value. In case (1), $\mathscr{S}$ sends $\perp$ to U and thus in G, $\bar{P}_{h_i}$ also outputs $\perp$. In case (2), (a) if P_{h_i} interacts with T, then $\mathscr{S}$ uses U's response as T's response to P_{h_i}; (b) if not, then to P_{h_i}, π finishes and $\mathscr{S}$ must have obtained U's response to query inputs for honest parties including P_{h_i}. Thus whether P_{h_i} interacts with T or not, $\bar{P}_{h_i}$ also outputs the same. Case (3) is the same as case (2.b). Then O_F and O_G have the same distribution.

As a result, O_G and $O_{P_1,P_2,\dots,P_n,\mathscr{A}}(x_1, x_2, \dots, x_n)$ are computationally indistinguishable. Denote by $event$ the event that $\mathscr{S}$ is not well-defined. Since $event$ occur with negligible probability, O_G and $O_{\bar{P}_1,\bar{P}_2,\dots,\bar{P}_n,\mathscr{S}}(x_1, x_2, \dots, x_n)$ are computationally indistinguishable. Then $\mathscr{S}$ is a computationally bounded algorithm such that for any $x_1, x_2, \dots, x_n$ such that $O_{P_1,P_2,\dots,P_n,\mathscr{A}}(x_1,$

---

[16] With negligible probability, $\mathscr{S}$ is not well-defined. I.e., $\mathscr{S}$ cannot simulate the game above with $\mathscr{A}$, for example, when the simulator defined in the *computational zero-knowledge* property of verifiable encryption exceeds polynomial time, when $\mathscr{S}$ decrypts $\hat{x}_{d_i}$ such that $(a_{d_i}, \hat{x}_{d_i}) \notin R_{d_i}$ for some $i \in \{1, 2, \dots, e\}$, and when some honest party outputs $\perp$ but $\mathscr{S}$ still has to send a response that includes honest parties' inputs.

$x_2, \ldots, x_n)$ and $O_{\bar{P}_1, \bar{P}_2, \ldots, \bar{P}_n, \mathscr{S}}(x_1, x_2, \ldots, x_n)$ are computationally indistinguishable. □

Next, we give the necessary lemma, which we use to verify that $\mathscr{S}$ as defined in the proof of Theorem 11 does not send conflicting messages to U except with negligible probability. However, instead of discussing $\mathscr{S}$, we state the lemma in a more general but equivalent way.

Lemma 17 (Simulation of $\mathscr{S}$)**.** *By Algorithm 11 and Algorithm 12, for any $e \in \mathbb{N}, 1 \leq e \leq n-1$ and any e malicious parties $P_{d_1}, P_{d_2}, \ldots, P_{d_e}$, for any computationally bounded algorithm $\mathscr{A}$ that controls the e malicious parties, $\forall x_1, x_2, \ldots, x_n$, for any honest party P,*

- *either P outputs $\perp$ with negligible probability,*
- *or P outputs $\perp$ with non-negligible probability and given that an honest party P outputs $\perp$, any other party Q outputs the honest parties' inputs except with negligible probability.*

By Lemma 17, for $\mathscr{S}$, as defined in the proof of Theorem 11, when $\mathscr{S}$ has to send $\perp$ to U with non-negligible probability, the probability that $\mathscr{S}$ has to send non-$\perp$ inputs to U is negligible. Lemma 17 also implies the inverse: if a party outputs the honest parties' inputs with non-negligible probability, then an honest party P outputs $\perp$ with negligible probability. In other words, when $\mathscr{S}$ has to send non-$\perp$ inputs to U with non-negligible probability, the probability that $\mathscr{S}$ has to send $\perp$ to U is negligible.

Proof of Lemma 17. We need only to prove the case where P outputs $\perp$ with non-negligible probability.

Since P is honest, then either (1) π has not started, or (2) $P = P_{j_k}$ for $0 \leq k \leq n-1$, or (3) $P = P_{j_k}$ for $n \leq k \leq l + 2n - 3$. (Some intermediary results are already deduced for the *completeness* property in the proof of Theorem 11 and is thus not repeated here.)

In cases (1) and (2), P has not sent x_{j_k} or VE_{j_k} and thus by the property of (a_{j_k}, R_{j_k}), any computationally bounded algorithm outputs x with negligible probability. In case (3), since P is honest, then by the determinism of the verification algorithm V of verifiable encryption, T accepts that req_k is consistent with P's claim, and in addition, P has not sent a request before. When P interacts with T, at least one of the two holds: (a) $\exists i \in \{1, 2, \ldots, n\}$, VE_i is not in req_k, or (b) $\forall i \in \{1, 2, \ldots, n\}$, VE_i is in req_k.

In case (3.a), P has not sent x_{j_k}. If Q interacts with T before P interacts with T, then T sends "aborted" to Q. If Q interacts with T after P interacts with T, then we assume that Q sends a request req_q. We show that the following two events occur at the same time with negligible probability: event A is $q > \min_{index \in \{k+1, k+2, \ldots, l+2n-3\}} \{index | j_{index} = j_q\} \triangleq next_k(q)$ and event B is that Q passes the consistency verification of req_q at T. We show this by contradiction. Suppose that the two events occur at the same time with non-negligible probability. Since $q > next_k(q)$, then $last_q \geq next_k(q) > k$; therefore, req_q includes message

m_k which includes P's signature on message m_{k-1}. Then Q is a computationally algorithm which forges P's signature on m_{k-1} (which P has not signed before) with non-negligible probability, a contradiction to the unforgeability of digital signatures. Therefore, A and B occur at the same time with negligible probability. Let $\bar{A}$ be the complement of A and let $\bar{B}$ be the complement of B. Then $\bar{A} \cup \bar{B}$ occurs except with negligible probability. Since $next_k(q) \leq end(j_q) \leq l+n-2$, T sends "aborted" to Q except with negligible probability. If Q does not interact with T, then Q only obtains VE_{j_k} from π. Thus by the property of (a_{j_k}, R_{j_k}) and by the *computational zero-knowledge* property of verifiable encryption, any computationally bounded algorithm outputs x with negligible probability.

In case (3.b), since $k \leq l+n-2$, then by the definition of end, $k \leq end(j_k)$. By Equation (4.2), P has not sent x. Similar to case (3.b), If Q does not interact with T, then Q only obtains VE_{j_k} from π. Clearly, if Q interacts with T but T sends "aborted" to Q except with negligible probability, then by the property of (a_{j_k}, R_{j_k}) and the *computational zero-knowledge* property of verifiable encryption, the probability that Q outputs x_{j_k} is negligible.

We show by contradiction that Q interacts with T but T sends "aborted" to Q except with negligible probability. Suppose that Q interacts with T but T sends a non-"aborted" value to Q with non-negligible probability. Assume that Q sends a request req_q to T. Let pg be the value of the variable $progress$ at T when Q starts to interact with T. Let event A be $q > next_pg(q)$ and let event B be the event that Q passes the consistency verification of req_q at T. Then similar to case (3.a), $\bar{A} \cup \bar{B}$ occurs except with negligible probability. If $\bar{B}$ occurs, then T sends "aborted" to Q. Since $\bar{A} \cup \bar{B}$ occurs except with negligible probability, then $\bar{A} \cap B$ occurs with non-negligible probability. Clearly, if the recovery of the inputs from their ciphertexts of verifiable encryption is not successful, then the condition κ is not satisfied and T sends "aborted" to Q. However, by the *validity* property of verifiable encryption, given that B occurs, the unsuccessful recovery occurs with negligible probability. In what follows, we consider the case where $\bar{A} \cap B$ occurs and the recovery for Q is successful.

W.l.o.g., Q is the first process that receives a non-"aborted" value from T. Then since Q is the first process that receives a non-"aborted" value, by the *validity* property of verifiable encryption, $pg \geq l+n-2$ except with negligible probability.

When Q invokes μ with request req_q, the variable z at T is $\perp$. By Algorithm 12, thus T updates $progress$ in a specific way: $progress$ is first updated with request req_{I_1} where I_1 is the first index of j_{I_1} in the suffix $\underline{j}[n-1:]$ of sequence $\underline{j}$, and then each update is with such a request req_{I_2} that I_2 is the first index of j_{I_2} in the suffix $\underline{j}[progress+1:]$. Let α be the sequence of parties who invoke μ and trigger T to update $progress$ before P_{j_q} invokes μ for req_q. Let σ be the sequence of the subscripts of those parties.

Since $pg \geq l+n-2$ except with negligible probability, σ is a subsequence of $\underline{i} = \underline{j}[n-1 : l+n-2]$ and, moreover, must be the prefix of some permutation of $\{1, 2, \ldots, n\}$ in $\underline{i}$ except with negligible probability.

Clearly, if T returns a non-“aborted” to Q, then $q \geq l+n-1$. Since $\bar{A}$ occurs, $next_pg(q) \geq l+n-1$. When $pg \geq l+n-2$, since σ ends at j_{pg} (inclusive) and there is no j_q between j_{pg} and j_{next-1}, j_q must occur in σ. (Otherwise, as $next \geq l+n-1$, there is no hope for σ to include j_q in the permutation before j_{l+n-2} (inclusive), contradictory to the definition of $\underline{i}$.) In other words, P_{j_q} must have invoked μ before, except with non-negligible probability. Then T returns “aborted” to Q for req_q except with negligible probability. A contradiction.

Thus, we conclude that when P outputs $\perp$ with non-negligible probability, then given that an honest party P outputs $\perp$, any other party Q outputs the honest parties’ inputs except with negligible probability. $\square$

4.5 Related Work

4.5.1 Optimistic fair computation

Cachin and Camenisch [6] formalized optimistic fair computation for two parties and a third party T (that can also be malicious). Asokan et al. [48] formalized optimistic fair exchange of digital signatures between two parties and T (where T is honest). In this chapter, we assume T is honest. We briefly compare here the two definitions above. Cachin and Camenisch [6] formalized fair computation using the *simulatability paradigm* [5], while Asokan et al. [48] formalized fair exchange through games [139]. As the former can provide stronger security guarantee, we follow the definition of fair computation in [6]. Both formalizations consider the *termination* property in an asynchronous setting. We model this property using Stop, which is equivalent to the signal of termination in [48]. Asokan et al. [48] also defined the *completeness* property regarding the case where all parties are honest, while there is an ambiguity regarding this case in [6]. We adapt the definition of the *completeness* property from [48]. The *optimism* property was defined differently in [6] and [48]. In [6], the asynchronous network must deliver messages instantly, whereas in [48], the asynchronous network is allowed to deliver messages arbitrarily (while the rest of the statement is the same). We adopt the *optimism* property from [48], as it provides a stronger guarantee. Following Asokan et al.’s work [48], Küpçü and Lysyanskaya [149] defined *optimism* similarly in games.

In addition, we include the *non-triviality* property to rule out trivial protocols that send no message and abort all the time. (Our proof of the lower bound is based on the existence of at least one optimistic execution guaranteed by *non-triviality* and *optimism*, but our fair computation scheme, on the other hand, allows arbitrarily many optimistic executions.)

4.5.2 Optimistic fair exchange

For two parties, Asokan et al. [48] proposed a 4-message optimistic fair exchange scheme that ensures termination. Since $\ell = 3$ for two parties, our Theorem 9 shows that the 4-message fair exchange scheme is optimal for two parties. This also implies that a 3-message fair exchange scheme does not meet all of the required properties. For example, the optimistic fair exchange scheme proposed in [128] was criticized by Asokan et al. [48] as not ensuring *termination*. Another example is Ateniese's 3-message optimistic fair exchange scheme [150], which also does not ensure termination as noted by the author himself [150]. A recent follow-up work [151] has the same drawback.

To the best of our knowledge, up to our work (presented in this chapter), no message-optimal optimistic fair exchange or optimistic fair computation scheme among n parties for an arbitrary n (with $n-1$ potentially malicious parties) has been proposed.

4.5.3 Optimal optimistic schemes

We explain here the relation between the optimal efficiency of optimistic schemes of related problems and our optimal message efficiency. Pfitzmann, Schunter, and Waidner (PSW) [54] determined the optimal efficiency of fair two-party contract signing, Schunter [55] determined the optimal efficiency of fair two-party certified email, whereas Dashti [56] determined the optimal efficiency of two-party fair exchange in the crash-recovery model with no amnesia [152]. None of these results implies our Theorem 9, even only for $n = 2$. For PSW's result as well as Schunter's result, this is because there is no reduction of the problem of fair computation to the problem of fair contract signing[17] or fair certified email; for Dashti's result, this is because our model can be considered as the Byzantine failure model [152], and is thus stronger than the model considered by Dashti. Our proof of the lower bound, together with our message-optimal scheme, can be applied to prove that $\ell + 2n - 3$ is the optimal message efficiency of fair n-party contract signing in the model of PSW. The special case where $n = 2$ can be used to prove PSW's result, while PSW's proof was, unfortunately, flawed.

Draper-Gil et al. [153] determined the minimal message complexity of contract signing schemes with *weak fairness* on four topologies. Weak fairness implies that the honest parties might have different outputs as long as they can prove their honest behavior. On the contrary, our optimal message efficiency $\ell + 2n - 3$ applies to any topology, and employs a stronger fairness definition than [153]. Thus their result does not imply our Theorem 9 and vice versa.

4.5.4 The shortest permutation sequence

Mauw, Radomirović and Dashti (MRD) [51] proved that the optimal number of messages of *totally-ordered* fair contract signing schemes[18] falls between $\ell + n - 1$ and $\ell + 2n - 3$. Later, Mauw and Radomirović (MR) [53] generalized the result of MRD to *DAG-ordered* fair contract signing schemes[19]. Both [51] and [53] considered fair contract signing as fair exchange of

---

[17] The main difference is that contract signing outputs a proof which binds a contract agreed in advance while computation usually does not require such binding.

[18] In a *totally-ordered* contract signing scheme, signers execute totally-ordered communication steps; i.e., at any point in time, only one signer has sufficient messages to calculate and send the next message.

[19] In a *DAG-ordered* contract signing scheme, communication steps can be ordered in a directed acyclic graph.

digital signatures. They use a model different from PSW, and fall within the coverage of our Theorem 9. Neither MRD's result nor MR's result implies our Theorem 9. Neither allows arbitrarily interleaved messages as our Theorem 9; instead, they assume that communication steps are either totally ordered or ordered following a directed acyclic graph (DAG). In addition, both results [51, 53] propose a range of the optimal efficiency for fair exchange, instead of a concrete lower bound for fair computation in general (as does our Theorem 9).

It is important to note that our Theorem 9 is not a generalization of MRD's result nor of MR's result. What MRD or MR count are the messages sent from some signer. This makes the proof difficult to extend: after a message m leaves its source s, due to the asynchronous network, m does not help s's knowledge about other parties' possible states. Thus m should not help s reach an agreement if s wants to stop after sending m, unless the messages after m are defined and ordered in advance. On the contrary, what we count throughout our proof are the messages received (or not) at a destination d, which affects d's stop event. This is the key in our case for not requiring any ordering.

Another crucial concept used by MRD is the idea of an *idealized* protocol. An *idealized* protocol is informally defined as a totally-ordered fair exchange protocol of which the number of messages in an optimistic execution is optimal [51]. (Here a protocol is equivalent as a Compute protocol in our Definition 20. The communication with a third party T is not considered as part of the protocol.) At the *end* phase of the *idealized* protocol, each of the n signers is supposed to send exactly one message [51]. It is not clear yet whether the assumption can be justified or not: the main theorem in [51] relates the end of an *idealized* protocol with part of the shortest permutation sequence; however, (the form of the end of) the shortest permutation sequence is still open for a large n [129]. This also leads to a non-optimal fair exchange protocol in [51] and a non-optimal protocol compiler in [52] which generates a protocol specification of an optimistic fair contract signing scheme given a shortest permutation sequence.[20] Compared with MRD's *idealized* protocol, our proof of Theorem 9 shows that, at the end of an optimal protocol, each of the n parties may receive exactly one message, and moreover, the end of an optimal protocol is *not* related to the shortest permutation sequence. We believe that this has further implications on the design of correct and efficient fair computation protocols.

---

[20] Although [52] proved that the resulting protocol needs at least $\ell + 2n - 3$ messages in an optimistic execution, the number of messages exchanged during every optimistic execution is actually strictly larger than $\ell + 2n - 3$ for $n \geq 3$, and is thus not optimal.

5 Concluding Remarks

In this dissertation, we study the complexity and propose optimal protocols of decentralized solutions for reliable and secure distributed transactions. Here a decentralized solution refers to the one which does not use a distinguished coordinator or use the coordinator as little as possible. To this end, we perform two analyses on atomicity and causal consistency in reliable distributed transactions and one study on optimistic fair computation in secure distributed transactions. We now summarize our complexity results and outline a few open issues and research directions for future work.

5.1 Summary

5.1.1 Distributed transaction commit

We present the first systematic study of the complexity of atomic commit. We study the best-case complexity, i.e., the time and message complexity of any nice execution of a commit protocol. To have a better understanding of the tradeoff between atomicity and efficiency, we have a more fine-grained view of atomicity, compared with previous work [16, 1, 25, 26]. We consider two types of failures, crash and network failures and we study the complexity of a commit protocol by its robustness, i.e., which property (of the classical non-blocking atomic commit) is required in which executions (including less likely executions with failures). Our systematic study exhaustively goes through 27 variants of of non-blocking atomic commit (NBAC) defined by robustness.

Interestingly, our complexity results show that

- The time complexity and the message complexity reach the maximum (among the 27 variants) respectively when NBAC is solved in the face of crash failures and agreement is satisfied despite both types of failures;
- The message complexity increases (from zero to non-zero, and from $n-1+f$ messages to $2n-2$ messages for at most f crashes among n processes) when validity needs to be

additionally satisfied;

These complexity results also highlight a tradeoff between time and message complexity in 18 out of the 27 variants. By the complexity results, we answer the open question on the time and message complexity of synchronous NBAC (which solves NBAC only in the face of crash failures) since Dwork and Skeen's lower bound (on the number of messages) [1].

We propose the INBAC protocol which solves indulgent atomic commit, the most robust form among atomic commit problems we study. INBAC performs almost as efficiently as the widely-used two-phase commit (2PC) [22]: in some special case (for example, where among n processes, at most one can crash), INBAC induces two communication rounds, the same as 2PC, and needs additionally two messages, compared with 2PC. Previous protocols, PaxosCommit, and faster PaxosCommit [73], solve indulgent atomic commit as well. Our INBAC protocol is the most efficient among these protocols in that

- INBAC is delay-optimal: same as faster PaxosCommit and better than PaxosCommit;
- INBAC is message-optimal among the delay-optimal protocols.

The comparison between PaxosCommit and our INBAC protocol also illustrates a tradeoff between time and message complexity.

5.1.2 Causal transactions

We present the formal complexity analysis of causal transactions. We study the complexity of read-only transactions, considered the most frequent in practice, and obtain two impossibility results regarding fast read-only transactions:

- In an asynchronous system, if a causally consistent transactional storage system supports every transaction to read and write multiple objects, then even read-only transactions alone cannot be fast.
- In an asynchronous system where only servers have access to a global accurate clock (while client requests are oblivious to their local clocks), if a causally consistent transactional storage system supports fast read-only transactions and single-write transactions only, then read-only transactions cannot be invisible, where (in)visibility refers to the complexity that a read-only transaction incurs some write to servers (or not).

Our impossibilities apply to causal consistency and hence to stronger consistency criteria. They hold without assuming any message or node failures and hence hold for failure-prone systems. Our impossibility results hold only assuming that no server stores all objects, independent from any particular partial replication scheme.

To complement our second impossibility result, we propose a protocol that implements visible fast read-only transactions. Compared with COPS-SNOW, the previous protocol that provides fast read-only transactions [44], our protocol also provides fast single-object write transactions while COPS-SNOW does not. We show that under different system assumptions, the impossibility results can break, by proposing two protocols. The first protocol supports generic transactions (that breaks the first impossibility) in a synchronous system where there is a known upper bound on the time spent on the communication and local computation and a global accurate clock is accessible to all servers and clients. The second protocol provides invisible read-only transactions (that breaks the second impossibility) in an asynchronous system where a global accurate clock is accessible to all servers and clients. Both protocols are based on timestamps thanks to the accurate clock.

5.1.3 Optimistic secure transactions

We present, for the first time, a tight lower bound on the message complexity of optimistic secure transactions. We study optimistic secure transaction in the model of optimistic fair computation. Here fairness ensures a property similar to atomicity: either all participants may output the result of the transaction or none can, and also preserves privacy: no participant may know information of others' private inputs beyond the result of the transaction. We consider the worst adversarial setting: a maximum number ($n-1$ out of n) of malicious participants (or Byzantine failures), and study the message complexity of any optimistic execution.

Interestingly, our main result shows that in every optimistic execution, if we order all messages according to when they are received and construct a sequence of the destinations of all messages based on this order, then the sequence must contain all permutations of the n participants. This relates the message complexity in our study to the permutation sequence in combinatorics. Although the length ℓ of the shortest permutation sequence in combinatorics is still open for large n, by relating our problem to the shortest permutation sequence, we prove that $\ell+2n-3$ lower bounds the number of messages exchanged; we propose a matching scheme of fair exchange of exact $\ell+2n-3$ messages so that the lower bound is tight. This fair exchange scheme can be applied to exchange digital signature (such as Schnorr signatures [134], DSS signatures [135], Fiat-Shamir signatures [136], Ong-Schnorr signatures [137], GQ signatures [138]), and hence can implement message-optimal electronic contract signing.

Clearly, an application of the scheme is to trade items in a secure and transactional way. Compared with previous proposals of secure transactions that involve trusted third parties in every execution, the time complexity of the scheme is $\ell+2n-3$, which is $\theta(n)$ according to the current progress in combinatorics [58, 59, 60, 130], while previous proposals finish in constant time complexity. This highlights a tradeoff between the introduction of trust assumptions to a protocol and the complexity of the protocol.

5.2 Future Directions

5.2.1 Reliable transactions

Atomicity

The atomic commit protocol lies at the heart of distributed transaction processing systems [17, 64, 18, 19, 20, 21] where 2PC is widely used. Although the 2PC protocol is efficient, 2PC does not guarantee termination when processes can crash and 2PC can be blocked by slow messages caused by network failures where message delays can be unbounded (until some unknown stabilization time).

According to our systematic study, the 2PC protocol actually solves the following atomic commit problem: NBAC is solved in any failure-free execution, while only validity and agreement are satisfied despite crash and network failures. Then our INBAC protocol can be considered as an alternative to 2PC, as it solves indulgent atomic commit, which ensures termination despite crash and network failures (in addition to what 2PC solves), and performs almost as efficiently as 2PC. Hence it is intriguing to implement INBAC in those existing transaction processing systems which employ 2PC and to evaluate the performance in the failure-free settings and failure-prone settings. As we support the optimal nice execution by complex failure-prone executions, the challenge of the implementation and further optimization of the protocol would lie in the cases that abort transactions.

Our complexity results highlight a tradeoff between time and message complexity among 18 out of 27 variants of the atomic commit problem which we study. Thus it is also intriguing to have a systematic study of the tradeoff. Among these 18 variants, the tradeoff between time and message complexity for indulgent atomic commit is particularly interesting. In fact, some tradeoff result exists, following our work: Goren and Moses [154] characterized the tradeoff between time and message complexity for the atomic commit problem in the crash-failrue system. In addition, they measured time complexity by rounds and distinguished between a round where some process decides and a round where some process halts (i.e., quits the protocol). Distinguishing the deciding round and halting round may also contribute to future research in the investigation of the complexity of the atomic commit problem.

We also propose the 0NBAC protocol which solves the following atomic commit problem: NBAC is solved in any failure-free execution, while only agreement and termination are satisfied despite crash and network failures. The 0NBAC protocol with zero message and one message delay in any nice execution, is both message-optimal and delay-optimal. Thus it might be of practical interest to work on an application of 0NBAC and evaluate its performance in the failure-free settings as well as failure-prone settings.

Transaction consistency

Causal transactions are practically appealing, since (1) replication does not need to be performed while a transaction is executed, as in the model of eventual consistency, and (2) causal consistency allows more meaningful applications than eventual consistency. Hence the protocols which we propose to break the impossibilities of fast read-only transactions are of practical interest as they can potentially perform as efficiently as transactions in the model of eventual consistency. Possible future work includes the implementation of these protocols (which support fast read-only transactions), evaluate their performance and compare them with these protocols which ensure only eventual consistency to have a better understanding of the cost of fast read-only transactions. We are particularly interested in the protocol of visible fast read-only transactions which we propose. In our protocol, the inherent updates on servers (i.e., visibility) are performed outside any transaction. This might reduce the impact on the overall performance and enable it to outperform COPS-SNOW.

As fast read-only transactions are of practical interest, a more fine-grained study on the assumptions where the impossibilities hold or not could benefit future design of causally consistent storage systems. For example, in practice, clients and servers are given access to their local clocks between which there can be arbitrarily large drift. Assuming that client requests are non-oblivious to the local clocks, it is not yet clear whether the two impossibilities we obtain still hold or not especially in the partially replicated setting in general.

A formal study on the inherent cost of read-only transactions in general would also be interesting. To this end, a definition of visibility in general is necessary. The challenge to define the visibility for transactions of more than one round lies in the fact that a server may batch messages to increase throughput yet it is hard to isolate formally the message which brings visibility without imposing a particular framework on the underlying protocols of distributed storage.

5.2.2 Secure transactions

In electronic commerce, secure transactions preserve the privacy of data so that goods and services are not taken advantage of due to an unsuccessful transaction. Hence considering the current throughput of electronic transactions, it is worthwhile to investigate the time complexity of optimistic secure transactions. Our result which relates the pattern of messages in every optimistic execution to the permutation sequence may lay a basis on the investigation.

As the question of the shortest permutation sequence has been answered for small n [58], possible future work includes the implementation of our protocol for a small number of participants, evaluate the performance and compare it with the protocols which rely on trusted third parties in every execution. For performance evaluation, one might be particularly interested in the setting of parallel executions of the same protocol: by these protocols to compare with, the parallel executions all access the same trusted parties, which may be a performance bottleneck, while by our protocol, the parallel executions access different parties.

Another future direction is to perform an exhaustive study on the complexity of optimistic secure transactions on different types of failures and different numbers of possible failures like in our study of distributed transaction commit. In practice, among a large number of participants, an honest party may distrust a few rather than all of them. Then this future study can further highlight the tradeoff between the trust or confidence in the failure-prone setting, and the complexity of secure transactions.

Bibliography

[1] C. Dwork and D. Skeen, “The inherent cost of nonblocking commitment,” in *PODC ’83*, pp. 1–11.

[2] J. Gray, “The transaction concept: Virtues and limitations (invited paper),” in *VLDB ’81*, pp. 144–154.

[3] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, “Minimizing commit latency of transactions in geo-replicated data stores,” in *SIGMOD ’15*, pp. 1279–1294.

[4] T. A. S. Foundation, “Apache cassandra,” Online, 2016, http://cassandra.apache.org/.

[5] R. Canetti, “Security and composition of multiparty cryptographic protocols,” *Journal of Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.

[6] C. Cachin and J. Camenisch, “Optimistic fair secure computation,” in *CRYPTO ’00*, pp. 93–111.

[7] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, “California fault lines: Understanding the causes and impact of network failures,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 315–326, Aug. 2010.

[8] P. Gill, N. Jain, and N. Nagappan, “Understanding network failures in data centers: Measurement, analysis, and implications,” *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 350–361, Aug. 2011.

[9] B. Schroeder and G. A. Gibson, “Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you?” in *FAST ’07*.

[10] K. V. Vishwanath and N. Nagappan, “Characterizing cloud computing hardware reliability,” in *SoCC ’10*, pp. 193–204.

[11] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems,” in *The 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[12] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., 1992.

[13] ANSI, “Database language sql, ansi x3.135-1992 edition.” 1992.

[14] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” *SIGMOD Rec.*, vol. 24, no. 2, pp. 1–10, May 1995.

[15] M. T. Özsu and P. Valduriez, *Introduction to Transaction Management*. Springer New York, 2011, pp. 335–359.

[16] D. Skeen, “Nonblocking commit protocols,” in *SIGMOD ’81*, pp. 133–142.

[17] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, “Sinfonia: A new paradigm for building scalable distributed systems,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 159–174, 2007.

[18] D. Peng and F. Dabek, “Large-scale incremental processing using distributed transactions and notifications,” in *OSDI ’10*, pp. 1–15.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

[20] J. Du, S. Elnikety, and W. Zwaenepoel, “Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks,” in *SRDS ’13*, pp. 173–184.

[21] M. K. Aguilera, J. B. Leners, and M. Walfish, “Yesquel: Scalable sql storage for web applications,” in *SOSP ’15*, pp. 245–262.

[22] J. Gray, “Notes on data base operating systems,” in *Operating Systems, An Advanced Course*, 1978, pp. 393–481.

[23] D. Skeen and M. Stonebraker, “A formal model of crash recovery in a distributed system,” *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 219–228, 1983.

[24] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[25] V. Hadzilacos, “A knowledge-theoretic analysis of atomic commitment protocols,” in *PODS ’87*, pp. 129–134.

[26] O. Wolfson and A. Segall, “The communication complexity of atomic commitment and of gossiping,” *SIAM J. Comput.*, vol. 20, no. 3, pp. 423–450, 1991.

[27] “Snapshot isolation in sql server,” Online, 2017, https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[28] “Chapter 11. berkeley db transactional data store applications,” Online, 2010, https://docs.oracle.com/cd/E17275_01/html/programmer_reference/transapp_read.html.

[29] “Ssi,” Online, 2018, https://wiki.postgresql.org/wiki/SSI.

[30] M. J. Cahill, U. Röhm, and A. D. Fekete, “Serializable isolation for snapshot databases,” *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 20:1–20:42, Dec. 2009.

[31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, “Transactional storage for geo-replicated systems,” in *SOSP ’11*, 2011, pp. 385–400.

[32] J. Gray, P. Helland, P. O’Neil, and D. Shasha, “The dangers of replication and a solution,” *SIGMOD Rec.*, vol. 25, no. 2, pp. 173–182, Jun. 1996.

[33] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, Nov. 2013.

[34] Amazon, “Amazon dynamodb - nosql cloud database service,” Online, 2018, https://aws.amazon.com/dynamodb/.

[35] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops,” in *SOSP ’11*, pp. 401–416.

[36] ——, “Stronger semantics for low-latency geo-replicated storage,” in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 313–328.

[37] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, “Orbe: Scalable causal consistency using dependency matrices and physical clocks,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC ’13, pp. 11:1–11:14.

[38] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, “Gentlerain: Cheap and scalable causal consistency with physical clocks,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC ’14, pp. 4:1–4:13.

[39] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, “Write fast, read in the past: Causal consistency for client-side applications,” in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware ’15, 2015, pp. 75–87.

[40] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, “Cure: Strong semantics meets high availability and low latency,” in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 405–414.

[41] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, “I can’t believe it’s not causal! scalable causal consistency with no slowdown cascades,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 453–468.

[42] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[43] M. Raynal, G. Thia-Kime, and M. Ahamad, “From serializable to causal transactions for collaborative applications,” in *EUROMICRO ’97*, 1997, pp. 314–321.

[44] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, “The SNOW theorem and latency-optimal read-only transactions,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 135–150.

[45] M. VISA, “Secure electronic transaction specification, books 1-3,” June 1996.

[46] S. Micali and P. Rogaway, “Secure computation,” in *CRYPTO 1991*, J. Feigenbaum, Ed., pp. 392–404.

[47] D. H. Steves, C. Edmondson-Yurkanan, and M. Gouda, “Properties of secure transaction protocols,” *Computer Networks and ISDN Systems*, vol. 29, no. 15, pp. 1809 – 1821, 1997.

[48] N. Asokan, V. Shoup, and M. Waidner, “Optimistic fair exchange of digital signatures,” *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 4, pp. 593–610, 2000.

[49] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[50] R. Cleve, “Limits on the security of coin flips when half the processors are faulty,” in *STOC ’86*, pp. 364–369.

[51] S. Mauw, S. Radomirovic, and M. Dashti, “Minimal message complexity of asynchronous multi-party contract signing,” in *CSF ’09*.

[52] B. Kordy and S. Radomirovic, “Constructing optimistic multi-party contract signing protocols,” in *CSF 2012*.

[53] S. Mauw and S. Radomirovic, “Generalizing multi-party contract signing,” in *POST 2015*.

[54] B. Pfitzmann, M. Schunter, and M. Waidner, “Optimal efficiency of optimistic contract signing,” in *PODC ’98*, pp. 113–122.

[55] M. Schunter, “Optimistic fair exchange,” Ph.D. dissertation, Universität des Saarlandes, 2000, http://scidok.sulb.uni-saarland.de/volltexte/2004/233.

[56] M. T. Dashti, “Efficiency of optimistic fair exchange using trusted devices,” *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 1, pp. 3:1–3:18, May 2012.

[57] L. Lamport, “Lower bounds for asynchronous consensus,” *Distrib. Comput.*, vol. 19, no. 2, pp. 104–125, 2006.

[58] M. C. Newey, “Notes on a problem involving permutations as subsequences.” Stanford, CA, USA, Tech. Rep., 1973.

[59] E. Zălinescu, “Shorter strings containing all k-element permutations,” *Information Processing Letters*, vol. 111, no. 12, pp. 605 – 608, 2011.

[60] S. Radomirovic, “A construction of short sequences containing all permutations of a set as subsequences,” *The Electronic Journal of Combinatorics*, vol. 19, no. 4, 2012.

[61] R. Guerraoui and J. Wang, “How fast can a distributed transaction commit?” in *PODS ’17*, 2017, pp. 107–122.

[62] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The notions of consistency and predicate locks in a database system,” *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[63] C. Mohan, B. Lindsay, and R. Obermarck, “Transaction management in the r* distributed database management system,” *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, 1986.

[64] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “Pnuts: Yahoo!’s hosted data serving platform,” *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[65] V. Hadzilacos, “On the relationship between the atomic commitment and consensus problems,” in *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, 1990, pp. 201–208.

[66] R. Guerraoui, “Revisiting the relationship between non-blocking atomic commitment and consensus,” in *WDAG ’95*, pp. 87–100.

[67] B. Charron-Bost, “Agreement problems in fault-tolerant distributed systems,” in *SOFSEM ’01*, pp. 10–32.

[68] R. Guerraoui, “Non-blocking atomic commit in asynchronous distributed systems with failure detectors,” *Distrib. Comput.*, vol. 15, no. 1, pp. 17–25, 2002.

[69] R. Guerraoui, V. Hadzilacos, P. Kuznetsov, and S. Toueg, “The weakest failure detectors to solve quittable consensus and nonblocking atomic commit,” *SIAM J. Comput.*, vol. 41, no. 6, pp. 1343–1379, 2012.

[70] K. V. S. Ramarao, “Complexity of distributed commit protocols,” *Acta Informatica*, vol. 26, no. 6, pp. 577–595, 1989.

[71] I. Keidar and D. Dolev, “Increasing the resilience of atomic commit, at no additional cost,” in *PODS ’95*, pp. 245–254.

[72] R. Guerraoui, M. Larrea, and A. Schiper, “Reducing the cost for non-blocking in atomic commitment,” in *ICDCS ’96*, pp. 692–697.

[73] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, 2006.

[74] L. Lamport, R. Shostak, and M. Pease, “The byzantine generals problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[75] R. Guerraoui, “Indulgent algorithms (preliminary version),” in *PODC ’00*, pp. 289–297.

[76] R. Guerraoui and N. Lynch, “A general characterization of indulgence,” *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 4, pp. 20:1–20:19, 2008.

[77] G. Taubenfeld, “Computing in the presence of timing failures,” in *ICDCS ’06*, pp. 16–16.

[78] P. Dutta and R. Guerraoui, “The inherent price of indulgence,” *Distrib. Comput.*, vol. 18, no. 1, pp. 85–98, 2005.

[79] R. Guerraoui and M. Raynal, “The information structure of indulgent consensus,” *IEEE Trans. Comput.*, vol. 53, no. 4, pp. 453–466, 2004.

[80] L. Sampaio and F. Brasileiro, “Adaptive indulgent consensus,” in *DSN’05*, pp. 422–431.

[81] O. Bakr and I. Keidar, “Evaluating the running time of a communication round over the internet,” in *PODC ’02*, pp. 243–252.

[82] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[83] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[84] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[85] C. Cachin, R. Guerraoui, and L. A. Rodrigues, *Introduction to Reliable and Secure Distributed Programming.* Springer Berlin Heidelberg, 2011.

[86] B. Charron-Bost and A. Schiper, “Uniform consensus is harder than consensus,” *J. Algorithms*, vol. 51, no. 1, pp. 15 – 37, 2004.

[87] R. D. Prisco, B. Lampson, and N. Lynch, “Revisiting the paxos algorithm,” *Theoretical Computer Science*, vol. 243, no. 1, pp. 35 – 91, 2000.

[88] R. Guerraoui and A. Schiper, “The generic consensus service,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 29–41, 2001.

[89] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[90] P. Dutta, R. Guerraoui, and B. Pochon, “Fast non-blocking atomic commit: an inherent trade-off,” *Inform. Process. Lett.*, vol. 91, no. 4, pp. 195 – 200, 2004.

[91] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[92] M. Abdallah, R. Guerraoui, and P. Pucheral, "One-phase commit: does it make sense?" in *ICPADS '98*, pp. 182–192.

[93] Y. J. Al-Houmaily and P. K. Chrysanthis, "An atomic commit protocol for gigabit-networked distributed database systems," *J. Syst. Architect.*, vol. 46, no. 9, pp. 809 – 833, 2000.

[94] J. W. Stamos and F. Cristian, "A low-cost atomic commit protocol," in *SRDS '90*, pp. 66–75.

[95] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *EuroSys '13*, pp. 113–126.

[96] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 661–672, 2013.

[97] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *SIGMOD '12*, pp. 1–12.

[98] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Distributed transactions: Dissecting the nightmare," *CoRR*, vol. abs/1803.06341, 2018. [Online]. Available: http://arxiv.org/abs/1803.06341

[99] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[100] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jgadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang, "On brewing fresh espresso: Linkedin's distributed data serving platform," in *SIGMOD '13*, 2013, pp. 1135–1146.

[101] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR 2011*, 2011, pp. 223–234.

[102] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: Achieving serializability with low latency in geo-distributed storage systems," in *SOSP '13*, 2013, pp. 276–291.

[103] W. S. I. Jake Brutlag, "Speed matters," Online, 2009, https://research.googleblog.com/2009/06/speed-matters.html.

[104] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[105] L. Lamport, “On interprocess communication,” *Distributed Computing*, vol. 1, no. 2, pp. 77–85, Jun 1986.

[106] E. A. Brewer, “Towards robust distributed systems (invited talk),” in *PODC ’00*.

[107] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[108] R. J. Lipton and J. Sandberg, “Pram: A scalable shared memory,” Department of Computer Science, Princeton University, Tech. Rep. TR-180-88, 1988, https://www.cs.princeton.edu/research/techreps/TR-180-88.

[109] H. Attiya and J. L. Welch, “Sequential consistency versus linearizability,” *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, May 1994.

[110] M. Mavronicolas and D. Roth, “Linearizable read/write objects,” *Theoretical Computer Science*, vol. 220, no. 1, pp. 267 – 319, 1999, distributed Algorithms.

[111] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Bolt-on causal consistency,” in *SIGMOD ’13*, 2013, pp. 761–772.

[112] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[113] C. H. Papadimitriou, “The serializability of concurrent database updates,” *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[114] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, “Making geo-replicated systems fast as possible, consistent when necessary,” in *OSDI 12*, 2012, pp. 265–278.

[115] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *EuroSys ’15*, 2015, pp. 6:1–6:16.

[116] S. Burckhardt, *Principles of Eventual Consistency*. Now Publishers, October 2014, vol. 1.

[117] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza, “On verifying causal consistency,” *SIGPLAN Not.*, vol. 52, no. 1, pp. 626–638, Jan. 2017.

[118] S. Almeida, J. a. Leitão, and L. Rodrigues, “Chainreaction: A causal+ consistent datastore based on chain replication,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13, pp. 85–98.

[119] L. A. Prince Mahajan and M. Dahlin, “Consistency, availability, and convergence,” Department of Computer Science, The University of Texas at Austin, Tech. Rep. UTCS TR-11-22, 2011, http://www.cs.utexas.edu/users/dahlin/papers/cac-tr.pdf.

[120] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 141–155, Jan 2017.

[121] Z. Xiang and N. H. Vaidya, "Lower bounds and algorithm for partially replicated causally consistent shared memory," *CoRR*, vol. abs/1703.05424, 2017. [Online]. Available: http://arxiv.org/abs/1703.05424

[122] M. Roohitavaf, M. Demirbas, and S. Kulkarni, "Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, 2017, pp. 184–193.

[123] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *PPoPP '08*, 2008, pp. 175–184.

[124] H. Attiya, E. Hillel, and A. Milani, "Inherent limitations on disjoint-access parallel implementations of transactional memory," *Theory of Computing Systems*, vol. 49, no. 4, pp. 698–719, Nov 2011.

[125] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia, "Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, ser. PODC '15, 2015, pp. 217–226.

[126] D. Perelman, R. Fan, and I. Keidar, "On maintaining multiple versions in stm," in *PODC '10*, pp. 16–25.

[127] R. Guerraoui and J. Wang, "Optimal fair computation," in *DISC 2016*, 2016, pp. 143–157.

[128] S. Micali, "Simple and fast optimistic protocols for fair electronic exchange," in *PODC '03*.

[129] D. E. Knuth, "Open problems with a computational flavor, mimeographed notes for a seminar on combinatorics," 1971.

[130] D. Kleitman and D. Kwiatkowski, "A lower bound on the length of a sequence containing all permutations as subsequences," *Journal of Combinatorial Theory, Series A*, vol. 21, no. 2, pp. 129 – 136, 1976.

[131] L. Adleman, "Short permutation strings," *Discrete Mathematics*, vol. 10, no. 2, pp. 197 – 200, 1974.

[132] P. Koutas and T. Hu, "Shortest string containing all permutations," *Discrete Mathematics*, vol. 11, no. 2, pp. 125 – 132, 1975.

[133] J. Camenisch and I. Damgärd, "Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes," in *ASIACRYPT '00*, pp. 331–345.

[134] C. Schnorr, “Efficient signature generation by smart cards,” *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[135] D. Kravitz, “Digital signature algorithm,” 1993, uS Patent 5,231,668.

[136] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *CRYPTO ’86*, pp. 186–194.

[137] H. Ong and C. Schnorr, “Fast signature generation with a fiat shamir-like scheme,” in *EUROCRYPT ’90*, pp. 432–440.

[138] L. Guillou and J.-J. Quisquater, “A “paradoxical” indentity-based signature scheme resulting from zero-knowledge,” in *CRYPTO ’88*, pp. 216–231.

[139] G. Oded, *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2009.

[140] T. Dierks, “The transport layer security (tls) protocol version 1.2,” 2008.

[141] S. Goldwasser and S. Micali, “Probabilistic encryption,” *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270 – 299, 1984.

[142] A. C. Yao, “Theory and application of trapdoor functions,” in *SFCS ’82*, pp. 80–91.

[143] S. D. Gordon and J. Katz, *TCC ’09*, ch. Complete Fairness in Multi-party Computation without an Honest Majority, pp. 19–35.

[144] S. D. Gordon, C. Hazay, J. Katz, and Y. Lindell, “Complete fairness in secure two-party computation,” *J. ACM*, vol. 58, no. 6, pp. 24:1–24:37, 2011.

[145] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[146] I. 9594-8, “Information technology - open systems interconnection - the directory: Authentication framework,” 1995, (equivalent to ITU-T Recommendation X.509, 1993).

[147] A. Fiat and A. Shamir, “How to prove yourself: practical solutions to identification and signature problems,” in *CRYPTO ’87*, pp. 186–194.

[148] M. Bellare and P. Rogaway, “Random oracles are practical: A paradigm for designing efficient protocols,” in *CCS ’93*, pp. 62–73.

[149] A. Küpçü and A. Lysyanskaya, “Usable optimistic fair exchange,” *Computer Networks*, vol. 56, no. 1, pp. 50 – 63, 2012.

[150] G. Ateniese, “Efficient verifiable encryption (and fair exchange) of digital signatures,” in *CCS ’99*, pp. 138–146.

[151] A. M. Alaraj, “Simple and efficient contract signing protocol,” *CoRR*, vol. abs/1204.1646, 2012. [Online]. Available: http://arxiv.org/abs/1204.1646

[152] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., 2006.

[153] G. Draper-Gil, J.-L. A. Ferrer-Gomila, M. Hinarejos, and J. Zhou, “On the efficiency of multi-party contract signing protocols,” in *ISC 2015*, pp. 221–232.

[154] G. Goren and Y. Moses, “Silence,” *CoRR*, vol. abs/1805.07954, 2018. [Online]. Available: http://arxiv.org/abs/1805.07954

Jingjing WANG

Route de Crochy 14
1024, Ecublens
Switzerland

(0041) 078-845-6668
jingjing.wang@epfl.ch
people.epfl.ch/jingjing.wang

Research Experience

Distributed Programming Laboratory, École polytechnique fédérale de Lausanne, Switzerland
Research Assistant. September 2013 - Present, Supervisor: Prof. Rachid GUERRAOUI

- **Causally consistent key-value stores**: Discovered impossibility results of an efficient, available and eventually visible implementation of a causally consistent key-value store, implying the cost of a fast implementation of read transactions
- **Permissionless (distributed) blockchain**: Formalized the probability of block mining in the presence of network delays; proved that among honest miners, a miner of lower computational power can mine much fewer blocks than expected, which indeed depends on the network delays
- **Optimal distributed atomic commit**: Found the time and message complexity of atomic commit considering crash and/or network failures; proposed a delay-optimal protocol that tolerates crash and network failures
- **Optimal optimistic n-party fair computation**: Proved the first tight lower-bound on the message complexity by reducing the problem to the shortest permutation sequence in combinatorics
- **Private recommender systems**: Proved the differential privacy guarantee of random sampling in a user-based collaborative filtering recommender; designed and implemented the prototype of an efficient homomorphic encryption scheme for integers
- **Private k nearest neighbors**: Formalized the privacy guarantee in terms of information leakage in the context of distributed approximate k nearest neighbor algorithms

Cryptography and Information Security Laboratory, Shanghai Jiao Tong University, China
Research Assistant. December 2009 - March 2013, Supervisor: Prof. Kefei CHEN

- **Fast spectra attacks**: Proved bounds on the data complexity of fast spectra attacks against stream ciphers in general
- **Algebraic attacks**: Proposed one of the most efficient algebraic attack against nonlinear filter generators (of which Bluetooth E0 keystream generator is an example)

Publications

- Didona, D., Guerraoui, R., **Wang, J.**, and Zwaenepoel W.: Causal Consistency and Latency Optimality: Friend or Foe? (Alphabetical Order)
  VLDB 2018 (The International Conference on Very Large Data Bases)
- Guerraoui, R. and **Wang, J.**: On the Unfairness of Blockchain (Alphabetical Order)
  NETYS 2018 (The International Conference on Networked Systems)
- Guerraoui, R. and **Wang, J.**: How Fast can a Distributed Transaction Commit? (Alphabetical Order)
  SIGMOD/PODS 2017 (Symposium on Principles of Database Systems)
- Guerraoui, R., Kermarrec, A-M., Patra, R., Valiyev, M., and **Wang, J.**: I Know Nothing About You But Here Is What You Might Like (Alphabetical Order)
  DSN 2017 (International Conference on Dependable Systems and Networks)
- Guerraoui, R. and **Wang, J.**: Optimal Fair Computation (Alphabetical Order)
  DISC 2016 (International Symposium on Distributed Computing)
- Frey, D., Guerraoui, R., Kermarrec, AM., Rault, A., Taiani, F., **Wang, J.**: Hide & Share: Landmark-based Similarity for Private KNN Computation (Alphabetical Order)
  DSN 2015 (International Conference on Dependable Systems and Networks)
- **Wang, J.**, Chen, K., Zhu, S.: Annihilators of Fast Discrete Fourier Spectra Attacks
  IWSEC 2012 (International Workshop on Security)
- **Wang, J.**, Li, X., Chen, K., Zhang, W.: Attack Based on Direct Sum Decomposition against Nonlinear Filter Generator
  AFRICACRYPT 2012 (International Conference on Cryptology in Africa)

Conference Presentations & Invited Talks

- On the Unfairness of Blockchain. Conference presentation at NETYS 2018, Essaouira, Morocco. May, 2018
- How Fast can a Distributed Transaction Commit?. Conference presentation at SIGMOD/PODS 2017, Chicago, USA. May, 2017
- Optimal Fair Computation. Conference presentation at DISC 2016, Paris, France. September, 2016 (A preliminary version also presented at Winter School on Hot Topics in Distributed Computing, La Plagne, France. March, 2014)
- I Know Nothing About You But Here Is What You Might Like. Invited talk at ABB, Zurich, Switzerland. April, 2015
- Attack Based on Direct Sum Decomposition against Nonlinear Filter Generator. Conference presentation at AFRICACRYPT 2012, Ifrane, Morocco. July, 2012

Education

- **École polytechnique fédérale de Lausanne**, Lausanne, Switzerland
  PhD candidate, Computer, Communication and Information Sciences. September 2013 - Present
- **Shanghai Jiao Tong University**, Shanghai, China
  Master of Science, Computer Science and Technology, March 2013
- **Shanghai Jiao Tong University**, Shanghai, China
  Bachelor of Science, Computer Science and Technology, June 2010

Work Experience

Software Engineer Intern at Nuance Communications, Shanghai, China. April - July 2013

- **Text-to-speech**: Implemented and evaluated DAG-based word segmentation with different data structures for dictionary, including a compact radix trie, a hashtable trie, and with/without computation of floating points; implemented tools: a POS tagger based on Hidden Markov Model and the trigram model, and a tool to clean data from different encoding schemes
- **User-friendliness and efficiency**: Implemented core functions in C under Linux; ported implementations to Windows; augmented each C program with Python interfaces for both Linux and Windows platforms

Summer Intern Program at Microsoft STBC, Shanghai, China. June - September 2012

- **Colorado server maintenance**: Investigated root causes of user-reported issues of Colorado servers and debugged C++ source codes
- **Azure service monitor**: Implemented automatic report generation on performance and connectivity alerts for Azure services by System Center Operations Manager and shell scripts; documented the goal, requirements, and design overview of the project

Summer Internship at Saybot (Shanghai) Inc., Shanghai, China. July - September 2010

- **Internal website**: Implemented Drag&Drop file upload in Django, HTML5 and JQuery for Firefox, Chrome and IE browsers
- **Internal tool development**: Implemented tools in Python to document audio files in a directory into CSV and JSON, and slice and join audio files

Teaching Experience

- **Teaching Assistant** for CS 453 Concurrent Algorithms (Fall 2015, Fall 2016, Fall 2017), MATH 232 Probability and Statistics (Spring 2016), PHYS 114 General physics II (Fall 2014), at Faculté Informatique et Communications, École polytechnique fédérale de Lausanne, Switzerland
- **Teaching Assistant** for CS 413 Cryptography and Computer Security (Spring 2012, Fall 2011), MA 208 Discrete Mathematics (Fall 2010), at Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

Mentoring Experience

- **Supervisor** for a semester project (12 credits for master students) on "Implementation of an indulgent atomic commit protocol in CloudTPS", Lorenceau Pablo Camille, September 2017 - December 2017
- **Supervisor** for a summer internship on "Performance evaluation of an indulgent atomic commit protocol", De Moor Florestan Laurentin Marie, May - August 2017

Awards and Honors

- **EPFL IC School Fellowship** (2013)
- **Google Anita Borg Memorial Scholarship**: China (2012): Awarded based on the strength of both academic background and demonstrated leadership
- **National Excellence Scholarship** (2011): Highest honorary scholarship for graduate students awarded by Ministry of Education in China
- **Shanghai Jiao Tong University Scholarship for Graduate Study** (2011): First-class scholarship. Full tuition waiver and stipend of 260 yuan/month
- Second Prize in the National Post-Graduate Mathematical Contest in Modeling (2011)
- People's Scholarship (2009)
- Shanghai Jiao Tong University Outstanding Scholarship (2008; 2009)
- **National Scholarship** (2007; 2008): Highest honorary scholarship for undergraduate students awarded by Ministry of Education in China

Languages

Chinese Mandarin (native), English (fluent), French (basic)