# Distributed Trust

Dissertation zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Eingereicht von Klaus Kursawe

Rüschlikon, Dezember 2001

Accepted on the Ides (15th) of March 2002 on the recommendation of

Prof. Dr. Rainer Schulze-Pillot-Ziemen, Dean

Prof. Dr. Birgit Pfitzmann, examiner

Dr. Victor Shoup, co-examiner

# Abstract

We address the problem of collaboratively performing a task, in which neither all participants nor the communication channels can be trusted.

Cryptography plays an important role in our approach, thus the first problem addressed is to make the models used by the distributed systems/fault tolerance community compatible with cryptographic assumptions.

We then introduce some cryptographic primitives. On top of these we present a new solution to the Byzantine agreement problem. Our solution is fully asynchronous, and does not need additional constructs such as failure detectors. Thanks to modern cryptography, it is also quite efficient and can compete with timed protocols, which we prove with an implementation.

Once the Byzantine agreement is solved, we develop higher-level protocols to offer various broadcast primitives. At the top of our protocol stack is an atomic broadcast primitive that provides a powerful synchronization mechanism on a fully asynchronous network. To improve performance, we also present optimistic versions of some protocols that have a very high performance if the network behaves “normally”, but do not violate liveness or safety if it does not.

# Kurzzusammenfassung

Seit dem Unmöglichkeitsbeweis von Fischer, Lynch und Paterson von 1985 ist bekannt, daß die Koordination eines verteilten Systemes ein schwieriges Problem ist, falls weder alle Teilnehmer noch die Kommunikation zwischen ihnen vertrauenswürdig ist.

Mit Hilfe neuer kryptographischer Methoden können wir eine neue, probabilistische Lösung für das Problem der Byzantinischen Übereinstimmung anbieten. Dies erfordert jedoch zunächst, die Modelle aus verteilten und fehlertoleranten Systemen mit kryptographischen Annahmen in Einklang zu bringen. Insbesondere muß ein Äquivalent von *Lebendigkeit* in polynomial beschränkten Systemen definiert werden.

Unsere Lösung ist vollständig asynchron und benötigt keine weiteren Hilfsmittel wie beispielsweise Fehlerdetektoren.

Dennoch steht die Effizienz unseres Ansatzes nicht hinter Protokollen zurück, welche auf Annahmen über das zeitliche Verhalten der Parteien und des Netzwerkes bauen. Dies wird durch eine Implementierung unseres Protokolls belegt.

Aufbauend auf der Byzantinischen Übereinstimmung entwickeln wir dann verschiedene Verteilungs-Protokolle bis hin zur atomaren und sicheren kausalen Verteilung. Diese Protokolle liefern ein mächtiges Werkzeug zur Synchronisierung auf einem asynchronen Netzwerk, das die darauf aufbauende Applikationen weitgehend vor Asynchronität und unehrlichen Parteien schützt.

Um die Effizienz noch weiter zu steigern, führen wir optimistische Versionen einiger Protokolle ein. Unter "normalen" Umständen haben diese Protokolle einen sehr hohen Durchsatz, ohne im Falle von Problemen die Sicherheits- oder Lebendigkeits-Annahmen zu verletzen.

# Zusammenfassung

In dieser Arbeit werden Grundbausteine zur fehlertoleranten Koordination redundanter Prozesse entwickelt. Unsere Protokolle sind die ersten vollständig asynchronen Protokolle, die sowohl von der Effizienz als auch von den zugrundeliegenden Annahmen her in der Praxis benutzt werden können. Ähnliche Ansätze in der Literatur sind entweder nur bedingt praxistauglich, oder sie basieren auf zeitlichen Annahmen. Die meisten praktischen Systeme gehen zudem davon aus, daß fehlerhafte Teilnehmer lediglich die Arbeit einstellen, sich aber nicht aktiv bösartig verhalten.

Die Abhängigkeit von zeitlichen Annahmen ist nicht nur ein theoretischer Schönheitsfehler – praktische Angriffe auf solche Systeme sind bekannt [ACBMT95], und selbst in Abwesenheit eines bösartigen Angreifers kann es zu Problemen kommen [Bir01].

Obwohl unsere Protokolle in einem stärkeren Modell arbeiten, kann ihr Durchsatz sich mit anderen Protokollen in der Literatur messen, selbst wenn diese nur Ausfälle (im Gegensatz zu aktiven Störern) tolerieren und auf zeitlichen Annahmen aufbauen.

Dies wird zum einen durch neuartige kryptographische Protokolle ermöglicht, zum anderen durch optimistische Protokolle. Diese Protokolle benutzen zeitliche Annahmen zur Steigerung des Durchsatzes — im Gegensatz zu den zeitbasierten Protokollen in der Literatur werden diese Annahmen jedoch nur zur Steigerung des Durchsatzes benutzt und nicht um Sicherheit (informell: "es passiert nichts Schlechtes") oder Lebendigkeit (informell: "irgendwann passiert etwas Gutes") zu garantieren.

**Modell.** Die Kombination eines vollständig asynchronen Modells mit kryptographischen Annahmen ist erstaunlich unüblich. Bestehende Modelle im Bereich der Verteilten Systeme sind in der Regel nicht präzise genug – es ist üblich, eine idealisierte Form der Kryptographie zu verwenden, welche die zugrundeliegenden Annahmen ignoriert [Lyn96, p. 115]; oft widerspricht schon das Modell selbst elementaren kryptographischen Annahmen.

Unter den kryptographischen Modellen hingegen gibt es keine, mit denen sich unsere Probleme vollständig ausdrücken lassen. Neuere Modelle [PW01, Can01] decken die wichtigsten Aspekte ab, sind aber noch nicht umfassend genug, um die Lebendigkeitseigenschaften unseres Systems auszudrücken (siehe Kapitel 7).

Für diese Arbeit kombinieren wir das I/O – Automaten Modell von Lynch und Tuttle [LT87, LT89, Lyn96] mit einer Beschränkung der Rechenleistung, wie sie für kryptographische Bausteine benötigt wird.

In unserem Modell gibt es einen *Angreifer*, der sowohl das Netzwerk als auch bis zu einem Drittel der teilnehmenden Parteien (die *Störer*) kontrolliert. Es werden — nach der Initialisierung des Systems — keine externen Hilfsmittel wie vertrauenswürdige externe Systeme oder Fehlerdetektoren [CT91] benötigt.

**Kryptographie.** Unsere Protokolle basieren auf zwei wichtigen kryptographischen Bausteinen. Ein *gemeinsamer Münzwurf* generiert eine (pseudo)-zufällige Münze. Solange nicht hinreichend viele Parteien an diesem Protokoll teilnehmen, ist der Ausgang dieses Münzwurfes (aus Sicht der Teilnehmer) völlig offen und unvorhersagbar. Eine andere wichtige Eigenschaft, die unseren Münzwurf von ähnlichen Protokollen in der Literatur unterscheidet [Rab83, BS93], ist eine praktisch unbegrenzte Anzahl von Münzen, auf die in einer beliebigen Reihenfolge zugegriffen werden kann, d.h., die Münzen werden durch eine (verteilt berechnete) Abbildung von Münznamen auf Werte generiert und nicht durch eine sequentielle Abfolge von Würfen. Dies erlaubt es uns, mehrere Instanzen des Münzwurfes parallel auszuführen, ohne Koordinierungsprobleme zu bekommen.

Der zweite Baustein für unsere Protokolle sind Schwellwertunterschriften (engl. *threshold signatures*). Dies sind normale digitale Unterschriften, die allerdings von mehreren Teilnehmern generiert werden müssen; anstatt zu beweisen, daß *ein* Teilnehmer diese Unterschrift geleistet hat, beweisen sie, daß *mehrere* Teilnehmer dies getan haben. Eine triviale Lösung ist es, einfach mehrere normale Unterschriften zu einer zusammenzufassen. Allerdings gibt es ein eleganteres Schema, in dem die kombinierte Unterschrift eine konstante Größe hat, unabhängig von der Anzahl der Teilnehmer [Sho00]. Für größere Systeme ist dieses Schema deshalb besser geeignet.

Die meisten Einschränkungen, welche wir dem Angreifer aufzwingen müssen, werden für die Beweise dieser kryptographischen Bausteine benötigt. Eine Beschränkung der Rechenleistung ist nötig, da der Angreifer sonst durch simples Raten alle kryptographischen Geheimnisse herausfinden kann. Die Simulationsbeweise benötigen desweiteren einen statischen Angreifer, d.h., die Störer werden ausgewählt, bevor die erste Nachricht verschickt wird. Das bedeutet nicht, daß die Protokolle ansonsten unsicher sind; wir wissen lediglich nicht, wie wir im dynamischen (d.h., nicht-statisch, der Angreifer darf zu jedem Zeitpunkt eine ehrliche Partei korrumpieren) Modell die Sicherheit der Bausteine beweisen können. Ähnlich verhält es sich mit dem *Random Oracle* Modell, einer idealisierten Hash-Funktion, die keinen berechenbaren Zusammenhang zwischen Ein– und Ausgabe herstellt.

**Byzantinische Übereinstimmung.** Das Problem der Byzantinischen Übereinstimmung ist, daß sich $n$ Parteien mit (binären) Eingabewerten auf einen gemeinsamen Wert einigen müssen — und das obwohl einige der involvierten Parteien bösartig sind und aktiv versuchen, diese Einigung zu verhindern.

Seit dem "Unmöglichkeitsbeweis" von Fischer, Lynch und Paterson [FLP85] ist bekannt, daß es keine deterministische Lösung zu diesem Problem gibt. Probabilistische Lösungen existieren [Rab83, BO83], wurden jedoch allgemein als unpraktisch angesehen. Bisherige Lösungen sind entweder nur unter speziellen Bedingungen nutzbar, z.B. wenn die relative Anzahl der Störer sehr klein ist [BO83], oder sie sind zu ineffizient für einen praktischen Einsatz [CR93].

Durch neue, effiziente kryptographische Techniken kann dieser Ansatz zu einem praktischen, beweisbar sicheren Protokoll zur Byzantinischen Übereinstimmung benutzt werden, das die maximal mögliche Anzahl an Störern toleriert.

**Einfache Verteilungsprotokolle.** Verteilungsprotokolle (engl. *broadcast protocols*) werden dazu benutzt, Nachrichten zwischen einem Sender und mehreren Empfängern zu übermitteln. Man unterscheidet dabei mehrere Klassen von Verteilungsprotokollen, die verschiedene Eigenschaften der Verteilung garantieren müssen. Eine mögliche Eigenschaft ist zum Beispiel, daß eine Nachricht entweder alle oder keinen ehrlichen Empfänger erreicht, unabhängig von der Ehrlichkeit des Senders.

Zwei einfache, aber wichtige Verteilungsprotokolle sind *zuverlässige Verteilung* und *konsistente Verteilung.* Zuverlässige Verteilung [BT85] garantiert, daß alle Teilnehmer die gleichen Nachrichten erhalten. Konsistente Verteilung ist etwas schwächer und stellt sicher, daß über den Inhalt ausgelieferter Nachrichten Einigkeit herrscht, nicht jedoch über die Menge ausgelieferter Nachrichten. Dies ist insbesondere nützlich im Zusammenhang mit Nachprüfbarkeit, welche es dem Empfänger einer Nachricht erlaubt, einem anderen Teilnehmer den Empfang dieser Nachricht effizient zu beweisen. Wir beschreiben effiziente Implementierungen von zuverlässiger und konsistenter Verteilung.

Ist der Sender fehlerhaft, garantiert konsistente Verteilung keine Einigung auf Nachrichten, und beide Protokolle garantieren keine Terminierung (zuverlässige Verteilung garantiert allerdings, daß entweder alle oder keine ehrliche Partei terminiert); deshalb wird keine Byzantinische Übereinstimmung benötigt.

**Nicht-Binäre Byzantinische Übereinstimmung.** Obige probabilistische Übereinstimmungsprotokolle funktionieren nur für binäre Entscheidungen. Erweiterungen auf größere Bereiche [Rab83] erlauben dem Protokoll normalerweise, sich auf einen vordefinierten Wert zu einigen, den keiner der Teilnehmer ursprünglich vorgeschlagen hat. Für viele Anwendungen – zum Beispiel atomare Verteilung – ist dies nicht ausreichend. Wir benötigen ein Übereinstimmungsprotokoll für beliebige Werte, das zudem garantiert, daß das Resultat "sinnvoll" ist. Wir stellen ein neues *nicht-binäres Byzantinisches Übereinstimmungsprotokoll* mit einer *externen Gültigkeitsbedingung* vor. Die externe Gültigkeitsbedingung garantiert, daß das Resultat des Übereinstimmungsprotokolls für die aufrufende Anwendung Sinn macht.

Das nicht-binäre Byzantinische Übereinstimmungsprotokoll ruft im Durchschnitt eine konstante Anzahl an binären Übereinstimmungsprotokollen auf, was durch eine neue Anwendung des gemeinsamen Münzwurfes erreicht wird. Es toleriert die maximale Anzahl an Störern und benötigt eine Anzahl von Nachrichten, die quadratisch in der Anzahl der Teilnehmer ist – dies ist im wesentlichen optimal.

**Atomare und Sichere Kausale Verteilung.** Ein *atomares Verteilungsprotokoll* garantiert, daß alle Parteien die gleiche Menge von Nachrichten in der gleichen Reihenfolge bekommen. Dies ist im Wesentlichen eine Einigung auf eine geordnete Menge von Nachrichten. Dieses Problem unterscheidet sich von den obigen Problemen insofern, daß die verschiedenen Protokollinstanzen nicht unabhängig sind – der Sinn eines atomaren Verteilungsprotokolles ist es, diese Instanzen (bzw. die dazugehörigen Nachrichten) untereinander zu koordinieren. Aus diesem Grund benötigen wir eine Verfeinerung unserer Definition der Terminierung und zusätzlich einen Begriff von Fairness. Unser Protokoll garantiert, daß die Nachricht eines ehrlichen Senders nicht ewig verzögert werden kann, sobald eine Mindestanzahl ehrlicher Parteien von dieser Nachricht kenntnis hat. Das Protokoll baut auf nicht-binärer Byzantinischer Übereinstimmung auf und benötigt einen Aufruf des Übereinstimmungsprotokolles für jeden ausgelieferten Satz Nachrichten.

Wir stellen ebenfalls ein optimistisches Protokoll vor, das zeitliche Annahmen zur Steigerung des Durchsatzes benutzt [CL99b], Sicherheit und Lebendigkeit jedoch durch probabilistische Protokolle ereicht. Im "Normalfall" sind die Kosten einer Nachricht ungefähr so hoch wie bei einem zuverlässigen Verteilungsprotokoll, und es wird keine teure Kryptographie verwendet. Dies ermöglicht eine immense Steigerung des Durchsatzes; die Geschwindigkeit unseres optimistischen Protokolles steht synchronen Protokollen in nichts nach.

Schließlich definieren und implementieren wir eine stärkere Variation der atomaren Verteilung, die *sichere kausale atomare Verteilung*. Hier wird zusätzlich noch eine kausale Ordnung der ausgelieferten Nachrichten garantiert, indem die Nachricht bis zur Auslieferung geheim gehalten wird. Dadurch ist es nicht möglich, daß eine Nachricht in irgendeiner sinnvollen Weise von einer später ausgelieferten Nachricht abhängt.

# Contents

# List of Figures

# Acknowledgments

After all those years, I've got quite a list of people who contributed in some way to this thesis, for which I would like to express thanks.

First of all my thanks go to Victor Shoup, my supervisor, for the enthusiasm and inspiration, which was always there when I needed it.

Birgit Pfitzmann, who gave me the most detailed corrections imaginable, and who also is a perfect oracle for all security related questions.

If Victor and Birgit are the ”parents” of this thesis, then Christian Cachin who should be considered the godfather.

Anna, who inspired new ideas by just being around (outside of being extremely clever, so she could tell me instantly why they where wrong).

Michael, who managed our group in a way that we never felt the icy breath of reality in the industrial world too badly, such that I could concentrate on the real issue.

Helga as a stand-in mother figure, for just keeping all our backs free, both internally and externally.

Nicola, Doris and Eliska form our library, for getting any book or article for me, no matter how obscure the request.

Charlotte and Lilli-Marie for their help with the english language.

Ahmad, for introducing me to the the people that finally gave me an office and funding to do my PhD.

And of course our little gang that made Zurich a very special place over all those years: Laura, James, Mike, Daniela, Stefan, Fay, Rob, April, Hugo, Dot, Brian, Ariane, Els, Paulo, Anthony, Endre, Joy, Dominique, Sandra, Sonja, Gero, Frank and all the others who have passed through.

I would also like to thank my parents for creating an environment in which following this path seemed so natural.

And last but not least Laura, who had to share me with my thesis all the time, and much to often got the smaller share.

# Chapter 1

# Introduction

The Byzantine generals were desperate. For generations their armies had successfully conquered city after city, and their empire was mightier than ever before. And then this guy appeared out of nowhere. "So how do you know you can trust each other ?", he asked. At first they laughed. A traitor among the Byzantine generals — that was beyond their imagination. But the seed of doubt had been planted. What *if* we cannot trust each other? What *if* only half of our army attacks, being left alone by a traitorous general? And thereafter their army stood immobilized at the walls of Lamportium, not daring to make a decision to attack in the presence of potential traitors...

**Byzantine**. adj. **a.** Belonging to Byzantium or Constantinople; also, reminiscent of the manner, style, or spirit of Byzantine politics. Hence, intricate, complicated; inflexible, rigid, unyielding.
*from: The Oxford English Dictionary, 2nd edition, ca. 500 years after the so called Byzantine Empire ceased existing.*

## 1.1 Distributing Trust

Trusting a single entity in the real world is and always has been a difficult business. If this entity is computerized, however, it becomes close to impossible.

The structure of a given entity somewhere on the Internet can be quite complex; what appears to be one entity may be a complex relation between programmers, system administrators, site owners etc.; and this does not even include the possibility of outside attacks by hackers or viruses. This not only makes it difficult to understand whom one actually trusts and how much one trusts them if computers are involved. It is also much harder to resolve problems, as legal positions involving computerized transactions are highly complex and have many unsolved problems.

Distributing trust among several entities is a natural solution. There are several different ways to achieve this. Which one is the best choice largely depends on the task to be distributed, and the exact trust model: while a redundant government seems to be quite difficult to implement, it is just as ridiculous to apply Montesquieus [Mon48] separation of power on rivets that hold together an airplane. When the task to be distributed becomes more complex than holding together some pieces of metal that form an airplane, the entities forming the system have to be coordinated. This coordination should not be done by one single entity, as this would again introduce a single point of failure; rather, it has to be done in a redundant way as well.

The scope of this thesis is to coordinate the different entities that form a redundant system in spite of some of the entities behaving *Byzantine*, i.e., maliciously trying to disrupt the system, and in spite of fully asynchronous communication between these entities.

To this end, we will provide robust protocols for agreement and broadcast protocols with different qualities. These protocols will ignore faulty parties rather than trying to detect them. We will use new cryptographic primitives, and present a combined model that allows a thorough analysis of the algorithms, while being broad enough to capture all aspects needed.

## 1.2 Related Work

The area of fault tolerance by redundancy is as old as automated computing itself [Bab74]. The first mechanical and vacuum-tube-based computers had to deal with unreliable components that could behave completely unpredictably. Thus, fault tolerance in the early days consisted mostly of dealing with unreliable hardware, and accordingly most solutions where based on hardware redundancy [Pie65].

Starting in the early 70s, the scope shifted towards software-implemented fault tolerance [Ran75, Hec76, Wen72]. One of the early works is the SIFT (Software Implemented Fault Tolerance) system [WLG+78], which discusses fault tolerance in aircraft control systems, and is one of the first approaches to implement fault tolerance in software.

In the context of the SIFT project, the problem of finding agreement on an output value in spite of faulty components, the *consensus* problem, was identified as fundamental for fault-tolerant computing [MPL80, LSP82]. This triggered a rapid development: in 1983, a groundbreaking paper showed that in a fully asynchronous system, consensus is difficult even if only one party is allowed to crash [FLP85]; more precisely, the authors show that no deterministic algorithm could solve consensus. Already in the same year, the first randomized algorithms appeared that circumvented this proof [Rab83, BO83]. Although these protocols triggered a boom in randomized consensus protocols, this approach is now out of fashion and mainly neglected by the fault tolerance community, being regarded as a more theoretical concept.

Around the same time as the early randomized protocols, the first middleware for dependable systems was developed [BJ87]. Starting in 1984, *Isis* was one of the first projects on dependable distributed systems [BJ87]. Isis is based on a failure detector, that is an abstraction of a local mechanism that can detect (with some certaincy) if another party crashes. Using this failure detector, a group membership service is offered that removes faulty parties from the active group and maintians a consistent view of the status of the group. This approach has since experienced a continuous evolution, resulting in various spin-offs. Work on the successors of Isis is still in progress [vRBM96, Hay98], and first attempts to tolerate Byzantine failures have been made [Rei95].

## 1.3 Contribution of this Thesis

In spite of the huge amount of work that has so far been done in fault-tolerant computing, most practical solutions that exist today are designed for fast, reliable networks with crash failures only. The environment we want our distributed system to run in, the Internet, provides few timing guarantees, and is wide open for malicious attacks. Therefore, our protocols have to live in a world with fewer guarantees and more malice than previous protocols described in the literature. However, to be acceptable in practice, their performance must be roughly comparable with that of existing implementations.

We achieve this by exploiting new cryptographic primitives that allow us to implement efficient, practical randomized protocols. These protocols are very robust, as they do not depend on timing assumptions and do not need to identify corrupted parties. Nevertheless, the efficiency is comparable to failure-detector-based protocols. We also introduce optimistic protocols that combine techniques from the failure-detector model with randomization. Here, timing assumptions are used to improve performance, while the robustness of the randomized protocols is maintained.

The protocol stack developed in this thesis provides roughly the same service as dependable Middleware of the Isis family does, which allows existing protocols to use it without major modifications.

### 1.3.1 Marrying Fault Tolerance with Security

This thesis is located between the fields of fault-tolerant distributed systems and cryptographic security. There are very few works in the literature that combine these fields. The distributed-systems community tends to use cryptography as an idealized black box that provides useful primitives, without taking the underlying model into account [Lyn96, p. 115]. In the cryptographic community, very precise models for multiparty computation exist, but so far no model is sufficiently exhaustive to cover all aspects of the problems addressed here. In this thesis, problems of the distributed systems / fault-tolerance community are attacked, while opening the "black box" that contains the cryptography and taking into account the impact this has on the system model. The black box was in fact the box of Pandora: the definitions and assumptions used in fault tolerant systems are incompatible with the cryptographic model, and many definitions used in asynchronous (timeless) systems do not work together with computationally bounded adversaries. For this reason, the models and definitions for the problems solved in this thesis had to be altered and refined.

### 1.3.2 Randomization.

Since the seminal paper by Fischer, Lynch and Paterson (FLP) [FLP85], it is known that in a fully asynchronous setting, deterministic protocols can hardly do anything useful. Essentially, the ways to circumvent their impossibility result are to use either randomized protocols, or apply some synchrony assumptions. The latter approach is the one in fashion today. For some reason, many people seem to think that using randomization is cheating, and that (outside of being inherently inefficient) it is inappropriate for real systems.

This thesis demonstrates the practicality of the randomized approach with new protocols, along with an implementation described in Section 4.8. The performance of our protocols can well compete with timing-based Byzantine protocols in the literature, and even in the crash-failure model out protocols are competitive.

By providing a significantly higher robustness with similar performance, we prove wrong the assumption that randomized protocols are inherently unpractical. The opposite is the case, especially in a Byzantine setting randomization seems to be superior to protocols that depend on timing assumptions [Bir01, ACBMT95].

There remains a problem that many people in distributed systems feel uncomfortable with randomization — telling people that we use randomized protocols sometimes feels like admitting that we are using smoke and mirrors.

Therefore, it is important at this point to argue that using randomization does not weaken the model; in fact, in a Byzantine setting almost all protocols have an implicit failure probability, and our only "crime" is to make the probabilistic arguments explicit.

In a Byzantine environment, we have an adversary that actively tries to attack the system. To do anything useful, however, some form of message authentication between honest parties is required. There are two ways to achieve that.

– There is a separate hardware link between each pair of parties, and the adversary is assumed to be incapable of accessing these links. This may be a plausible setting in some environments, but in most networks every party has only one cable to connect to the outside. Furthermore, if a separate hardware link exists between all honest parties, it does not make much sense to consider asynchrony.

– Messages are authenticated by some software means. In this case, the authentication is based on some secret, which the adversary can guess with some small, but nonzero probability

Thus, in a real world, essentially all protocols subject to an attack by an adversary are only deterministic on an abstract level; any implementation can be broken with some probability, thus introducing randomization does not weaken the protocol once it is actually being used.

### 1.3.3 Optimism

In comparing the average efficiency of our protocols with that of existing approaches, the fully asynchronous approach naturally has a disadvantage compared with algorithms that work in a weaker model. Even though our protocols turned out to be quite competitive (e.g., losing only a factor of 1.5 to 2 for Byzantine agreement), the goal is to reach the same or even better efficiency.

This leads to the concept of optimism. In the "normal" case, i.e., if the system is not under attack and the network behaves reasonably, the protocol can terminate very quickly; however it will discovers whether something went wrong. In this case, it switches to the slower, but more secure protocols. Not only does this reduce the communication complexity. In the optimistic case, it also allows us to get rid of all expensive cryptographic operations.

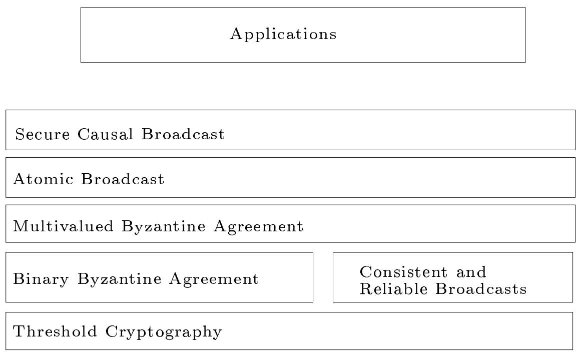
### 1.3.4 Middleware for Higher-Level Applications

The protocols developed in this thesis can be seen as middleware to support fault tolerant higher level applications.

In their extension to the FLP impossibility proof [DDS87], Dolev *et al.* identified the existence of an atomic broadcast primitive as sufficient to solve deterministic Byzantine Agreement in an asynchronous system. Subsequently, it has become a popular synchronizer, protecting high level protocols from the problems of asynchronous communication. Several projects implemented various flavors of atomic broadcast [BJ87, Rei95, Hay98, vRBG$^+$95, MMSA94]. However, all but one approach tolerate only crash-failures, and all atomic broadcast primitives in the literature rely heavily on failure detectors. Furthermore, no rigorous security proofs are given. In fact, practical attacks have been identified for many existing systems [ACBMT95]. This thesis presents the first atomic broadcast primitive that tolerates Byzantine failures and full asynchrony. Furthermore, it is provably secure and maintains comparable performance. This, along with the other protocols presented here, is sufficient to build a distributed state machine and similar constructs, allowing a fault-tolerant implementation of relatively generic protocols.

## 1.4 Overview

After introducing the model we work in, the thesis will mostly follow the following protocol stack, starting with the cryptographic tools in Chapter 3 up to Atomic and Secure Causal Broadcast in Chapter 8.


Applications
Secure Causal Broadcast
Atomic Broadcast
Multivalued Byzantine Agreement
Binary Byzantine Agreement
Consistent and Reliable Broadcasts
Threshold Cryptography

**Model.** We work in the fully asynchronous model, i.e., we do not make any assumptions about timing – from the point of view of our protocols, time does not even exist. We assume a static set of servers and rely on a trusted dealer that is used once to set up the system; after setup however, the dealer is never used again and we do not need any additional external constructs later (such as failure detectors or (trusted) outside parties). These choices maintain the safety of the service no matter how bad the timing behavior of the network is – even if the network is temporarily disrupted (e.g., by a denial–of–service attack). This model also avoids the problem of having to assume synchrony properties and of fixing timeout values for a network that is controlled by an adversary; such choices are difficult to justify if liveness and especially safety depend on them, as is shown by practical attacks [ACBMT95] and timing problems in the implementations of Horus and Ensemble that lead to instability if the groups grow too large [Bir01].

Two important cryptographic building blocks are needed by our protocols. A *common coin* is a coin-tossing primitive that creates a (pseudo-) random coin. The chief property of this primitive is that the value of the coin is unpredictable until a sufficient number of honest parties participate in the tossing of the coin. Other important properties that distinguish our primitive from common coins as proposed in the literature [Rab83, BS93] is that the number of coins to be tossed should be virtually unlimited, and that the coins can be accessed in a random order – that is, if the coins are not accessed as a sequential stream of coins, but as a function mapping coin names to values. This allows several instances of the coin protocol to run in parallel without having to be coordinated.

The second primitive we need are *threshold signatures*. Threshold signatures are like normal digital signatures, but rather than showing that *one* party signed a message $m$, they prove that *several* parties signed $m$. It is always possible to use a trivial solution by just concatenating several ordinary signatures. However, more elegant schemes exist that create constant-size signatures and are thus better applicable for larger groups of parties [Sho00].

Most restrictions we impose on the adversary are due to the cryptographic proofs. Naturally, we have to restrict its computation power to protect the cryptographic keys. Also, the simulation proofs used require a static adversary, i.e., the corrupted parties are chosen before the protocol starts. This does not imply that otherwise the protocols can be broken; we merely do not know how to prove security if the corruptions depend on the the run of the protocol. Similarly, we need the random oracle model, i.e., access to idealized hash functions that impose no computational relations between input and output.

**Byzantine Agreement.** Despite the practical appeal of the asynchronous model, not much research has concentrated on developing efficient asynchronous protocols or implementing practical systems that need consensus or Byzantine agreement. Since the result of Fischer, Lynch, and Paterson [FLP85], which shows that even in an asynchronous system subject to only a single crash failure, no deterministic protocol for Byzantine agreement exists, the practical approach generally used was to weaken the asynchrony, whereas randomized protocols were avoided. The first randomized solutions were only usable in special circumstances, for example, in very small groups of parties or with a small number of corruptions [BO83], or in the presence of a trusted dealer after a constant number of protocol invocations [Rab83]. By employing modern, efficient cryptographic techniques, this approach can be extended to a practical yet provably secure protocol for Byzantine agreement in the cryptographic model that withstands the maximal number of possible corruption.

**Low-Level Broadcast.** The basic broadcast protocols (following Bracha and Toueg [BT85]) are *reliable broadcast*, which ensures that all servers deliver the same messages, and a variation of it that we call *consistent broadcast*, which only provides agreement among the actually delivered messages. Consistent broadcast is particularly useful in connection with a *verifiability* property for the delivered messages, which ensures that a party can transfer a "proof of delivery" to another party in a single piece of information. We describe message- and communication-efficient implementations of reliable and consistent broadcast. Both of these broadcast primitives do not ensure termination for faulty senders (although reliable broadcast guarantees that either all or none of the honest parties terminate), and consistent broadcast does not provide agreement on the messages. Therefore, these broadcasts can be implemented without Byzantine agreement.

**Multivalued Byzantine Agreement.** The randomized agreement protocols mentioned above work only for binary decisions. Extensions to larger domains usually allow the protocol to decide on a fallback value that no party initially proposed [Rab83]. For many applications – for example, for the atomic broadcast protocol below – this is not sufficient. One also needs agreement on values from large sets, while guaranteeing that the agreement value is "useful". We propose a new *multi-valued Byzantine agreement* protocol with an *external* validity condition. External validity ensures that the decision value is acceptable to the particular application that requests agreement.

The multi-valued Byzantine agreement protocol invokes only a constant number of binary Byzantine agreement sub-protocols on average and achieves this by using a cryptographic common coin protocol in a novel way. It withstands the maximal possible corruption of up to one third of the parties and has expected quadratic message complexity (in the number of parties), which is essentially optimal.

**Atomic and Secure Causal Broadcast.** An atomic broadcast protocol ensures that all parties receive the same set of broadcast messages in the same order; essentially, it is an agreement on messages and their order. This problem is rather different from the one described above, in that different invocations of the protocol are not independent – by agreeing on an order of messages, the very purpose of the protocol is to coordinate the protocol invocations (i.e., broadcast requests). This needs a refinement of our definitions of termination, as well as some sort of fairness. Our protocol guarantees that a message from an honest party cannot be delayed arbitrarily by an adversary as soon as a minimum number of honest parties are aware of that message. The protocol invokes one multi-valued Byzantine agreement per batch of payload messages that is delivered.

We also present an optimistic protocol along the lines of [CL99b]. In the "normal" case, the cost of one broadcast is roughly equivalent to a reliable broadcast, and no cryptography is needed. This offers a vast performance improvement and allows our approach to compete with and partially outperform timed implementations of similar services.

Finally, we define and implement a variation of atomic broadcast called *secure causal atomic broadcast*. This is a robust atomic broadcast protocol that tolerates a Byzantine adversary and also provides secrecy for messages up to the moment at which they are guaranteed to be delivered. Thus, client requests to a trusted service using this broadcast remain confidential until they are answered by the service. In our asynchronous environment, this is crucial for applying the state-machine replication method to services that involve confidential data.

Secure causal atomic broadcast works by combining an atomic broadcast protocol with robust threshold decryption. This notion and a heuristic protocol were proposed by Reiter and Birman [RB94], who called it "secure atomic broadcast" and also introduced the term "input causality" for its properties. Recent progress in threshold cryptography allows us to present an efficient robust protocol together with a security proof in the appropriate formal models from cryptography.

# Chapter 2

# System Model

In this chapter, we describe our basic system model for an arbitrary multi-party protocol where a number of parties communicate over an insecure, asynchronous network, and where an adversary may corrupt some of the parties.

This thesis is situated between the areas of cryptography and fault-tolerant distributed computing. Our model abuts on the I/O automata model of Lynch and Tuttle [LT87, LT89, Lyn96], which we adapt to the polynomial-time model, a basic requirement for using cryptographic primitives.

Alternatively, we could have adapted cryptographic models for multiparty computation [Gol01b, PW01, Can01]. However, by the time this thesis was started, those models where not broad enough to cover asynchronous reactive systems, and moreover, owing to their significantly higher precision, they are quite complex and not easy to extend.

## 2.1 Basic Setting and Definitions

In an $n$-party multiparty protocol, $n$ parties $P_1, \ldots, P_n$ are connected by a point-to-point network. We assume a fully asynchronous network, i.e., there are no bounds on network delays and computation time; in fact, our model provides no time at all.

At the beginning, we assume an initialization phase, wherein a trustworthy dealer distributes cryptographic keys among the participants. The dealer then is destroyed and never used again.

**Parties.** The system consists of $n$ parties (or players, or processors). The parties are computationally bounded, i.e., there exists a security parameter $\kappa$, and a party is allowed a number of computation steps bounded by a polynomial in $\kappa$ [LP81, Gol01a].

**The adversary.** The system is attacked by an adversary that has complete control over the environment; the network as well as higher- and lower-level protocols are completely incorporated into the adversary. Furthermore, the adversary corrupts up to $t$, $t < n/3$, of the participating parties. Like the parties, the adversary runs in time bounded by a polynomial in $\kappa$. We assume a static adversary, i.e., the adversary has to decide which parties to corrupt before seeing the first protocol messages; this is the only restriction on the choice of corruptions.

**Communication.** The parties communicate via a fully connected point-to-point network. We assume authenticated links, i.e., it is not possible to manipulate the content or fake the origin of a message; this can be implemented quite cheaply using message authentication codes. The protocols presented here neither need private links, i.e., the adversary is allowed to read all communication between honest parties, nor a broadcast channel. It is straightforward to modify our protocols to work in a partially connected network if all honest parties are connected either directly or indirectly via a sufficient number of independent routes; if this is not the case, an honest party can be completely cut off, and it is impossible to do anything useful at all. The communication is fully asynchronous, i.e., there are no timing restrictions on the message delay on the channels.

**The problem.** To understand some of the choices made in our model, it is helpful to have an example application in mind. We use the problem of *distributed consensus* for this purpose. The definition given here is only for demonstration purposes; a more refined definition is given in Section 4.3.

Four important properties make consensus an excellent demonstration object and one of the most prominent problems in fault-tolerant distributed computing:

– it is simple to state;

– it is "necessary" (hardly anything useful can be done if parties cannot even find consensus);

– it is "sufficient" (many important problems can be reduced to consensus);

– it is difficult to solve (as shown for example by Fischer, Lynch and Paterson [FLP85] and the fact that it is still possible to write a thesis about it 20 years after its definition).

**Definition 2.1 (Distributed Consensus).** Given $n$ parties $P_1, \ldots, P_n$, each of which gets an input value $v_i \in \{0, 1\}$. A protocol solves distributed consensus if the following holds:

**Termination**: All honest parties eventually output (decide) something.

**Agreement**: All honest parties output (decide on) the same value. This decision is final.

**Validity**: For all values $\rho \in \{0, 1\}$, there is at least one input vector and some admissible execution of the protocol such that $\rho$ is the decision value.

There are various variations of this definition, especially regarding the validity condition. In our setting, where some of the parties behave actively malicious, consensus is also called *the Byzantine agreement problem* [LSP82].

To allow several instances of a protocol to run simultaneously, each protocol instance has a unique identifier *ID*, which is included in all messages belonging to this protocol instance.

## 2.2 Cryptography

In incorporating cryptography into our model, instead of treating it as an idealized black box, our way of defining a distributed system differs significantly from the way used in the literature.

Most importantly, we work in a computationally bounded model. This requires us to define some form of computation time within the asynchronous model. Furthermore, some concepts used in the literature (e.g., eventuality and infinitely long runs) do not make sense in a computational model, and adequate replacements have to be found.

Secondly, the cryptographic model is much more adversary-oriented than usual in distributed systems. In our model, there is no "outside world"; everything that is not an instance of our protocol running on an honest party is incorporated into the adversary. This is a much more pessimistic view than the literature on distributed systems usually takes, and certainly more pessimistic than any real-world scenario; however, for security proofs, this pessimism is appropriate.

So far, there is no cryptographic model that is comprehensive enough to cover all aspects of the problem setting we have; especially the liveness of atomic broadcast (see Chapter 7) does not fit into any model. The area of *multiparty computation* dates back about 20 years [SRA81, Yao82a]. Most commonly, the "ideal-world/real-world" model [MR91, Gol01b] is used. Problems are specified by defining a protocol that lives in an idealized world (the *ideal world*) with all sorts of nice properties such as a trusted third party; the real protocols lives in our real world and thus have to deal with all sorts of real problems. A security proof in this model then shows that the real-world and the ideal-world protocols are equivalent — the precise definition of that term differs with the various flavors of this approach.

Traditionally, multiparty computation addresses only non-interactive functions that receive an input once, perform a computation, and produce an output. This is sufficient to tackle most of our problems; some problems, however, have a continuous stream of inputs that has to be dealt with (see the atomic broadcast in Chapter 8).

Pfitzmann and Waidner [PW00] and Canetti [Can00] were the first to model synchronous reactive systems. Both models have now been advanced into the asynchronous model [PW01, Can01], and now cover asynchronous, computationally bounded reactive systems. It is not possible though to express liveness in the way we need it (see Chapter 7).

### 2.2.1 Cryptographic Definitions

**Security parameter.** The security parameter of our computational security model is denoted by $\kappa$. We make the convention that the parameter $n$ is bounded by a fixed polynomial in $\kappa$, independent of the adversary.

We make a similar assumption on the sizes of all messages in the protocol: excessively large messages are never generated by or allowed to be delivered to honest parties.

**Negligibility.** We say that a function $\epsilon(\kappa)$ is called *negligible* if for all $c > 0$ there exists a $\kappa_0$ such that $\epsilon(\kappa) < \frac{1}{\kappa^c}$ for all $\kappa > \kappa_0$. We will consider negligible functions also in other parameters than in $\kappa$, but we assume that the parameter of a negligible function is $\kappa$ if not explicitly mentioned otherwise. In this sense, a "negligible quantity" is a negligible function in the security parameter $\kappa$. As $\kappa$ is sometimes not mentioned in other contexts either, keep in mind that all system parameters are bounded by polynomials in $\kappa$. We say that some *quantity* is negligible if it is bounded by a negligible function in $\kappa$. We say a probability for event $X$ is *overwhelming* if the probability for the event $(\neg X)$ is negligible.

**Infeasibility.** We say that a problem is *computationally infeasible* if the probability that a probabilistic polynomially-bounded Turing machine finds a solution is negligible.

**Message Authentication Codes.** A message authentication code (MAC) assures the integrity and the source of a message [MvOV97, p. 325]. More formally, a MAC algorithm is a family of functions $h_k$, parameterized with the secret key $k$, having the following properties:

– Ease of computation: for a known function $h_k$, given a value $k$ and an input $x$, $h_k(x)$ is easy to compute. This result is called *MAC-value* or *MAC*.

– Compression: $h_k$ maps an input $x$ of arbitrary finite bitlength to an output $h_k(x)$ of fixed bitlength.

– Computation-resistance: given a description of function family $h$, for every fixed allowable value of $k$, whereas $k$ is unknown to the adversary, and given zero or more text-MAC pairs $(x_i, h_k(x_i))$, it is computationally infeasible to compute any text-MAC pair $(x, h_k(x))$ for any new input $x \neq x_i$.

As MACs can be implemented very cheaply using standard symmetric-key cryptographic techniques it is reasonable to build authentication into our model. Thus, we assume that all messages are authenticated, which implies that the adversary cannot generate a message on behalf of an honest party or undetectedly modify messages.

### 2.2.2 Executions and the Adversary

As our network is insecure and asynchronous, protocol execution is defined entirely via the adversary. The adversary is a polynomial-time interactive Turing machine that schedules and delivers all messages and corrupts some parties.

The adversary incorporates the entire environment in which the protocol runs. In particular, the adversary incorporates all dishonest parties, the network, and possibly all higher- and lower-level protocols, whereby some restrictions are applied for the behavior of the network and the other protocols.

After the initial setup phase, the adversary repeatedly activates a party with some input message(s) and waits for the party to generate some (authenticated) output message(s). The output is given to the adversary and perhaps indicates to whom these messages should be sent, and the adversary may choose to deliver these messages faithfully at some time. But in general, the adversary chooses to deliver any message it wants, or no message at all; we sometimes impose additional restrictions on the adversary's behavior, however.

The adversary corrupts up to $t$ parties, and receives the initial state of the corrupted parties as produced by the dealer. Like the network, the corrupted parties are completely absorbed into the adversary; in some sense, corrupted parties do not exist, merely their identities that are used by the adversary.

At some points we will refer to *corrupted parties* for simplicity; this should be seen as an abbreviation of *the adversary impersonating a corrupted party.* Uncorrupted parties are called *honest*.

One distinguishes between *static* and *adaptive* corruptions in cryptography: in the *static* corruption model, the adversary must decide whom to corrupt independently of the execution of the system. In the *adaptive* corruption model, the adversary can adaptively choose whom to corrupt as the attack is under way, based on information it has accumulated so far. We adopt a *static* adversary in this work. Although the higher protocols do not require this, we do not know yet how to analyze the underlying cryptographic primitives for adaptive adversaries. The problem with adaptive adversaries is that the simulator in the security proof has to "guess" which parties will be corrupted; the number of guesses needed to be correct is exponential in $n$. We could circumvent this problem by restricting $n$ to be bound by $\log(\kappa)$, in which case exponential in $n$ is still polynomial in $\kappa$. In a real-world setting, this would probably not matter much; the number of parties we have in mind for our protocols is 5 to 20, and cryptographic security parameters are usually much higher than that.

Our formal model leaves control over the application interface for invoking broadcasts and agreement protocols is up to the adversary. The protocol definitions merely state that *if* the adversary invokes the protocol in a certain way—in the same way an intended application would do—*then* the protocol should satisfy some specific conditions. This reflects that applications might be partially influenced by an adversary, which might cause some security problems if this is not allowed in the model. For simplicity, the application program interface is also mapped onto the single messaging interface.

In short, after the initialization phase, nothing but the adversary exists outside of a protocol running on honest parties. The network, the dishonest parties and (possibly) the other protocol layers only exist in the "imagination" of the honest parties.

### 2.2.3 Mind the Gap: Too Few Traitors

In our protocols and proofs, we assume that the adversary corrupts the maximum number of parties it is allowed to corrupt. This section discusses what happens if the adversary is not that capable, i.e., the number $f$ of Byzantine corruptions that actually occur is smaller than $t$, the maximum number of corruptions tolerable.

It might seem strange that there is a problem if the adversary does not use its full capacity, and that the case of $f < t$ is not only a special case of $f = t$; after all, corrupted parties can behave exactly like honest ones. Indeed, there is no fundamental problem with having less traitors; however, to simplify some proofs, we will use some counting arguments that rely on the number of honest parties being limited.

Our proofs use arguments of the form "This means that $x$ honest parties did $A$, but as $y$ honest parties do $B$ and no honest party does both $A$ and $B$, we have a contradiction."

If the number of honest parties is increased, this argument might break. More concretely, some of our proofs use the following property:

"Each two sets of $n - 2t$ honest parties intersect."

Now suppose $f = 0$. This increases the number of honest parties in the system by $t$, while this may suffice to invalidate the threshold. As a numeric example, take $n = 4, t = 1, f = 0$. Now $n - 2t = 2, n = 4$, and there can easily be two sets of $n - 2t = 2$ honest parties that do not intersect.

We use the following trick that allows us to ignore that matter:

Consider an adversary that chooses to corrupt a set $\mathcal{C}$ of $f < t$ parties. Let $\mathcal{H}$ denote the set of $n - f$ honest parties. We choose an arbitrary subset $\mathcal{Q} \subset \mathcal{H}$ of $t - f$ "quasi-corrupted" parties. The idea is that they behave honestly, and for the purposes of the protocol specification and requirements, parties in $\mathcal{Q}$ are considered to be honest.

They do however surrender their secrets to the adversary. Thus, for the purposes of the cryptography that bases on the secrets of individual parties, parties in $\mathcal{Q}$ are considered corrupted. This maintains compatibility with our threshold cryptography.

Also, for all counting arguments, the parties in $\mathcal{Q}$ can be considered as not honest.

Because they follow the protocol, all statements that argue about what honest parties do (i.e., deciding a certain value, terminating the protocol) are still valid for the parties in $\mathcal{Q}$, and all proofs that honest parties satisfy a protocol specification also apply for the parties in $\mathcal{Q}$.

This construction has no influence on our proofs. From the point of view of the honest parties that are not in $\mathcal{Q}$, as well as for all our counting arguments, there are exactly $t$ traitors.

The parties in $\mathcal{Q}$ themselves still follow the protocol, and their internal state is not influenced by the fact that the adversary knows their secrets – they still update their internal state according to the protocol. Thus, all statements about the nature of the output value and about efficiency and deadlock freeness remain unaltered.

## 2.3 Quantitative Measures

This section introduces some quantitative measures that determine the quality of a protocol, along with a new metric, the *protocol statistic*. Protocol statistics are useful in combination with cryptographic proofs, as they blend in well with the polynomial time model and deal better with subtle problems in protocol composition than expectancies do.

### 2.3.1 Quantities

There is a considerable number of quantities that define the quality of a fault-tolerant distributed protocol. It is difficult to derive the practical value of a protocol from these quantities, as the real-world performance is determined by a complex interplay between the environment and the protocol: Ultimately, only an experimental implementation allows precise statements about the real-world behavior of a protocol. Nevertheless, with the quantities introduced here one can obtain good indications about the quality of a fault-tolerant distributed protocol.

**Message and Communication Complexity.** We define the *message complexity* of a protocol as the number of messages generated by all honest parties. This is a random variable that depends on the adversary and $\kappa$. We denote it by $MC(ID)$, where $ID$ identifies a particular protocol instance.

Analogous to message complexity, we define the *communication complexity* of a protocol as the number of all bits of all messages generated by honest parties. We denote it by $CC(ID)$, where $ID$ identifies a particular protocol instance.

As we assume that messages have a polynomial length in $\kappa$, if the message complexity is [probabilistically] polynomially bounded, then so is the communication complexity (and, trivially, vice versa).

**Computational Complexity.** Although most distributed protocols require very little computation, use of public-key cryptography can be expensive enough to have a significant impact on the overall performance. We do not try to model any details here, as this would require specific knowledge about the environment; we simply state that in our protocols here operations involving public-key cryptography are expensive, whereas all other computations are not.

**Work.** A party does *work* if it receives a message and updates its internal state, which in our protocols always goes along with sending some message. One could argue that this way of counting is inappropriate as the adversary can send a message with an invalid syntax or, worse, an invalid public-key signature that entails a real computational effort to verify it, but does not lead to a state update. Our definition ignores these issues for the following reasons:

– For any computer connected to any network (and reading messages from it), it is possible to run a denial-of-service attack by sending nonsense messages. This is unavoidable and independent of what the computer is doing (or supposed to do). It is possible, though, to sort out bogus messages that fail a syntax check or an authentication mechanism efficiently. Thus, if a denial-of-service attack can be dealt with at this level, the system is already quite safe.

– In our model, all messages are authenticated, thus a party can reliably and relatively cheaply identify the sender of a message. If a message from $P_i$ causes $P_j$ a lot of computational work without moving the protocol forward (e.g., it contains an invalid public-key signature), the $P_j$ can identify this as a denial-of-service attack by $P_i$, as only $P_i$ could have authenticated this message Thus, $P_j$ can safely ignore $P_i$ in the future, and the overall additional work the adversary can cause an honest party to do is limited to the effect of $t$ messages.

**Round Complexity.** In a synchronous system, the round complexity of a protocol is the number of synchronous rounds a protocol needs to terminate. In an asynchronous system, these kind of rounds do not exist; rounds are defined by the protocol logic. Following [FLP85], we define the round complexity as follows: Each party $P_i$ has an *incoming message buffer*, in which the adversary can place messages. There are two operations on this buffer, *receive* and *detect*. Let $Q$ be a predicate. The *receive(Q)* operation waits for a set of messages satisfying $Q$ to appear in the incoming buffer, deletes them from the buffer, and returns them. The *detect(Q)* operation does the same, but leaves the messages in the buffer. Note that the *detect* operation is not used in [FLP85]. Some protocols require this operation, as a message in the *incoming buffer* has to be used twice (see Section 8.3.1 and [Bra84] for examples). For a more detailed discussion on accessing the incoming buffer, see Section 2.7.4.

We say that a party executes one *step* if it receives or detects a set of messages, updates its internal state and sends out a finite set of messages. The round complexity of a protocol is the maximum number steps performed by an honest party prior to termination. We also use the different, but related notion of *protocol rounds*, which are defined by the syntax of the protocol: For example, a in protocol of the form **while** $X$ **do** Y; **do** Z; **endwhile**, we count one execution of the while loop as one protocol round, whereas two rounds of communication at needed for the two steps executed.

**Resilience.** The *resilience* of a protocol is the maximum number of corrupted parties the protocol can tolerate. For Byzantine agreement in an asynchronous network, the best resilience possible is $t < n/3$, as shown by Bracha and Toueg [BT85]. This is also the resilience of the protocols presented here. If a more flexible failure model is applied, which allows several different failure types simultaneously, the optimal resilience might be different; this will be discussed in Section 2.5.

## 2.3.2 Metrics: Protocol Statistics and Expectancy

There are two different motivations for measuring quantitative properties of our protocols, e.g., message complexity or number of rounds needed:

First, we use the message complexity to define the computing time available to a party has available (see Section 2.4). Thus, our interest here is that the quantity to measure is polynomially bounded with all but negligible probability.

Second, we want to make statements about the performance of our protocols.

As we work in a randomized world — and even for deterministic protocols, there is some implicit randomization if cryptography is used and thus secret keys can be guessed — we have to involve probabilities in our quantitative measurements.

The standard way to do this is to use expected values, which are quite standard and easy to calculate. For the purpose of defining termination of a protocol, expectation has some problems. First of all, it is difficult to compute using expectations; for example, $E(X) \cdot E(Y) = E(X \cdot Y)$ only holds if $X$ and $Y$ are independent. In our model, all protocol instances are connected through a common adversary, and thus there is a dependency.

Also, expectations do not make statements on the distribution. To be compatible with cryptographic assumptions, we need to argue about worst-case behavior (with all but negligible probability) rather than averages.

To make clean arguments about termination, we introduce a new metric that is defined more along the lines of cryptographic definitions, and that is closed under addition and multiplication.

**Protocol Statistics.** A *protocol statistic* $X$ is a measure for the work performed by the honest parties. In general, $X$ is a family of real-valued, non-negative random variables $\{X_{\mathcal{A}}(\kappa)\}$, parameterized by adversary $\mathcal{A}$ and security parameter $\kappa$, where each $X_{\mathcal{A}}(\kappa)$ is a random variable on the probability space induced by $\mathcal{A}$'s attack on the protocol with security parameter $\kappa$. We only allow statistics that are polynomial in the security parameter (but depending on the adversary); assume w.l.o.g. that every statistic is bounded by the running time of the adversary. The key to defining bounds on quantitative measures (such as the communication complexity) is to bound the statistic *independently* of the adversary, i.e., in such a way that the bound depends only on the particular protocol.

As we consider deterministic and randomized protocols (which may not always terminate after a polynomial number of steps), we introduce two corresponding notions for polynomially bounding a statistic as follows.

**Definition 2.2 (Uniformly Bounded Statistics).** Let $X(\kappa)$ be a statistic in $\kappa$, i.e., a family $\{X(\kappa)\}_\kappa$ of discrete non-negative random variables induced by the adversary. We say that

1. $X(\kappa)$ is *[deterministically] uniformly polynomially bounded (by $T$)* if there exists a polynomial $T(\kappa)$ on the integers such that for all adversaries $\mathcal{A}$, there exists a negligible function $\epsilon_{\mathcal{A}}(\kappa)$ such that for all $\kappa \geq 0$

$$\Pr[X_{\mathcal{A}}(\kappa) > T(\kappa)] \;\leq\; \epsilon_{\mathcal{A}}(\kappa);$$

2. $X(\kappa)$ is *probabilistically uniformly polynomially bounded (by $T$ and $U$)* if there exist polynomials $T(\kappa)$ and $U(\kappa)$ on the integers and a negligible function $\delta(l)$ such that for all adversaries $\mathcal{A}$, there exists a negligible function $\epsilon_{\mathcal{A}}(\kappa)$ such that for all $l \geq 0$ and $\kappa \geq 0$

$$\Pr[X_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa)] \;\leq\; \delta(l) + \epsilon_{\mathcal{A}}(\kappa).$$

The second notion is slightly more complicated than the first one. This is because the protocol might fail with a negligible probability that is independent of the security parameter. Thus, probabilistically uniformly bounded statistic is allowed to exceed the uniform bound with non-negligible probability in the security parameter, but this probability must again be negligible, *independent* of the adversary. To "visualize" the definition, consider a randomized Byzantine agreement protocol. Such a protocol proceeds in rounds, and in each round it terminates with probability $\frac{1}{2}$. Now the polynomial $T$ represents the deterministic cost (e.g., initialization and outputting the result), $U$ represents the cost per round, and $l$ represents the number of rounds — for increasing values of $l$, the probability that round $l$ is reached has to vanish faster than any polynomial.

We now have to show that protocol statistics are closed under modular protocol composition, i.e., combining uniformly bounded protocols leads to another uniformly bounded protocol. To this end, we show that uniformly bounded statistics are closed under addition and multiplication.

**Lemma 2.1.** *If $X(\kappa)$ and $X'(\kappa)$ are [probabilistically] polynomially bounded statistics in $\kappa$, then $(X(\kappa) + X'(\kappa))$ is also a [probabilistically] polynomially bounded statistic in $\kappa$.*

*Proof.* We will only show the probabilistic case here; the non-probabilistic proof comprises the same calculation with fewer parameters.

We know that there exists polynomials $T(\kappa), T'(\kappa), U(\kappa), U'(\kappa)$ and negligible functions $\delta(l), \delta'(l), \epsilon_{\mathcal{A}}(\kappa), \epsilon'_{\mathcal{A}}(\kappa)$ such that for all $l \geq 0$ and $\kappa \geq 0$

$$\Pr[X_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa)] \ \leq \ \delta(l) + \epsilon_{\mathcal{A}}(\kappa)$$

and

$$\Pr[X'_{\mathcal{A}}(\kappa) > lU'(\kappa) + T'(\kappa)] \ \leq \ \delta'(l) + \epsilon'_{\mathcal{A}}(\kappa).$$

Now,

$$\begin{aligned}
&\Pr[X_{\mathcal{A}}(\kappa) + X'_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa) + lU'(\kappa) + T'(\kappa)] \\
&\leq Pr[X_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa)] \vee X'_{\mathcal{A}}(\kappa) > lU'(\kappa) + T'(\kappa)] \\
&\leq \Pr[X_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa)] + \Pr[X'_{\mathcal{A}}(\kappa) > lU'(\kappa) + T'(\kappa)] \\
&\leq \delta(l) + \epsilon_{\mathcal{A}}(\kappa) + \delta'(l) + \epsilon'_{\mathcal{A}}(\kappa).
\end{aligned}$$

As $\delta(l) + \delta'(l)$ and $\epsilon_{\mathcal{A}}(\kappa) + \epsilon'_{\mathcal{A}}(\kappa)$ are sums of negligible functions and thus also negligible, the statistic $(X(\kappa) + X'(\kappa))$ satisfies the definition for a [probabilistically] polynomially bounded statistic in $\kappa$. □

**Lemma 2.2.** *If $X_{\mathcal{A}}(\kappa)$ and $X'_{\mathcal{A}}(\kappa)$ are [probabilistically] uniformly polynomially bounded statistics in $\kappa$, then $(X_{\mathcal{A}}(\kappa) \cdot X'_{\mathcal{A}}(\kappa))$ is also a [probabilistically] polynomially bounded statistic in $\kappa$.*

*Proof.* Again we will only show the probabilistic case here.

As $X_{\mathcal{A}}(\kappa)$ and $X'_{\mathcal{A}}(\kappa)$ are uniformly bounded, there exists polynomials $T'(\kappa), T''(\kappa), U'(\kappa), U''(\kappa)$ and negligible functions $\delta(l'), \delta''(l)$, $\epsilon'_{\mathcal{A}}(\kappa), \epsilon''_{\mathcal{A}}(\kappa)$ such that for all $l \geq 0$ and $\kappa \geq 0$

$$\Pr[X_{\mathcal{A}}(\kappa) > lU'(\kappa) + T'(\kappa)] \ \leq \ \delta'(l) + \epsilon'_{\mathcal{A}}(\kappa)$$

and

$$\Pr[X'_{\mathcal{A}}(\kappa) > lU''(\kappa) + T''(\kappa)] \ \leq \ \delta''(l) + \epsilon''_{\mathcal{A}}(\kappa).$$

We first simplify the formulas by defining polynomials and negligible functions that can be used for both $X_{\mathcal{A}}(\kappa)$ and $X'_{\mathcal{A}}(\kappa)$ . Let

$$\begin{aligned}
T(\kappa) &= T'(\kappa) + T''(\kappa) \\
U(\kappa) &= U'(\kappa) + U''(\kappa) \\
\epsilon(\kappa) &= \max\{\epsilon'(\kappa), \epsilon''(\kappa)\} \\
\delta(l) &= \max\{\delta'(l), \delta''(l)\}
\end{aligned}$$

With these definitions, we have

$$\Pr[X_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa)] \leq \Pr[X_{\mathcal{A}}(\kappa) > lU'(\kappa) + T'(\kappa)] \leq \delta'(l) + \epsilon'_{\mathcal{A}}(\kappa) \leq \delta(l) + \epsilon_{\mathcal{A}}(\kappa)$$

and

$$\Pr[X'_{\mathcal{A}}(\kappa) > lU(\kappa) + T(\kappa)] \leq \Pr[X'_{\mathcal{A}}(\kappa) > lU''(\kappa) + T''(\kappa)] \leq \delta''(l) + \epsilon''_{\mathcal{A}}(\kappa) \leq \delta(l) + \epsilon_{\mathcal{A}}(\kappa).$$

Next, we show the following equation:

$$\Pr[X_{\mathcal{A}}(\kappa) \cdot X'_{\mathcal{A}}(\kappa) > (lU(\kappa) + T(\kappa))^2] < 2 \cdot (\delta(l) + \epsilon_{\mathcal{A}}(\kappa)) \tag{2.1}$$

**Corollary 2.3.** *Provided* $\Pr[X_{\mathcal{A}}(\kappa) > a] \leq p$ *and* $\Pr[X'_{\mathcal{A}}(\kappa) > a] \leq p$*, the following holds:*

$$\Pr[X_{\mathcal{A}}(\kappa) \cdot X'_{\mathcal{A}}(\kappa) > a^2] < 2 \cdot p$$

*Proof.* It is straightforward to see that

$$X_{\mathcal{A}}(\kappa) X'_{\mathcal{A}}(\kappa) > a^2 \Rightarrow (X_{\mathcal{A}}(\kappa) > a) \vee (X'_{\mathcal{A}}(\kappa) > a),$$

i.e., if $\Pr[X_{\mathcal{A}}(\kappa) \cdot X'_{\mathcal{A}}(\kappa) > a^2]$ is larger than $2p$, then so is $\Pr[X_{\mathcal{A}}(\kappa) > a) \vee (X'_{\mathcal{A}}(\kappa) > a)]$.

The probability $\Pr[Y \vee Z]$ is bound by $\Pr[Y] + \Pr[Z]$, which implies the corollary. □

By setting $a = lU(\kappa) + T(\kappa)$, and $p = \delta(l) + \epsilon_{\mathcal{A}}(\kappa)$, the corrolary immediately implies equation 2.1.

It is left to show that the Lemma follows from equation 2.1. By multiplying out the squares, we get

$$\Pr[X_{\mathcal{A}}(\kappa) \cdot X'_{\mathcal{A}}(\kappa) > l^2 U^2(\kappa) + 2lU(\kappa)T(\kappa) + T(\kappa)^2] < 2 \cdot (\delta(l) + \epsilon_{\mathcal{A}}(\kappa)) \tag{2.2}$$

With $\hat{U} = U^2(\kappa) + 2U(\kappa)$, $\hat{T} = T^2(\kappa)$ , $\delta(\hat{(l)}) = 2\delta(\sqrt{l})$ and $\epsilon(\hat{\kappa}) = 2\epsilon_{\mathcal{A}}(\kappa)$, we finally get

$$\Pr[X_{\mathcal{A}}(\kappa) \cdot X'_{\mathcal{A}}(\kappa) > l\hat{U}(\kappa)\hat{T}(\kappa)] < \hat{\delta}(l) + \cdot\hat{\epsilon}(\kappa).$$

□

**Protocol Statistic vs. Expectancy.** Protocol statistics are a useful means to discuss about guaranteed worst-case behavior of a protocol, and integrate nicely into cryptographic definitions. Furthermore, as they are closed under addition and multiplication, they can be used for combined protocols — if two protocols are polynomially bounded in $\kappa$, any modular composition of these protocols is polynomially bounded in $\kappa$ as well. For these reasons, we will use protocol statistics to define liveness properties in the computational model (see Section 2.4).

If we want to argue about the performance of a protocol, one has to keep in mind that protocol statistics and expectation measure two different things. A protocol statistic measures the worst-case running time of a protocol. The probability that the protocol runs longer than the value measured here is negligible, i.e., it lies in the same dimension as the probability that the cryptographic keys are broken or that everybody who cares about the outcome of the protocol is struck by lightning. Thus, this value is (as far as this is possible in an asynchronous network) a "real-time guarantee". Another issue is that protocol statistics are a rather rough metric. The way we defined them, they can only measure polynomials, and constants play no role. We could extend them to arbitrary functions, but making a tight analysis with this metric is sill difficult.

An expectation measures how long a protocol run takes on average (but still assuming a worst case adversary). A good measure to argue about is how long it will take to finish a large number of protocols, while individual instances might need significantly longer. As opposed to protocol statistics, this allows a very fine granuallity. However, to compute expectations of composed protocols, the exact way protocols are composed plays an important role, thus some manual analysis is difficult to avoid.

An example for the difference is our protocol for Byzantine agreement (see Chapter 4.3). The expected number of rounds this protocol needs is constant, but the probability that it does not terminate in constant time is not negligible; the smallest polynomial that binds the number of rounds needed is $T(x) = ax + b$. As another example, consider the parallel agreement (see Section 4.4.4), which combines $k$ protocols with an expected constant number of rounds. For reasonable choices of $k$, this protocol is polynomially bounded, provided the single Byzantine agreement protocol also is, by with the same polynomial.

The expected running time of the combined protocol is $O(\log k)$ rounds; however, obtaining this result is not possible without actually looking at the protocol manually.

As protocols statistics define a worst-case (with all but negligible probability) metric, and expectations are defined over the average, an expectation is bound by the corresponding protocol statistic.

**Lemma 2.4.** *Suppose* $X(\kappa)$ *is a probabilistically polynomially bounded statistic of a multi-party protocol, with polynomials* $T(\kappa)$ *and* $U(\kappa) \neq 0$*. Then the expected value of* $X(\kappa)$ *is bounded by* $c\big(U(\kappa) + T(\kappa)\big) + \epsilon'(\kappa)$*,* *where $c$ is a constant that is independent of the adversary and $\epsilon'$ is a negligible function that depends on the adversary.*

*Proof.* As all parties and the adversary are bounded by polynomials is $\kappa$, the statistic is, by definition, bounded by some polynomial $q(\kappa)$ in the security parameter, depending on the adversary, but never exceeding its running time; thus the random variable $X(\kappa)$ exceeds $q(\kappa)$ with probability zero.

Set $X'(\kappa) = (X(\kappa) - T(\kappa))/U(\kappa)$; it follows $X'(\kappa) \leq q'(\kappa)$ for some polynomial $q'(\kappa)$. Because $X(\kappa)$ is probabilistically polynomially bounded, we know that $\Pr[X'(\kappa) > l] \leq \delta(l) + \epsilon(\kappa)$ for all $l > l_0$ and all sufficiently large $\kappa$, where $l_0$ is independent of the adversary. Together with $\mathrm{E}[Y] \leq \sum_{l \geq 0} \Pr[Y > l]$ for any non-negative discrete random variable $Y$, it follows

$$\mathrm{E}[X'(\kappa)] \;\leq\; \sum_{l \geq 0} \Pr[X'(\kappa) > l] \;=\; \sum_{l=0}^{q'(\kappa)} \Pr[X'(\kappa) > l] \;\leq\; \sum_{l=0}^{q'(\kappa)} \big(\delta(l) + \epsilon(\kappa)\big).$$

Now fix $\delta$ to a function whose sum converges to a constant, say, $\delta(l) = l^{-2}$. We have

$$\sum_{l=0}^{q'(\kappa)} \big(\delta(l) + \epsilon(\kappa)\big) \;\leq\; l_0 + \sum_{l=l_0+1}^{q'(\kappa)} l^{-2} + q'(\kappa)\epsilon(\kappa) \;\leq\; l_0 + c_0 + q'(\kappa)\epsilon(\kappa)$$

for constants $c_0$ and $l_0$, which are independent of the adversary. Because $\epsilon$ is negligible and by the linearity of expectation, this implies that $\mathrm{E}[X(\kappa)] = c_0 l_0 (U(\kappa) + T(\kappa)) + \epsilon'(\kappa)$ for a negligible $\epsilon'$. □

## 2.4 I/O Automata, Liveness and Eventuality.

In the distributed-computing community, the I/O automata model of Lynch and Tuttle [LT87, LT89, Lyn96, AW98] seems to be the most established model for asynchronous protocols, and it has also been extended to allow for modeling of randomized protocols. As we want to solve problems of the distributed systems world, we orient our model along these lines instead to modifying cryptographic models to our needs.

Modeling the parties as I/O Automata has the advantage that several automata can easily be combined; the entire system then can be seen as one big automaton comprising the parties as components. In cryptographic applications, it is more common to model parties as (probabilistic) interactive Turing machines [LP81, Gol01a]. This has the advantage that Turing machines are easier to handle for a complexity analysis, but it is harder to combine several Turing machines into one as the tapes tend to entangle [LP81, p.180]. An I/O automaton according to [Lyn96] consists of a (possibly infinite) set of states and a set of state transitions, called actions. The actions are classified as either *input*, *output* or *internal*. Output and internal actions are under the control of the automaton, while input actions are triggered by the environment (which, in our notation, is the adversary).

On defining properties of such a distributed system, one traditionally distinguishes between two types of properties: *safety* and *liveness*. Informally, safety properties have the form "No bad things happen", while liveness properties have the form "eventually, something good happens". Another formulation is that safety deals with the finite behavior of the system, while liveness specifies the infinite behavior.

**Definition 2.3.** A *trace property* $T_P$ consists of a signature $\mathrm{sig}(T_P)$ and a set $T$ of (finite or infinite) sequences of actions in $\mathrm{acts}(sig(T_P))$.

$T_P$ is a *safety property* if $T$ is a nonempty set of traces such that

- $T$ is *prefix closed*, i.e., if $\beta \in T$ and $\beta'$ is a finite prefix of $\beta$, then $\beta' \in T$
- $T$ us *limit closed*, i.e., if $\beta_1, \beta_2, \ldots$ is an infinite sequence of finite sequence in traces in $T$, and for each $i, \beta_i$ is a prefix of $\beta_{i+1}$, then the unique sequence $\beta$ that is the limit of the $\beta_i$ under the successive extension ordering is also in $T$.

$T_P$ is a *liveness property* if $T$ is a set of traces such that every finite sequence of acts(sig($T_P$)) has an extension in $T$.

In this definition, an *execution* is a sequences of states and actions generated by the run of an I/O automaton, a *trace* is an An execution that is reduced to input– and output–actions and the *signature* $S$ of an automaton is a triple consisting of three disjoint sets of actions, the *input actions*, the *output actions*, and the *internal actions.* The set acts($S$) is the set of all actions of $S$[1]

### 2.4.1 Liveness in a computationally bounded World

The I/O automata model assumes that cryptography — if used at all — is treated as an idealized black box; authentication and digital signatures can only be used as long as the underlying assumptions, namely the computational bounds, are left out.

If we try to combine a computationally bounded model with I/O automata, some problems arise. Some are deeply burrowed in the model and of little consequence to protocol designers. Others, especially the lack of infinite runs which are used to define liveness properties, require some work to even be able to define the classical problems in the computational model. We will now point out the greatest difficulties and describe a way to (relatively) generally transform a problem from the I/O Automata model into the computationally bounded setting.

**Automata.** In the cryptographic world, automata play little to no role. Parties are modeled either as Turing machines or as circuits. As cryptography depends much more on subtleties in the model than distributed systems, we use the view established in the cryptography-community.

**Infinity.** A lot of very nice and elegant definitions in the I/O automata model base on infinite traces. In a computationally bounded model, we have to eliminate all traces of infinity from the model and definitions.

**Time.** The I/O automata model is somewhat implicit about time. Some scheduler is distributing computation time, and all we know is that some fairness condition eventually holds. To argue about computational bounds, we need to be more explicit and precise in defining computation time.

**Modeling Asynchronous Time.** We model time by counting messages sent from honest parties, i.e., one message generated by an honest party corresponds to one clock tick. Thus, the scheduler that schedules computing time is included in the scheduler that delivers messages, and the computation time available is bound by the number of messages generated by honest parties. Thus, a typical liveness condition of the form

$\mathcal{L}$ : Eventually, event $X$ happens.

is modified as

$\mathcal{L}_1$ : If infinitely many messages are sent by honest parties, event $X$ will happen.

A consequence of this model is that time just stops if no honest party sends messages; thus, any protocol that does nothing at all will satisfy condition $\mathcal{L}_1$. We need an additional property to outrule this behavior:

$\mathcal{L}_2$ : If event $X$ has not happened, then there is at least one message sent between two honest parties that has not been delivered yet.

We call the property $\mathcal{L}_2$ the *deadlock freeness* property; together, $\mathcal{L}_1$ and $\mathcal{L}_2$ replace the liveness condition in the classical form.

[1] At this point I owe the reader an apology for using several different meanings for some terms – in combining definitions of two fields this kind of collision was bound to happen. I choose not to change the definitions of the standard literature as most terms defined in this section are not needed again in this thesis – besides an intuition about *liveness* and *safety*, one can safely forget the definitions made here at the end of this section

**Binding Asynchronous Time.** It is now left to restrict the amount of work a protocol is allowed to do; the changes made so far only introduce a more explicit notion of measuring time in a fully asynchronous network, but they do not enforce polynomial bounds. Thus, the property $\mathcal{L}_1$ again needs to be modified. The new property $\mathcal{L}_1'$ introduces a [probabilistic] upper bound on the number of messages sent between honest parties.

> $\mathcal{L}_1'$: The communication complexity caused before event $X$ occurs is [probabilistically] polynomially bounded.

We call the property $\mathcal{L}_1'$ the *efficiency* property. By replacing liveness properties by the corresponding deadlock freeness and efficiency properties, definitions can easily be transfered from the conventional computationally unbounded model [LT87, LT89, Lyn96] into our computationally bounded model.

Note that efficiency in our sense is a theoretical measure needed for security proofs. An efficient protocol in our definition does not need to perform well in a real implementation. If we talk about the latter aspect, we will use the term *performance*.

Our definition of *deadlock freeness*, together with the new liveness condition $\mathcal{L}_1'$, implies that an adversary *could* quickly make event $X$ happen [with overwhelming probability] (relative to cryptographic assumptions), by delivering a (fixed) polynomially bounded number of messages; however, we do not *force* the adversary to do so—if no messages are delivered, the liveness condition simply does not require the protocol to do anything.

The definitions so far assumed that protocol instances do not interact. This is true for most protocols in this thesis. However, for some protocols – like atomic broadcast – different protocol instances must interact, as the purpose of those protocol is to coordinate different protocol instances.

Our definitions do not directly work in this setting, as implementation messages can not be assigned to protocol instances in an easy way, and we have to prevent the adversary from starting many instances (thus communication between honest parties) without ever letting any instance finish. A detailed discussion of this point is made in Chapter 7, when this problem first occurs.

## 2.5 A Classification of Failures

The failure model used in this thesis is a worst–case model in that all corrupted parties and the network become part of the adversary and behave as malicious as possible. Although this adversary may be unrealistically strong, it is not possible to define in which way the adversary in our model is stronger than the worst case a system would encounter in the real world.

The first projects dealing with software implemented fault tolerance argued along the same line, assuming the worst case for everything that couldn't be exactly specified otherwise:

> "[...] we cannot allow a system failure to be caused by any condition whose probability cannot be quantified, regardless of how implausible that condition may seem. This means our [...] procedure must be reliable in the face of the worst possible behavior of the failing component, even though that behavior may seem unrealistically malicious." [WLG$^+$78]

Once the full complexity of the problem had been understood, weaker failure models where introduced; this went so far that the malicious (Byzantine) setting was mostly considered unrealistic by the community:

> "Byzantine Elm Disease (a joke on Dutch Elm Disease, real disease of trees) was invented to describe some of the Byzantine work in the 70s and 80s that looked at increasingly complicated and unrealistic aspects of the problem. " [Her00]

By now, a variety of failure models has emerged. Essentially, there are two major classifications of failures in the literature; the classification by effect (e.g. a crash versus a malicious failure), and the classification by origin.

There are various means to to reach fault tolerance are for example to identify corruptions, and to exclude corrupted parties or roll back actions. This approach requires a detailed analysis of the nature of the failure that occurred. The approach to reach fault tolerance presented in this thesis is failure masking, i.e., the protocols work *in spite of failures*; they do not try to identify failures, and it might well be that a corruption remains undetected forever. Therefore, we do not care about the origin of the failure; all that is important for us is to tolerate the its effect. Therefore, we do not care about the origin of the failure; for an excellent survey on that topic, see [Gär98].

It might however be interesting to be more detailed about the effect of a failure, as a system can tolerate a greater number of weaker failures. For example, if we allow crash failures only, the optimal resilience of in $t < \frac{n}{2}$ instead of $t < \frac{n}{3}$ in the Byzantine case. Depending on the environment, it might therefore be advantageous to differ between failure types.

### 2.5.1 Failure Types

There is a vast number of different failures in the literature. For this work, we distinguish between the following failure types:

**Crash Failures.** A crashed party simply stops operating at some point. The adversary can cause this to happen at any time, even while the corresponding party is in the middle of sending a message.

Note that this contradicts the assumption that the step of a party receiving a message, evaluating it and sending out subsequent messages is atomic — the adversary may crash a party at any time during that step. We can simulate this effect in our atomic model by allowing the adversary to delete messages from the last broadcast a crashing party sent. This results in a slightly stronger adversary, as the adversary can see a message, then decide that the party crashes and delete the message, simulating that the message has never been sent. However, in our protocols, this does not provide the adversary with any significant advantage.

If a party crashes while sending a message, the worst that can happen is that the message is truncated. It is easy to ensure that this results in a syntactically incorrect message. Furthermore, if message authentication is used, the truncated message will almost certainly not authenticate correctly. As the receiver collects only syntactically correct messages, such a message is simply ignored.

There is no means for other parties to detect such a crash, and a crashed party will remain crashed forever.

We consider a crashing party as "quasi-corrupted"; it counts as corrupted for *liveness conditions* such as termination, but not for *safety conditions* such as agreement. If we allow crash failures in the consensus problem, and a party crashes *after* deciding, the decision value still must satisfy the agreement property.

The distinction between crash and Byzantine failures increases the total number of tolerable failures; if $b$ is the number of Byzantine failures, and $c$ is the number of crash failures, we will show that Byzantine agreement can be solved as long as $3b + 2c < n$; this bound is optimal (a generalization of this will be shown in Section 4.6.3). As a crashing party does not expose any secrets to the adversary, we do not require static failures for this failure type — the adversary is free to choose the set of crashing parties at any time.

**Crash Recovery.** It is possible that a party crashes and later recovers and restarts the protocol. If we assume persistent storage, i.e., the party does not lose any state information, this behavior only slows the corresponding party down and causes it to miss some messages; in this case, this type of failure corresponds to linkfailures introduced below.

If the party can lose state information, recovery is somewhat more complicated:

– The party cannot merely rejoin into the protocol as if nothing has happened, because this might imply a violation of protocol invariants. The party may, for example, issue several different votes in a voting protocol. From this point of view, we have to consider this party malicious (Byzantine).

– We cannot treat this party like a "normal" malicious party, either; while we do not care which results the dishonest parties obtain from a protocol (e.g., if they disagree after an agreement protocol), the crashed and recovered party should achieve sensible results: in the agreement protocol, for example, it should either agree with the honest parties or output an error message.

Our protocols can deal with this kind of failure, but only in a rather crude way. A recovered party counts towards the number of corrupted parties for all protocol instances it started before the crash that have not yet been terminated by all honest parties, but it will count as honest as far as liveness and safety conditions are concerned, i.e., it still has to terminate and output the correct result.

It is possible to make it count only towards the number of crashed parties by "passifying" the corresponding party, i.e., a recovered party follows the protocols, but does not send out any messages unless it is sure it did not process the corresponding protocol instance before. The disadvantage of this is that the party has to find out when it should be reactivated and send messages again. This is rather difficult, as it requires determining all protocol instances for which messages from this party are still floating around. In practice, this would be done using some global time, and we will not try to find an asynchronous solution to this problem here.

**Passive Byzantine ("curious but honest").** A party that fails passively surrenders all its secrets to the adversary (and keeps on doing so if it obtains new secrets), but otherwise follows the protocols.

In our model, if a party surrenders its authentication keys to the adversary, the adversary can impersonate this party completely. Moreover, the adversary can delay all messages sent by this party until everything is finished, which has the same effect as if the party was corrupt in the first place.

One can assume a more refined model, in which two levels of secrecy are defined: cryptographic keys that are inaccessible even for the party itself (e.g., if they are stored in a cryptographic coprocessor), and other secrets that occur during the run of the protocol which are accessible and thus known to the adversary.

This model makes most sense if information-theoretically secure protocols are analyzed, as for example the agreement protocol by Canetti and Rabin [CR93].

In our protocols, all the secrets our parties have are cryptographic keys which could be stored securely during initialization (the only exception is the secure causal broadcast protocol in Section 8.4, but here the problem does not make sense for a party that leaks secrets to the adversary).

Another model of "honest but curious" is that the corresponding parties act like honest parties during the run of the protocol, but afterwards pool their information to violate privacy conditions. However, our protocols do not deal with privacy problems; the only protocol that deals with secrets other than the cryptographic keys is the secure causal broadcast, and there the secrets only need to be protected until the corresponding broadcast has been delivered, i.e., after the termination of the protocol there is no privacy to violate.

**Timing / Performance Failures.** A *timing* or *performance* failure occurs if a party does not perform a task in the time it is supposed to. As this failure can by definition only occur in timed models, we will not consider it for our protocols.

**Byzantine Failures.** If a Byzantine failure occurs, the adversary has taken full control over the corresponding machine. All secrets this machine has are handed over to the adversary, who now controls its entire behavior.

**Link Failures.** A link failure occurs if a message between honest parties is modified or lost.

As we have inexpensive message authentication, we can reduce link failures to lost messages: With all but negligible probability, a modification of a message by the adversary will be detected and thus the message is invalid (i.e., lost).

The problem here is, to some extent, related to the crash-recovery problem — an honest party behaves as if it were dishonest. While it still has to come to a valid output of the protocol, it does not participate actively in it.

Sayeed et al. [SAAAA95] introduced a protocol that can tolerate up to $\lfloor \frac{n-2}{2} \rfloor$ Byzantine links in a asynchronous system with $n$ honest processes. They also show that this is the maximum number of Byzantine links tolerable. Gong et al. [GLR95] also analyze a hybrid model with Byzantine, crash and linkfailures, but only for a synchronous communication model. This kind of failure will be the subject of Section 4.5.

**Exclusion failures.** If the honest parties identify a corrupted party in some way, they might want to exclude it from further participation. The problem here is that the corresponding party still has the cryptographic keys and thus, even though the thresholds change, that party still has some effect. One solution would be to use proactive protocols [CGHN97]. From time to time, parties can be "repaired", i.e., the corruption is removed (for example by a clean reboot from a CD-ROM), and potentially exposed secrets are rendered useless by computing new secrets. This repairing work is not restricted to corrupted parties; one appealing issue is that we can repair every party in the system without having to know who is corrupted. These protocols are not cheap, however, and it might be worthwhile living with an identified traitor for a while before repairing corruptions.

### 2.5.2 Adversary Structures

Another generalization is to go away from numeric thresholds and rather work with sets of parties. Instead of specifying the number of corruptions tolerated, an *adversary structure* explicitly specifies all sets of parties (the *coalitions*) which may be corrupted. Numeric conditions (for example, that the number of corrupt parties is less than a third of the total number of parties) are then replaced by conditions on the allowed coalition (e.g., that no union of three of those sets is the full set of parties).

Note that this does not allow us to tolerate more corruptions, but we can shift the balance, i.e., we can tolerate much more than a third of corruptions (if the right parties are corrupted), but it the same time much less (if the wrong parties are corrupted).

These structures have first been defined for secret sharing schemes [ISN87, BL90, Sim88], where an *access structure* $\Gamma$ defines which sets (coalitions) of parties are allowed to reconstruct the secret.

The first usage of adversary structures beyond secret sharing goes back to Fitzi, Hirt and Maurer [HM97]. Initially, they consider active and passive failures separately, whereas the corresponding adversary structures are as follows:

**Definition 2.4.** Call $\mathcal{P}$ the set of all parties involved in the protocol, i.e., $\mathcal{P} = \{P_1, \ldots, P_n\}$. A set $\mathcal{G} \subseteq 2^{\mathcal{P}}$ of parties is called *monotone* if $\forall G \in \mathcal{G}, G' : G' \subseteq G \Rightarrow G' \in \mathcal{G}$, i.e., if $G$ is a member of $\mathcal{G}$, then all supersets of $G$ are members of $\mathcal{G}$.

An adversary structure now defines all sets $c \subset \mathcal{P}$ of parties that we allow to be corrupted.

**Definition 2.5.** An adversary structure $\mathcal{Z}$ is a monotone set of subsets of $\mathcal{P}$, the set of all parties. It satisfies the predicates

$Q^2(\mathcal{P}, \mathcal{Z})$ if no two coalitions $A, B \in \mathcal{Z}$ satisfy $A \cup B = \mathcal{P}$.

$Q^3(\mathcal{P}, \mathcal{Z})$ if no three coalitions $A, B, C \in \mathcal{Z}$ satisfy $A \cup B \cup C = \mathcal{P}$.

We can show that $Q^2$ is a necessary condition to solve agreement in the crash failure model, whereas $Q^3$ is necessary in the Byzantine model. A proof of a more general theorem that encompasses these statements is given in Section 4.6.3.

Later, the authors consider hybrid failures, i.e., parties may fail in different ways; this also leads to more complex adversary structures that combine $Q^2$ and $Q^3$ into one structure.

While Fitzi, Maurer and Hirt concentrate on passive and active failures (in a later paper they start to add crash failures [FHM98]) in the synchronous model, we consider only crash and Byzantine failures. The reason is that passive failures as defined in [HM97] assume parties that follow the protocol, but later pool all the data they collected to violate privacy. For our protocols, this does not make much sense; the only protocol that has some sort of private data (outside of cryptographic keys) is secure causal broadcast (see Section 8.4), and even there the data becomes publicly available as soon as the corresponding broadcast is delivered.

### Adversary Structures in the Hybrid Crash/Byzantine Failure Model

To define an *adversary structure* $\mathcal{Z}$, one has to define every coalition of parties whose corruption the system should tolerate, e.g., a coalition of all machines with the same operating system. The set of all these sets then is the adversary structure.

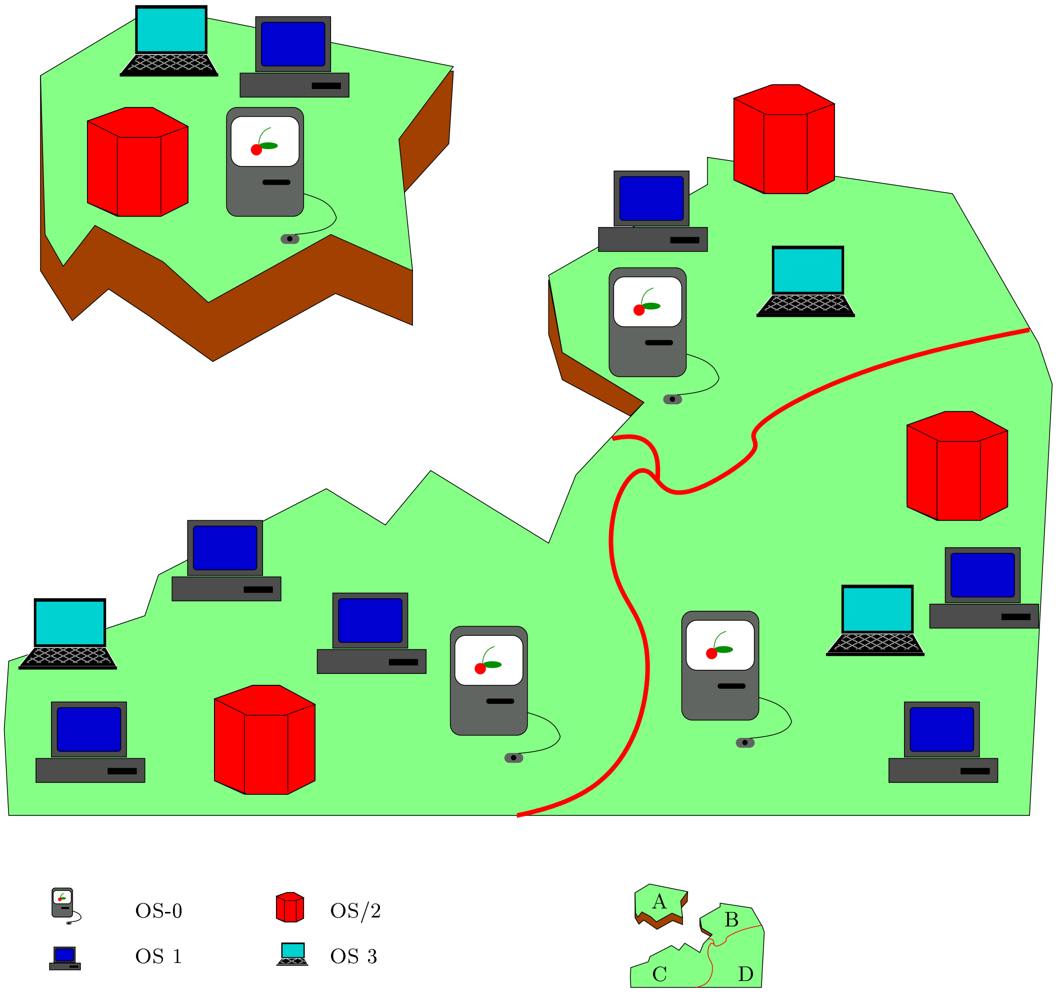


Figure 2.1: An inhomogeneous distributed system

Figure 2.1 shows one example of 19 sites distributed in a structured way. They are distinguished by two features: the operating system (OS-0 to OS-3) and the location (Countries A to D). In conventional $t$-out-of-$n$ structures with $3t < n$, any set of six failing parties can be tolerated. A possible adversary structure is to tolerate the simultaneous failure of one operating system and one location. In our example, this would satisfy $Q^3$. Then the number of tolerable failures can be as high as 10 parties (e.g., failure of all parties that are located in country D or run operating system OS 1), or only four if the corruptions are well distributed, i.e., four computers covering all countries and all operating systems.

If one puts more restrictions on the number of Byzantine failures, additional crash failures can be tolerated in such a way that the total number of tolerable failures increases.

A similar approach has been adopted in [FHM99], where the two types of failure are *active Byzantine* and *passive Byzantine*.

The coalitions defined by the adversary structure $\mathcal{Z}$ now consist of pairs $(\mathcal{B}, \mathcal{C})$, where $\mathcal{B}$ is the set of malicious parties and $\mathcal{C}$ is the set of crashing parties.

**Definition 2.6.** Let $\mathcal{P}$ be the set of all parties. A (hybrid) adversary structure $\mathcal{Z}$ is a pair of monotone set of subsets of $\mathcal{P}$

We say that the (hybrid) adversary structure $\mathcal{Z} = \{\mathcal{B}, \mathcal{C}\}$ satisfies the predicate $Q^{(3,2)}(\mathcal{P}, \mathcal{Z})$ if no combination of two coalitions in $\mathcal{Z}$ and of the Byzantine parties in one coalition in $\mathcal{Z}$ covers $\mathcal{P}$. More formally,

$$Q^{(3,2)}(\mathcal{P}, \mathcal{Z}) \Leftrightarrow \forall (B_1, C_1), (B_2, C_2), (B_3, C_3) \in \mathcal{Z} : B_1 \cup B_2 \cup B_3 \cup C_1 \cup C_2 \neq \mathcal{P}.$$

In particular, if no crash failures are allowed, then $\mathcal{Z}$ satisfies $Q^3(\mathcal{P}, \mathcal{Z})$; if no Byzantine failures are allowed, then $\mathcal{Z}$ satisfies $Q^2(\mathcal{P}, \mathcal{Z})$. Thus, both $Q^3(\mathcal{P}, \mathcal{Z})$ and $Q^2(\mathcal{P}, \mathcal{Z})$ are special cases of $Q^{(3,2)}(\mathcal{P}, \mathcal{Z})$.

We will from here on use the term *adversary structure* for both hybrid and normal adversary structures; the precise definition will be clear from the context.

## 2.6 Models between Synchrony and Asynchrony

Whereas assuming a fully synchronous system may be too optimistic for a real-world system, the fully asynchronous view of the world may be a bit too far off on the pessimistic side. In this section we discuss the use of timing information in a more pragmatic way, trying to model a realistic system without being too pessimistic or optimistic. There are various such models in the literature; the novelty of our approach is that we base only the performance of our protocols on timing assumptions, while remaining fully pessimistic as far as liveness and safety of our protocols are concerned.

### 2.6.1 Timed Models in the Literature

In 1983, Fischer, Lynch and Paterson [FLP85] showed that agreement in a fully asynchronous network cannot be solved by a deterministic protocol. Since this “impossibility” result, various attempts have been made to define a timing model that offers enough timing to deterministically solve Byzantine agreement, but still reflects the behavior of a realistic network.

In a refinement of [FLP85], Dolev et al. [DDS87] differentiate between several parameters of a synchronous system. They analyze which synchrony properties are needed to solve consensus in the presence of crash-failures. Although it has not explicitly been worked out, the basic results also apply for the Byzantine case. They identify five critical parameters:

*Processor Synchrony:* Are the processors synchronous or asynchronous?

*Communication Synchrony:* Is communication synchronous or asynchronous?

*Message Order:* Is the (total) message order synchronous or asynchronous? A synchronous message order implies that if party $P_i$ sends a $m_1$ message at real time $r_1$, and $P_j$ sends a message $m_2$ at real time $r_2$, then a party that receives these messages receives $m_1$ before $m_2$.

*Transmission Mechanism:* Are the messages delivered by broadcast or by point-to-point transmission?

*Atomic Receive/Send:* Is receiving and sending one atomic action?

The bounded processor speed and message-delivery time mean that both lower and upper bounds exist. An upper bound is not sufficient, as it would still allow an arbitrary (relative) timing unless real time clocks are assumed.

Considering these parameters, one can analyze which of them are necessary to reach deterministic agreement. It is shown that four cases exist in which sufficient synchrony is provided:

1. synchronous processors and synchronous communication,
2. synchronous processors and ordered messages,

3. broadcast transmission and ordered messages,
4. synchronous communication, broadcast transmission, and atomic receive/send.

There are two interesting observations to make here:

It is possible to simulate broadcast transmission on an asynchronous network. Furthermore, ordering messages essentially is a complex form of agreement, i.e., agreement on the order of messages. Thus, once an agreement protocol exists, the requirements for the third case can be offered by a software implementation. This is what most implementations of dependable middleware do [BJ87, Rei95, Hay98, vRBG$^+$95, MMSA94]; given a (more or less) asynchronous network, they offer broadcast transmission and ordered messages on which application protocols can be built relatively comfortably. It is one of the main goals of this thesis to offer this kind of service building on randomized algorithms in a fully asynchronous network.

If one wants to model a not-really-but-still-somewhat-synchronous system, the most reasonable model is to start with synchronous processors and synchronous communication. Some work has been done working with partially ordered messages [PS98] or incomplete broadcasts [FM00], but the overwhelming majority of the literature uses a timing model between asynchronous and synchronous.

Although our main approach is to use randomization and thus remain fully asynchronous, we will now investigate some of these approaches.

### Quasi-Synchrony

Quasi-synchronism [VA95] is the model that is closest to a synchronous environment. The only difference is that a new class of failures is introduced so that the timing assumptions might be violated sometimes. A system is called *quasi-synchronous* if

– It can be defined by the properties of a synchronous system, namely

  1. bounded and known processing speed,
  2. bounded and known message delivery delays,
  3. bounded and known local clock rate drift,
  4. bounded and known load patterns and
  5. bounded and known difference among local clocks.

– There is at least one bound where there is a known nonzero probability that the bound assumptions do not hold — this probability is called *assumption coverage.*

– Any property can be defined in terms of a series of pairs (bound, assumption coverage).

In a fully synchronous system the fact that a process did not send a message is carrying clear information, and one can assume that any process that did not send a message when it was supposed to is faulty. This is not the case in the quasi-synchronous model. Protocols that rely on this information have to work with probabilities and can only speculate why an expected message did not arrive.

### Timed Asynchrony

The *timed asynchronous* system model [CF95] is similar to the quasi-synchronous model, but more assumptions are dropped to achieve a realistic behavior. The basic idea is to give the processes access to hardware clocks, so they can measure (global) real time within some (bounded) error range. No timing assumptions exist on the network. The system assumes a *user timeout* for the overall service (i.e., the highest-level applications). If the protocols do not finish in this time, they are externally aborted and considered failed. (one could, for example, think of a teller machine; if the client does not get his money after 10 minutes, she will press the “abort” button and go home to file a complaint).

The failures in this model can either be network or process failures. Network failures are either omission– (a message is lost) or performance failures. Processes failures are either crash– or performance failures; there are no Byzantine failures in this model.

There is no limit on the number of failures; the frequency of failures (both communication and network) is unbounded.

The important point to enable Byzantine agreement is the timed services. Although no timeout for network and processing speed is used, this model assumes that the user of a system will not wait forever and thus implicitly defines a timeout (the *user-timeout*). Therefore, any network/processor delays that exceed this time lead to the entire system to be considered faulty, which is then is inevitable.

Moreover, timing failures of individual processes are treated as "real" failures; a process that does not answer in time is a faulty process. However, this means there now is a kind of failure from which a process can recover. A process that once failed might work perfectly well again at a later point in time.

Furthermore, Cristian and Fetzer [CF99] develop a careful model of real-world systems, including for example formal definitions of clocks and clock drifting. Their analysis shows that within this model, a local network of workstations is nicely represented. On the other hand, this causes problems in networks that are less reliable than a LAN, such as the Internet. Furthermore, the model only considers crash and performance failures; the presence of Byzantine adversaries might spoil this model, as these adversaries can influence the speed of the network and of otherwise honest processes, for example by intentionally increasing the clock drift.

A protocol that solves consensus in this model is given in [FC95]. A fixed timeout is assumed, allowing a standard leader-election protocol as in a fully synchronous model.

#### Partial Synchrony

The above models are nice to work with, but they lack flexibility as they have very concrete requirements on the environment (e.g., the user timeout or known probabilities) that may not always be satisfiable. For example, a Byzantine adversary on a network could perform an easy attack by slowing the network down. In quasi-synchronism, this would alter the probabilities and is thus not covered. In the timed asynchronous model, some sort of timeouts are always included, so similar problems occur with a highly variable network speed. Timeouts might either be too high (which results in less performance) or they are low, which results in a good performance in the failure-free case, but a high error rate in case of an attack. Furthermore, not all networks have a constant or predictable behavior; while a LAN is under good control, in the Internet everything can happen.

In the partial synchrony model [DLS88], fixed upper and lower bounds for message delivery exist, but are not known to the system; another definition of partial synchrony by the same authors assumes that the bounds are known but not valid until some undetermined future time. Both these models are equally strong.

#### Failure Detectors

A *failure detector* is an abstraction that allows a protocol to detect events that normally are undetectable in an asynchronous system. It was first introduced by Chandra and Toueg [CT91], initially for the crash-failure model.

A failure detector (locally) *suspects* parties of being faulty. Even though failure detectors are usually allowed to be unreliable, i.e., to make false suspicions, they are not allowed to always be wrong. More specifically, a failure detector has to satisfy two basic conditions, namely, *accuracy* and *completeness*.

For the weakest failure detector that is strong enough to deterministically solve agreement, the *accuracy* condition states that after some point, there is some honest party that will never be suspected by any honest party anymore; the *completeness* condition states that every dishonest party should be permanently suspected after some point.

In the Byzantine world, it is not possible to detect that a party is faulty; a faulty party can behave completely honestly and thus no difference from an honest party is observable.

Thus, Byzantine failure detectors only try to identify certain faulty behaviors. Whereas other means, like digital signatures, prevent corrupt parties from lying, the failure detectors only prevent them from not doing anything (useful) [MR97, DS98, KMMS97].

Failure detectors are a quite abstract construction; they can be built on top of all models discussed above, and most models can be built on an appropriate failure detector. Therefore, this model is most commonly used in the literature today.

#### Trusted Timely Computing Base

This model [VC99, Ver01] originates in the real-time world, where timing is an important aspect — if a job is not done in time, a fault has already occurred. This is similar to the *timed asynchrony*, and this model is considered a successor of timed asynchrony. All parties have access to a local *trusted timely computing base*, or short TTCB. The TTCB consists of a trusted coprocessor for each party, connected to the other coprocessors by a reliable, synchronous, but low-bandwidth network (the control channel).

Based on these trusted components, it offers services for higher-level protocols. There are two kinds of services. The *distributed security kernel* (DSK) offers low level primitives like consensus, distributed authentication, local authentication, and a random number generator. The actual implementation of these services can be quite simple, as they base on a trusted, synchronous hardware. In addition to the DSK, the TTCB offers timed services, basing on a tamper-resistant clock. These services encompass duration measurement and time-failure detection so that the corresponding party, even though it is subject to timing faults, can at least detect whether something went wrong.

### 2.6.2 Optimism and Pseudo-Time

To compete with the performance of synchronous protocols, we use *optimistic protocols*. These protocols make use of timing information to improve the performance, while maintaining the same security properties as their fully asynchronous counterparts.

For synchronous systems, some literature going a similar way exists [GP90, BNDDS92, Zam96], but reversing the role of the timed and the randomized protocol – the timed protocol is used as a backup to compensate “bad luck” in the randomized protocol. First, a randomized protocol is run to obtain an expected constant running time. If the protocol did not terminate after a certain number of rounds, a deterministic protocol is invoked to obtain a guaranteed (non-constant) upper bound on the running time.

**Abstract time.** We first define an abstract timeout mechanism. Each protocol instance has a *timer*. One can view the *timer* as a special state variable that takes on two values: *stopped* or *running*. Initially, the timer is *stopped*. A thread may change this value by executing **start timer** or **stop timer** commands. A thread may also simply inspect the value of the *timer*. A thread may also execute a **wait until** or **case** statement (see Section 2.7.4) with the *condition* **timeout**. Additionally, when the adversary activates a party, instead of delivering a message, it may deliver a *timeout signal* to a thread whose timer is *running*; when this happens, the timer is stopped, and if that thread is waiting on a *timeout*, the thread is activated.

This abstract timer can be implemented quite easily in our formal system model which does not include a timer mechanism. A **start timer** command is implemented by sending a unique message to oneself, and the adversary delivers a timeout signal by delivering this message.

Of course, in an actual implementation, the timeout mechanism is implemented by using a real clock. However, by giving the adversary complete control over the timeout mechanism in our formal model, we effectively make no assumptions about the accuracy of the clock, except that the clock runs forward.

**The optimistic aspiration.** Using the timer defined above, we can make an *optimistic aspiration*[2]. Intuitively, the optimistic aspiration is that as long as the network behaves normally (whatever that might mean), the running time of a protocol is predictable. In our model, the adversary can *always* violate this aspiration; protocols cannot base any safety condition on the optimistic aspiration. It is possible, however, to improve the performance of a protocol such that it can well compete with synchronous protocols as long as the optimistic aspiration holds.

[2]Originally, this was the optimistic assumption; it is, however, nothing we assume, but something we hope for.

On modeling our optimistic aspiration in terms common in the literature, i.e., in the *failure detector* model, the corresponding failure detector does not need any accuracy, at least not to guarantee liveness and safety — if all parties are suspected to be dishonest at all times, the performance will drop, but the protocol will still work.

Furthermore, our failure detector does not need to detect the failure of a particular party, it is sufficient to detect that the protocol does not proceed the way it should; therefore, it is more a *deadlock* detector than a *failure detector*.

The main difference to a failure detector as used in most of the literature is a difference in the motivation: Classically, a failure detector is used to allow deterministic protocols to circumvent the FLP impossibility result. It is not possible to implement this kind of failure detector in an asynchronous system without making extra assumptions. In contrast, the *deadlock* detector we need can very well be implemented asynchronously; a trivial (but stupid) solution would be to permanently suspect a deadlock.

To stress the fact that our detection mechanism lies completely within the asynchronous model and — as opposed to the classical failure detector — does not introduce any new assumptions, the term "failure detector" will not be used in this thesis, even though it is straightforward to reformulate all optimistic protocols and definitions in a way that they use the failure-detector terminology.

The performance gain of an optimistic protocol can be broken down into two factors:

The *optimistic performance* covers all quantitative measures that apply if the optimistic aspiration never fails, e.g., computation and communication complexity, number of protocol rounds etc.

The *stability* is a measure on how the performance drops relative to the severity of the violations of the optimistic aspiration.

Finding a metric to measure stability is quite a difficult task; there are many factors in the network that can influence the performance of the system (maximum message delay, average message delay, number of time the message delay exceeds some expected value, level of malice in scheduling messages), and finding a general model to evaluate generic protocols would be a thesis in itself.

For the protocols in this thesis, we can define a relatively easy metric. All our optimistic protocols have two main phases, an optimistic phase running a fast, but insecure protocol, and a (hopefully not needed) pessimistic phase running the fallback protocol. The outline of such a protocol is given in figure 2.2

```
upon getting the start message
    start timer
    start the fast protocol

upon finishing the fast protocol
    stop timer

upon receiving a timer signal
or detecting a problem with the fast protocol
    stop the fast protocol and start the fallback protocol
```

Figure 2.2: Outline of our optimistic protocols

The stability is then simply measured by the number of corruptions and unexpected network delays it takes to make the protocol switch to the fallback protocol. This approach to optimistic protocols differs from other approaches in the literature [AT96], in which the optimistic (failure detector based) and the pessimistic protocols are interleaved and run simultaneously. Our metric on stability does not make sense in this case, as these protocols degrade more continuously.

## 2.7 Architectural Issues

So far, we have only looked at a single protocol in isolation, without considering the interplay between different protocols.

In this section, we address the concurrent execution of several protocols and protocol instances. Furthermore, our architecture is modular in that protocol instances may also invoke other protocol instances on their behalf as sub-protocols.

The implications of this setting go beyond naming and interfacing problems. Having several protocol instances and layers simultaneously implies that resources are shared between protocol instances. The most important one is the cryptographic infrastructure, as it is unreasonable to assume that every protocol instance has its own, independent cryptographic keys. Thus, some care has to be taken in preventing interleaving attacks and in accessing the cryptographic primitives. For example, the distributed coin generator in [BS93] requires a linear (i.e., ordered) opening of the coins and is thus not applicable if several protocols independently access coins.

### 2.7.1 Transactions and Modular Protocol Composition.

Every sub-protocol has one unique parent protocol. Therefore, the dynamic relation between all concurrently running protocol instances is given by a forest (i.e., a collection of trees) in which every sub-protocol points to its parent. The "root" protocols with no parents represent instances directly invoked by a user application. All other protocol instances are invoked as sub-protocols of an already running parent instance.

To identify protocol instances, we assume that each instance is associated with a unique *tag* $ID$. The value $ID$ is an arbitrary bit string whose structure and meaning are determined by a particular protocol and application. In our formal model, the tags of the root instances are chosen by the adversary because the adversary invokes them.

We do not allow the adversary to choose the same $ID$ for different transactions. If two different protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ were initialized with the same $ID$, it would result in a partition of the set of honest parties; some would participate in $\mathcal{P}_1$, ignoring messages for $\mathcal{P}_2$ as they do not comply the syntax for the expected messages, and vice versa. Thus, both protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ may have too few honest participants to terminate and may stall.

Sub-protocols are identified by hierarchical tags of the form $ID|ID'|\ldots$. The tag value $ID|ID'$ typically identifies a sub-protocol of the parent protocol instance $ID$ and is determined by the parent. The adversary may not introduce a new tag on its own if this extends any previously introduced tag, i.e., the set of tags chosen directly by the adversary must be prefix-free.

In our formal model, the adversary impersonates the user application. The reason for this is that a protocol that meets our specifications will behave accordingly in any application, as our definition requires that it works for *any* adversary. Applications whose behavior can be influenced by an adversary are common in distributed systems with security requirements.

The adversary may also impersonate sub-protocols and parent protocols, as long as it satisfies the specification of those protocols. Thus, a protocol behavior is only defined by the formal specification, and might be as malicious as is possible within these limits.

### 2.7.2 Messages and Interface.

The protocols communicate with a single communication interface to which the adversary delivers messages. This interface is used for both communication between similar protocols on different machines, and communication between higher- and lower-level protocols, as a placeholder for local invocation of sub-protocols. Thus, syntactically, invoking a sub-protocol appears as if it were a request of the adversary in our formal model. This fits the reality of our model, as higher- and lower-level protocols are incorporated into the adversary and can behave arbitrarily, as long as they meet the requirements in the corresponding specification.

Each party runs an *dispatcher* that delivers messages to the protocol instance associated with the corresponding $ID$. The detailed mechanism for composing protocols is part of the dispatcher described in Section 2.7.4

There are three different types of messages: input actions, output actions, and protocol (implementation) messages. *Input* and *output* actions represent local events and provide local input or carry local output to or from a protocol instance, which might be a sub-protocol of an already running protocol.

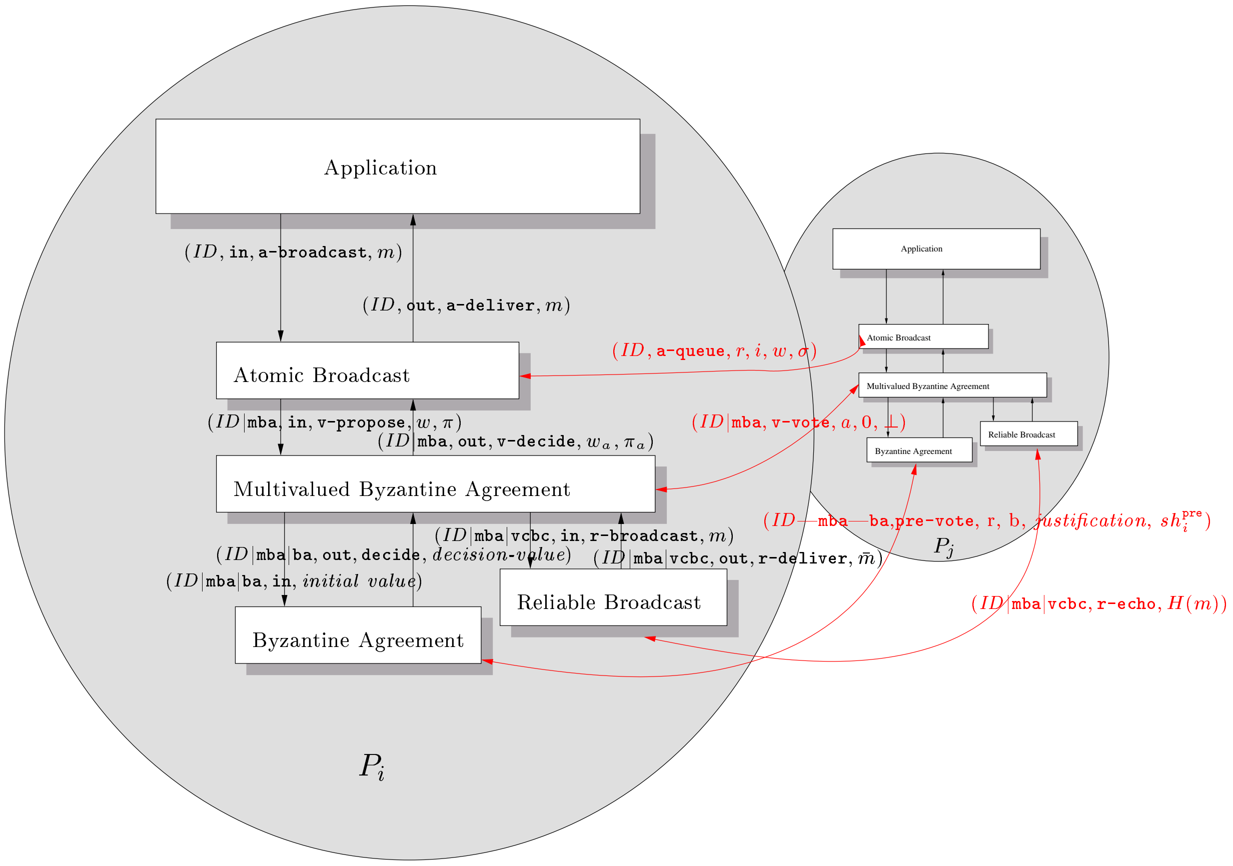

Application
$(ID, \texttt{in}, \texttt{a-broadcast}, m)$
$(ID, \texttt{out}, \texttt{a-deliver}, m)$
Atomic Broadcast
$(ID, \texttt{a-queue}, r, i, w, \sigma)$
$(ID|\texttt{mba}, \texttt{in}, \texttt{v-propose}, w, \pi)$
$(ID|\texttt{mba}, \texttt{out}, \texttt{v-decide}, w_a, \pi_a)$
Multivalued Byzantine Agreement
$(ID|\texttt{mba}, \texttt{v-vote}, a, 0, \perp)$
$(ID|\texttt{mba}|\texttt{vcbc}, \texttt{in}, \texttt{r-broadcast}, m)$
$(ID|\texttt{mba}|\texttt{ba}, \texttt{out}, \texttt{decide}, \textit{decision-value})$
$(ID|\texttt{mba}|\texttt{vcbc}, \texttt{out}, \texttt{r-deliver}, \bar{m})$
$(ID|\texttt{mba}|\texttt{ba}, \texttt{in}, \textit{initial value})$
Reliable Broadcast
Byzantine Agreement
$(ID—\texttt{mba}—\texttt{ba},\texttt{pre-vote}, \mathrm{r}, \mathrm{b}, \textit{justification}, sh_i^{\texttt{pre}})$
$(ID|\texttt{mba}|\texttt{vcbc}, \texttt{r-echo}, H(m))$
$P_i$
$P_j$
Application
Atomic Broadcast
Multivalued Byzantine Agreement
Byzantine Agreement
Reliable Broadcast

Figure 2.3: Message flow between protocol layers and parties

On the "protocol stack" of the layered architecture, input and output actions travel vertically: inputs "down" to sub-protocols and outputs "up" to higher-layer protocols. All other messages are *protocol messages*, generated and processed by the protocol implementation; they are intended for the peer instances running at other parties on the same level of the stack (directed "horizontally").

These messages are internal implementation messages and are distinct from the messages actually disseminated as payloads of the broadcast protocols, which messages are sometimes explicitly called *payload* messages.

An *input action* is a message of the form

$$(ID, \texttt{in}, \textit{action}, \ldots),$$

where *action* is specific to the protocol and followed by arbitrary data. Input actions represent local invocations of a protocol, either as a root protocol instance by the adversary or as a sub-protocol of an already running protocol instance. An input action is used to request a service from the protocol instance. For all protocols, there is a special input action *open*, represented by

$$(ID, \texttt{in}, \texttt{open}, \textit{type}),$$

which must precede any other input action with tag *ID* for that party. When $P_i$ processes such a message with tag *ID* for the first time, it initializes the instance; *type* specifies the type of the protocol being initialized. We say that $P_i$ has *opened* a "channel" with tag *ID* or *activated* a "transaction" with tag *ID*. (Although it is a crucial element, it mostly occurs implicitly before the first regular input action and will not be mentioned when describing the protocols.)

An *output action* is a message of the form

$$(ID, \texttt{out}, action, \dots),$$

where *action* is dependent on the particular protocol. These messages typically contain an output from the protocol instance to the calling entity. There is a special output action *halt*, represented by

$$(ID, \texttt{out}, \texttt{halt}),$$

after which no further messages tagged with $ID$ are processed by the party. When $P_i$ generates such a message with tag $ID$, we say that $P_i$ has *halted* instance $ID$.

If a protocol has only one input action (besides `open`) and one output message (besides `halt`), we also call the corresponding messages *start-message* and *end-message*.

We stress that in a real protocol implementation, neither the input nor the output actions involve any real network communication but will be mapped onto local events being generated or processed by the calling entity. But in the formal model at least some of them are generated and received by the adversary.

The third type of message generated by $P_i$ is of the form

$$(ID, i, j, \dots),$$

where $1 \leq j \leq n$ denotes the index of the recipient. Such a message is called a *protocol message*; the idea is that the adversary delivers it to $P_j$, where it is processed by the corresponding protocol instance.

Recall that messages are *authenticated*, which means that we restrict the adversary's behavior as follows: if $P_i$ and $P_j$ are honest, and the adversary delivers a protocol message $m$ of the form $(ID, i, j, \dots)$ to $P_j$, then $m$ was generated by $P_i$ at some earlier point in time.

There is a little subtlety concerning the message size. To identify the sender and the recipient of the message, an address of size at least $\log n$ is required, thus the size of a message cannot be independent of the system parameters. In our model, this is dealt with by the fact that all system parameters are bound by a *fixed* polynomial in the security parameter $\kappa$; in the real world, there is an upper bound on $n$ implied by the environment. With the current Internet for example, the address size is 32 bit.

### 2.7.3 Protocol Syntax

Our protocol descriptions are mostly written in reactive style. They consist of one or several threads, each of which consist of a set of clauses that **wait for** some condition happen (which usually involves the arrival of a message), update their state, and produce an output (i.e., one or several messages), and then wait for some new condition (or terminate the thread):

```
wait for condition 1
    action 1
wait for condition 2
action 2 ...
```

In addition to waiting on a single condition, a thread may wait on a number of conditions simultaneously by executing the statement:

**case upon** *condition*$_1$ : *action*$_1$ ; $\cdots$ **upon** *condition*$_k$ : *action*$_k$ ; **end case**

Each *action*$_i$ is an action to be executed when the condition is satisfied. If more than one condition is satisfied, then an arbitrary choice is made.

To handle incoming or interprocess messages, every party has one global message buffer, the *message queue*. We specify two special conditions that access the buffer in different ways. The first condition is of the form **receiving** *messages*. In this case, *messages* describes a set of of one or more messages satisfying a certain predicate, possibly involving other state variables. Upon executing this statement, a thread enters a wait state, waiting for the arrival of messages satisfying the predicate given; moreover, when this predicate becomes satisfied, the matching messages are *moved* out of the incoming message queue, and into local state variables. Unless otherwise specified, if there is more than a single message that satisfies the predicate, then the first such (i.e., oldest) message in the incoming message queue is selected; otherwise, no particular order is guaranteed.

The second condition we specify has the form **detecting** *messages*. The semantics of this are the same as for **receiving** *messages*, except that the matching messages are *copied* from the incoming message queue into local state variables.

There also is a specific **halt** operation. This is translated into the output action *halt* for the corresponding tag *ID*, and the instance *ID* is removed; further messages tagged with *ID* are ignored.

Outside of those instructions, we use standard constructions like **if ... then ... else**, **repeat ... until** or **for ... to ...**

### 2.7.4 Message Dispatching

As already mentioned in the description of the protocol syntax, every party runs one or several threads. Each opened protocol instance runs in at least one separate thread, but some protocols might involve several threads. However, at any given point in time, only one threat is active. Our protocols are reactive protocols, i.e., on being activated, they may perform a (brief) action after which they enter a *wait state*, waiting to be activated again.

Therefore, at the time the adversary activates a party, each thread of all protocol is in some *wait state*, waiting for some condition on the message buffer and its local state variables to hold.

On activating the party, the adversary usually delivers one or more messages to this party, which are appended to the message buffer. Then, the *dispatcher* is invoked to schedule the threads that may react on a message.

The delivered message may cause some *wait for* conditions a to hold. As soon as this happens, we call the corresponding thread *ready*.

The adversary may also deliver an input action *open* with a tag *ID* that has not been opened before. In this case, a new protocol instance with the specified *ID* is initialized and the dispatcher remembers that it was started externally over the network, as opposed to being started by a local higher level protocol.

The dispatcher now selects an arbitrary thread which is *ready*; this thread then follows the instructions defined for this condition, i.e., does a state-update and maybe produces some output messages. The dispatcher schedules the ready threads in some arbitary order, with one restriction: if both a child and its parent (or any ancestor) are ready, the child thread must be scheduled before its parent.

The messages generated by an instance *ID* are treated as follows. *Protocol messages* are simply written to the outgoing communication interface. For each input action *open*, a new protocol instance with the specified child *ID* is initialized, as if the message came from the network. The dispatcher remembers the *ID* of the parent instance; all subsequent *input actions* from the parent addressed to the child are not written out to the network, but included directly in the buffer. Each *output action* of a sub-protocol instance *ID* is mapped directly into a corresponding internal message for its parent; *output actions* of a root protocol instance are written to the outgoing communication interface. These steps allow local activation of sub-protocols and local processing of their output to be described in terms of the single message interface.

We assume atomic scheduling, i.e., a thread is not interrupted; once activated, it only gives back control to the dispatcher when it waits for a new event to happen. Of course, we restrict ourselves to polynomial-time protocols that always relinquish control to the adversary, i.e., both the computation effort for a state update and the number of possible conditions are bounded.

The dispatcher continues scheduling ready threads until no more thread is ready. Once this happens, control is given back to the adversary and the party waits to be activated again.

A message that triggered the condition may be removed from the buffer (if the condition was *receiving* it, or it might remain in the buffer (if it was *detected*). Therefore, and because there are some messages that no protocol waited for, there may still be messages left in the buffer. Those messages stay in the buffer for the next activation of the corresponding party.

Correctness and security of a protocol instance should not depend on the implementation of the dispatcher, as long as it obeys these rules (to be consistent with our previous definitions, one can see the dispatcher as a part of the adversary).

### 2.7.5 Implementation Issues and Machine Model

The model introduced so far uses some abstractions that help concentrating on the main aspects of the cryptographic model, security proofs and protocols, but hide some factors that might be of importance in a real-world system.

**Eventual message delivery.** Even though our model does not guarantee that all messages are delivered, any useful protocol can only tolerate a limited message loss. There are various ways a protocol can deal with this problems, all of which require some timing – after all, message loss can only be dealt with if it is detected, which is impossible in a fully asynchronous system.

There are two fundamental ways of dealing with lost messages, the *push* and the *pull* model. The *push* model puts the responsibility for message delivery onto the sender. The receiver acknowledges every message received, and the sender keeps on sending a message until it receives this acknowledgment. This is the model used, for example, in TCP/IP.

Alternatively, in the *pull* model, the responsibility is put onto the receiver. It has to know when to expect a message, and it complains to the sender if no message is received in time until it gets one.

For generic applications, the push model has several advantages. The receiver knows when a message has arrived and thus can be forgotten, as opposed to the pull model, were a complaint could arrive at any time, and thus (theoretically) the sender has to store all messages it ever sent until the end of time. Furthermore, it is not always the case that the receiver knows when to receive a message – it is not possible to send unexpected messages reliably if the pull model is used. Finally, on a lossy network, the push model is more efficient – in the pull model, it is possible that an arbitrary number of complaints are sent before the sender tries to send the actual message.

Nevertheless, if the protocol in question can tolerate a certain message loss, i.e., it is only important that some messages arrive, the pull model might prove more efficient (see Section 4.8).

**Atomic actions.** In our model, we assume an atomic receive/compute/send step, i.e., an honest party is activated by receiving some message, performs an action, and then sleeps until it is activated again.

Thus, the adversary is not able to activate some honest party while it or another honest party is still processing a former activation. In the asynchronous model, this does not weaken the adversary. For honest parties, there is no difference between the atomic model and a non-atomic model, as all communication goes through the adversary and no clocks are available. Whether two parties perform some action in parallel or one after the other cannot be detected by the honest parties.

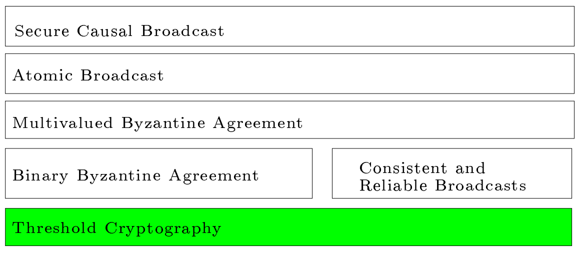
For the adversary, there is no difference either, as the computation of an an honest party is activated by the delivery of a single message. Activating party $P_j$ before the earlier activated party $P_i$ has finished its computation does not help, as the adversary cannot influence $P_i$ anymore before it finishes and goes to sleep again.

**Memory.** Our theoretical model requires the parties to have infinite memory, or at least enough memory to store all messages that can be generated in polynomial time. This is unavoidable; a party might receive an arbitrary number of messages that it cannot immediately process, and these messages have to be stored somewhere.

In a real-world system, this problem can be dealt with using garbage collection and timeouts; if a message could not be processed for two years, the odds are that it will never be processed.

# Chapter 3

# Cryptographic Primitives


Secure Causal Broadcast
Atomic Broadcast
Multivalued Byzantine Agreement
Binary Byzantine Agreement
Consistent and Reliable Broadcasts
Threshold Cryptography

## 3.1 Introduction

Our protocols employ modern cryptographic techniques to a far greater extent than has been done previously in the literature of fault tolerance. The basic cryptographic primitives required are a threshold signature scheme and a threshold, random-access, coin-tossing scheme.

A threshold signature scheme allows a party to prove that it receives signatures on a message by several parties. Every participant can issue signature shares on a message, and a party that has collected enough shares can combine them to one valid signature.

A random access coin-tossing scheme is a function mapping a name onto a pseudorandom binary value, in a way that even participants of the protocol can not previously predict the value unless the number of parties participating in predicting the value exceeds some threshold the number of parties participating in predicting the value exceeds some threshold. This protocol is needed to provide pseudo-randomness for distributed protocols.

Our primitives can be implemented quite practically, while still providing a strong security guarantee. More precisely, these primitives can be efficiently implemented and proved secure under standard intractability assumptions in the *random oracle* model. In this model, one treats a cryptographic hash function *as if* it were a black box containing a random function.

The random oracle model was first used in a rather informal way by Fiat and Shamir [FS87]; it was first formalized and used in other contexts by Bellare and Rogaway [BR94] and has since been used to analyze a number of practical, cryptographic protocols. Of course, it would be better not to rely on random oracles, as they are essentially a heuristic device; nevertheless, random oracles are a useful tool—they allow us to design truly practical protocols that admit a security analysis, which yields very strong evidence for their security. The notion of a *threshold signature scheme* was introduced by Desmedt and Frankel [Des88, DF90, DF92] and has been widely studied since then (T. Rabin [Rab98] provides new results and a survey of recent literature). We use a "dual-parameter" threshold signature scheme, meaning that the "reconstruction threshold" $k$ (i.e., the number of signature shares needed to obtain the signature) is allowed to be higher than $t+1$ (whereas $t$ is the number of corrupted parties tolerated). This is in contrast to all previous work on threshold signatures in the literature where $k = t+1$. V. Shoup [Sho00] presents a practical "dual-threshold" signature scheme that is secure in the random oracle model under standard intractability assumptions. The signatures created by this scheme are ordinary RSA signatures. Moreover, the scheme is completely non-interactive, an individual share of a signature is not much larger than an ordinary RSA signature, and even for $k = t + 1$, it is the first rigorously analyzed non-interactive threshold signature scheme with small shares. A non-interactive threshold signature scheme is one where each party outputs a signature share upon request and there is an algorithm to combine $k$ valid signature shares to constitute a valid signature.

Coin-tossing schemes are used in one form or another in essentially all solutions to the asynchronous Byzantine agreement problem. Many schemes, following Rabin's pioneering work [Rab83], assume that coins are predistributed (and possibly signed) by a dealer using secret-sharing [Sha79]. This approach has two problems: first, the coins will eventually be exhausted; second, parties must somehow associate coins with transactions (which itself represents an agreement problem). Because of these problems, protocols that rely on a "Rabin dealer" are not really suitable for use as a transaction processing service as described above. The same applies to the coin-tossing scheme of Beaver and So [BS93], which essentially gives parties sequential access to a bounded number of coins. A "Rabin dealer" has been used in other contexts as well, e.g., for threshold decryption [CG99]. The beautiful work of Canetti and T. Rabin [CR93] presents a coin-tossing scheme that allows common coins to be generated entirely "from scratch". Unfortunately, this scheme, while polynomial time, is completely impractical.

Our approach to coin-tossing is to use a random-access coin-tossing scheme—essentially a distributed function mapping the "name" of a coin to its value. Such coin-tossing schemes have been studied before [MS95, NPR99]. We also define the notion of a "dual-parameter" coin-tossing scheme, which is convenient and can lead to lower communication complexity, but does not necessarily. One could easily implement such a coin using the non-interactive threshold signature scheme of Shoup [Sho00]; however, we present a dual-parameter threshold coin-tossing scheme that is based on the Diffie-Hellman problem, the analysis of which may be interesting in its own right. This scheme is essentially the same as the one of Naor *et al.* [NPR99], but our analysis is more refined: first, for the single-parameter setting, we need a weaker intractability assumption, and second, we provide an analysis of the scheme in the dual-parameter setting, which is not considered by Naor *et al.*

We stress that such dual-parameter threshold schemes provide stronger security guarantees than single-parameter threshold schemes, and are in fact more challenging to construct and to analyze (see Section 3.5.3). Our notion of a dual-parameter threshold scheme should not be confused with a weaker notion that sometimes appears in the literature (e.g., [MS95]). For this weaker notion, there is a parameter $l > t$ such that the reconstruction algorithm requires $l$ shares, but the security guarantee for a given signature/coin is lost if just a *single* honest party reveals a share. In our notion, no security is lost unless $k - t$ honest parties reveal their shares.

This dual threshold will be used in the Byzantine agreement protocol to obtain more robustness; it is helpful that the coin is not revealed when the first honest party reaches the point where it reveals its share (i.e., the adversary has $t + 1$ shares), but only when $t + 1$ honest parties did so, i.e., the protocol is more advanced.

In defining schemes, we will distinguish between a syntactical description (*the action*) and a semantical one (*security requirements*). The syntactical description makes sense here as the schemes consist of several functions that take each others output as input, and it is important to clearly define which function operates on which input. In later chapters, we will not need this distinction anymore; most higher level protocols have a quite boring syntax (a Byzantine agreement protocol takes an input value for each party and produces an output value for each party), and there are few syntactical dependencies that already appear on that level of the description.

## 3.2 Digital Signatures

A digital signature scheme [GMR88] consists of a *key generation* algorithm, a *signing* algorithm, and a *verification* algorithm. The key generation algorithm takes as input a security parameter, and outputs a public key/private key pair $(PK, SK)$. The signing algorithm takes as input $SK$ and a message $M_i$, and produces a signature $\sigma_i$. The verification algorithm takes $PK$, a message $M_i$, and a putative signature $\sigma_i$, and outputs either `accept` or `reject`. A signature is considered *valid* if and only if the verification algorithm accepts. All signatures produced by the signing algorithm must be valid.

The basic security property is *unforgeability.* The attack scenario is as follows. An adversary is given the public key, and then requests the signatures on a number of messages, where the messages themselves may depend on previously obtained signatures. If at the end of the attack, the adversary can output a message $M_i$ and a valid signature $\sigma_i$ on $M_i$, such that $M_i$ was not one of the messages whose signature he requested, then the adversary has successfully *forged* a signature. Security means that it is computationally infeasible for an adversary to forge a signature.

## 3.3 Threshold Signatures

In this section, we define the notion of an $(n, k, t)$ threshold signature scheme. The basic idea is that there are $n$ parties, up to $t$ of which may be corrupted. The parties hold "shares" of the secret key of a signature scheme, and may generate "shares" of signatures on individual messages. As mentioned above, we have a dual threshold — the number of corrupted parties $t$ is independent of the number $k$ of signature shares both necessary and sufficient to construct a signature. The only requirement on $k$ is that $t < k \leq n - t$. As mentioned in the introduction, previous investigations into threshold signatures have only considered the case $k = t + 1$. Also, we shall require that the generation and verification of signature shares is completely non-interactive—this is essential in the application of asynchronous Byzantine agreement.

A threshold signature scheme is a multi-party protocol, and we shall work in our basic system model for such protocols (see Chapter 2).

### 3.3.1 The Action

The scheme is initialized by a trusted dealer that is destroyed afterwards. It is possible to distribute the dealing phase as well [Ped91, GJKR99]; however, it is not known how to do this asynchronously by now.

The dealer generates a public key $PK$ along with secret key shares $SK_1, \ldots, SK_n$, a global share verification key $VK$, and local share verification keys $VK_1, \ldots, VK_n$. The initial state information for party $P_i$ consists of the secret key $SK_i$ along with the public key and *all* the verification keys.

After the dealing phase, the adversary submits signing requests to the honest parties for messages of his choice. Upon such a request, party $P_i$ computes a *signature share* for the given message using $SK_i$.

The threshold signature scheme also specifies three algorithms: a *signature verification* algorithm, a *share verification* algorithm, and a *share combining* algorithm.

– The *share verification* algorithm takes as input a message, a signature share on that message from a party $P_i$, along with $PK$, $VK$, and $VK_i$, and determines if the signature share is valid.

– The *share combining* algorithm takes as input a message and $k$ valid signature shares on the message, along with the public key and (perhaps) the verification keys, and (hopefully) outputs a valid signature on the message.

– The *signature verification* algorithm takes as input a message and a signature, along with the public key. It outputs either **accept** or **reject**; if it outputs **accept**, we say that the signature is valid.

### 3.3.2 Security Requirements

The two basic security requirements are *robustness* and *non-forgeability.*

*Robustness.* It is computationally infeasible for an adversary to produce $k$ valid signature shares such that the output of the share combining algorithm is not a valid signature.

*Non-forgeability.* It is computationally infeasible for the adversary to output a valid signature on a message that was submitted as a signing request to less than $k - t$ honest parties. Note that if the adversary actually corrupts $f < t$ parties, the relevant threshold is still $k - t$ and not $k - f$.

### 3.3.3 Implementation

Note that our definition of a threshold signature scheme admits the "trivial" implementation of just using a set of $k$ ordinary signatures. For relatively small values of $n$, this may very well be a perfectly adequate implementation. The way we use threshold signatures, there is no disadvantage in this approach. Other application however might need additional properties, for example a deterministic scheme that produces the same signature independent of the set of signers; those applications need a more advanced scheme.

The scheme of Shoup [Sho00] is well suited to our purposes and is much more efficient than the above "trivial" implementation when $n$ gets large. Each signature share is essentially the size of an RSA signature, and shares can be quite efficiently combined to obtain a completely standard RSA signature. The signature shares come with "proofs of correctness." These correctness proofs are not much bigger than RSA signatures; however, in an efficient implementation, one would most likely omit these proofs (and their verification), and only provide them if they are explicitly requested, presumably by a party whose share combination algorithm has failed to produce a correct signature.

### 3.3.4 Replacing Cryptography with Interaction

It is possible to build a threshold signature scheme that does not rely on cryptographic assumptions; furthermore, it requires no expensive computations at the price of a higher communication complexity, making it useful if communication is cheap relative to computation.

**Definition 3.1 ((interactive) Non-rejecting threshold signatures).** A *non-rejecting threshold signature scheme* is a threshold signature scheme in which the signature verification algorithm never outputs **reject**; it either outputs **accept** or does not terminate.

An *interactive non-rejecting threshold signature scheme* is a non-rejecting threshold signature scheme where the share generation is an interactive protocol initiated by the signer that may or may not terminate; if the signer is honest, it does terminate. The only other protocol is the signature verification protocol, that outputs **accept** if enough share generation protocols have terminated, and otherwise waits for this to happen.

For compatibility with threshold signatures, the share generation and verification algorithms exist, but do not do anything.

Note that a protocol $P$ must work with any threshold signature scheme, and therefore sees the scheme as a black box; it may not access the signatures by any means other than the algorithms specified in the definition of threshold signature schemes.

**Theorem 3.1.** *Assume that a protocol does not use the information that a threshold signature or signature share is invalid other than to ignore the corresponding message.*

*Then, a threshold signature scheme with non-rejecting verification is usable instead of a normal threshold signature scheme.*

*Proof.*

To prove this theorem, we define a filter protocol. This filter takes as an input a message $m$ and a signature verification algorithm $v$, runs the signature verification algorithm $v$ over all threshold signatures in $m$, and only forwards $m$ once the signature verification algorithm outputs **accept** for all those signatures.

Suppose we have a protocol $\mathcal{P}$ that uses a normal threshold signature scheme. Then we construct a protocol $\mathcal{P}'$ that uses a non-rejecting threshold signature scheme as follows.

The new protocol $\mathcal{P}'$ works as $\mathcal{P}$, but applies the filter protocol with the non-rejecting signature verification algorithm on all incoming messages.

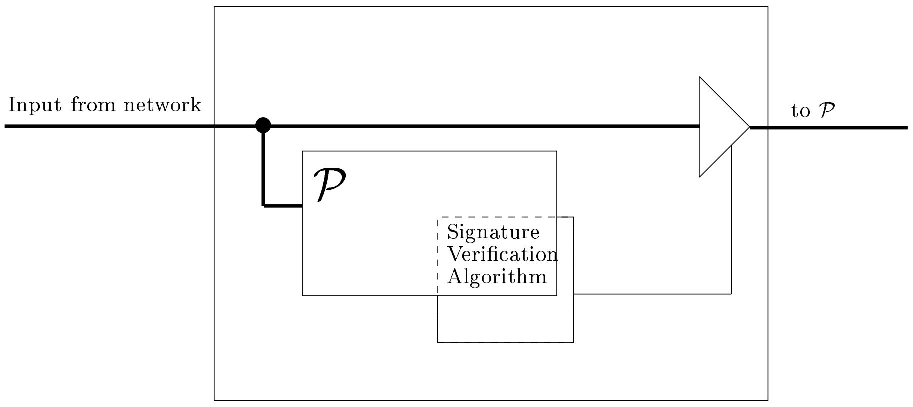


Figure 3.1: The filter protocol for $\mathcal{P}'$

This filter receives an incoming messages, validates all signatures $\mathcal{P}$ would validate on this message. This might require access to the same source of randomness that $\mathcal{P}$ has (alternatively we could assume that it is not randomly determined which part of a message forms a signature, which in most protocol is a quite safe assumption). If at least one of those verifications is negative, then the message is not passed to $\mathcal{P}$; otherwise, it passes the filter and is processed by $\mathcal{P}$. Thus, the filter can extract all signatures from a message that $\mathcal{P}$ uses, even if the signature is additionally encrypted or disguised.

Furthermore, whenever $\mathcal{P}$ applies the threshold signature verification algorithm, $\mathcal{P}'$ applies an algorithm that always outputs **accept**. Now suppose there is an adversary $\mathcal{A}$ that can break $\mathcal{P}'$. Then we construct a new adversary $\mathcal{A}'$ as follows.

The new adversary $\mathcal{A}'$ works as $\mathcal{A}$, but all corrupted parties apply the filter with the normal threshold signature verification algorithm on all outgoing messages.

As invalid signatures carry no information for $\mathcal{P}$ other that the corresponding message should be ignored, both combinations of adversary $\mathcal{A}$ with $\mathcal{P}'$ and adversary $\mathcal{A}'$ and $\mathcal{P}$ behave identical; the only difference is where the invalid signatures are filtered out.

This shows that $\mathcal{P}'$ satisfies the same specification as $\mathcal{P}$.

□

The restriction that invalid shares are only dropped is necessary. For example, a protocol could encode bitstrings as a sequence of valid and invalid threshold signatures. Furthermore, the whole message containing the invalid threshold signature must be ignored; a protocol could attach an invalid threshold signature to all messages, and the modified protocol would not let any message arrive.

Practically, we can assume that honest parties never generate an invalid threshold signature. Thus, messages containing invalid signatures are only sent by corrupted parties and therefore should not be used for anything but identifying a dishonest party as such. Furthermore, it is easy for a dishonest party to avoid being identified like this by simply not sending an invalid signature; thus, if a protocol only satisfies its specification if it can identify dishonest parties due to invalid threshold signatures, something is already wrong.

Therefore, in most practical settings (especially for all protocols in this thesis), non-rejecting threshold signature schemes are applicable to trade communication for computation.

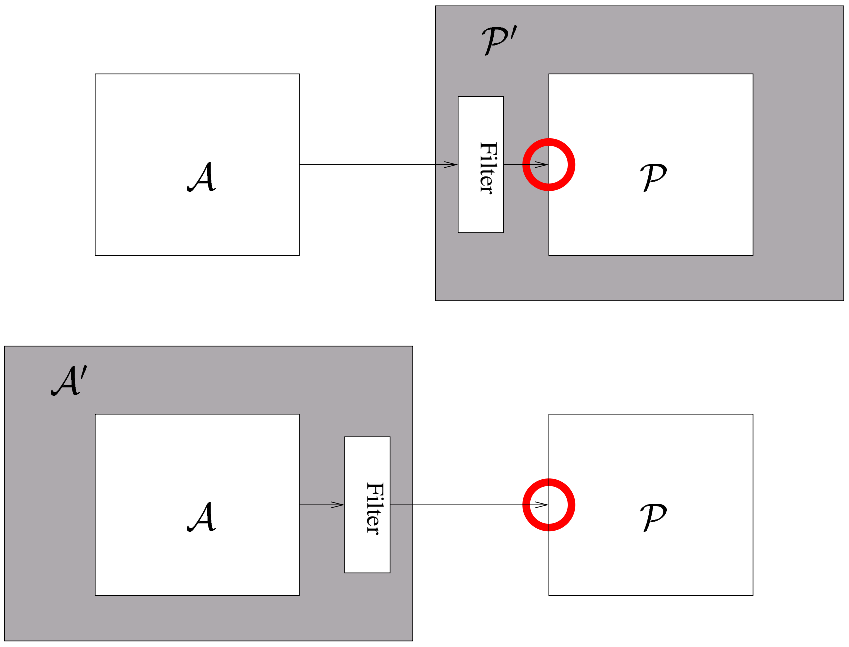


Figure 3.2: Equivalence of the pairs $\mathcal{A}/\mathcal{P}'$ and $\mathcal{A}'/\mathcal{P}$

**A non-rejecting scheme.** We now present a non-rejecting threshold signature scheme. The scheme builds on a reliable broadcast primitive as defined in Definition 5.4 in Section 5.4. In essence, an authenticated reliable broadcast guarantees that if one honest party receives a message $m$ broadcasted by some (not necessarily honest) party $P_j$, then all honest parties will receive this message. Furthermore, if some honest party $P_i$ completes the authenticated reliable broadcast of a message believing it has been sent by $P_j$, then it has been sent by $P_j$, provided $P_j$ is honest.

A protocol that implements authenticated reliable broadcast based on a collision–free hash function is presented in Section 5.4; a slightly less efficient protocol that does not rely on a hash function has been introduced by Bracha [Bra84].

Note that using a reliable broadcast to implement signatures contradicts the logical order in which the protocols are presented in this thesis. As reliable broadcast can be implemented without using any of the primitives introduced before, we can slightly violate the order here use it at this point anyway.

The reason that this scheme is non-rejecting lies in the asynchrony of the reliable broadcast; a party can not distinguish between a broadcast that never happened and itself just being slow. In a synchronous system, this problem would not apply and the scheme would not need to be non-rejecting.

With this primitive, our $(n, k, t)$ threshold signature scheme works as follows:

**share generation.** To generate a signature on $m$, reliably broadcast the message $(\texttt{shared}, m)$.

**share verification.** Any share that arrived (via completed reliable broadcast) and that has the correct syntax is valid.

**share combining.** If $k$ or more shares from different parties have been received (via completed reliable broadcast), output $(\texttt{signed}, m)$.

**signature verification.** On verifying a signature $(\texttt{signed}, m)$, wait until completing $k$ reliable broadcasts of the message $(\texttt{shared}, m)$, sent by different parties. Output **accept**.

If any honest party receives $k$ valid shares, by the properties of the reliable broadcast, all honest parties do. Thus, once an honest party's share combining algorithm outputs $(\texttt{signed}, m)$, it is guaranteed that all honest parties will accept the message once their reliable broadcasts terminate. Thus, the scheme is *robust*.

If less that $k - t$ honest parties issue a message $(\texttt{shared}, m)$, no honest party will ever complete the authenticated reliable broadcast of $k$ shares. Thus, no honest party will ever accept a signature $(\texttt{signed}, m)$. Therefore, the scheme is non-forgeable.

Note that the threshold signatures generated by this scheme can only be transfered to parties that received the reliable broadcast. This scheme does not allow to generate signatures that can later be transfered to third parties; this problem is solvable by using witnesses (e.g., if $t + 1$ parties say the signature is valid, then it is), but this property is not needed for the protocols presented in this thesis and will not be further investigated here.

For our protocols, using this scheme can be a really practical option if computation is expensive and communication is cheap.

## 3.4 Threshold Coin-Tossing Scheme

In this section, we define the notion of an $(n, k, t)$ threshold coin-tossing scheme. The basic idea is that there are $n$ parties, up to $t$ of which may be corrupted. The parties hold "shares" of an unpredictable function $F$ mapping the name $C$ (which is an arbitrary bit string) of a coin to its value $F(C) \in \{0, 1\}$ (this function $F$ is implicitly defined by the secret shares). The parties may generate "shares" of a particular coin—$k$ coin shares are both necessary and sufficient to construct the value of a coin. The only requirement on $k$ is that $t < k \leq n - t$, analogous to threshold signatures. The generation and verification of coin shares are completely non-interactive (see below); we work in the basic system model of Chapter 2.

### 3.4.1 The Action

The dealer generates secret key shares $SK_1, \ldots, SK_n$, and verification keys $VK, VK_1, \ldots, VK_n$. The initial state information for party $P_i$ consists of the secret key $SK_i$ along with all the verification keys.

After the dealing phase, the adversary submits *reveal requests* to the honest parties for coins of his choice. Upon such a request, party $P_i$ outputs a *coin share* for the given coin, which it computes using $SK_i$.

In addition, a threshold coin-tossing scheme specifies two algorithms: a *share verification* algorithm, and a *share combining* algorithm.

- The *share verification* algorithm takes as input the name of a coin, a share of this coin from a party $P_i$, along with $VK$ and $VK_i$, and outputs **accept** or **reject**. If it outputs —bf accept, we say that the share is valid.

- The *share combining* algorithm takes as input the name $C$ of a coin and $k$ valid shares of the coin, along with (perhaps) the verification keys, and (hopefully) outputs $F(C)$.

### 3.4.2 Security Requirements

The two basic security requirements are *robustness* and *unpredictability.*

*Robustness.* There is a function $F(C)$ that maps names to $\{0, 1\}$. It is computationally infeasible for an adversary to produce a name $C$ and $k$ valid shares of $C$ such that the output of the share combining algorithm is not $F(C)$.

*Unpredictability.* An adversary's *advantage* in the following game is negligible. The adversary interacts with the honest parties as above, and at the end of this interaction, he outputs a name $C$ that was submitted as a reveal request to fewer than $k - t$ honest parties, and a bit $b \in \{0, 1\}$. The adversary's advantage in this game is defined to be the distance from $1/2$ of the probability that $F(C) = b$. Note that if the adversary actually corrupts $f < t$ parties, the relevant threshold is still $k - t$ and not $k - f$.

### 3.4.3 Unpredictability for Sequences of Coins

The unpredictability property above implies a more general unpredictability property that we need in order to analyze agreement protocols.

Consider an adversary $\mathcal{A}$ that interacts with the honest parties as above, but as it interacts, it makes a sequence of predictions, predicting $b_i \in \{0,1\}$ as the value of coin $C_i$ for $i = 1, \ldots, q$ for some $q$. $\mathcal{A}$'s predictions are interleaved with reveal requests in an arbitrary way, subject only to the restriction that at the point in time that $\mathcal{A}$ predicts the value of coin $C_i$, it has made fewer than $k - t$ reveal requests for $C_i$. After it has predicted $C_i$, it may make as many reveal requests for $C_i$ as it wishes. For $1 \leq i \leq q$, let $e_i = F(C_i) \oplus b_i$. This defines the *error vector* $(e_1, \ldots, e_q)$.

The unpredictability property above implies that the error vector is *computationally indistinguishable* from a random bit-vector of length $q$. This means that there is no efficient statistical test that distinguishes the error vector from a random vector—the important point is that we are considering statistical tests that receive *only* the test vector as input, and no additional information about $\mathcal{A}$'s interaction in the above game.

A proof of this appears in the work of Beaver and So [BS93], although their setting is slightly different. The idea of the proof runs as follows. By the universality of the next-bit test [Yao82b], if the error vector were distinguishable from a random vector, then there would be an algorithm $D$ that on input $j$, chosen randomly from $\{1, \ldots, q\}$, along with $e_1, \ldots, e_{j-1}$, outputs a value that correctly predicts $e_j$ with probability significantly better than 1/2. Given this $D$ and $\mathcal{A}$, we construct a new adversary $\mathcal{A}'$ that predicts a single coin, contradicting the unpredictability assumption. $\mathcal{A}'$ runs as follows. First, it chooses $j \in \{1, \ldots, q\}$ at random. Next, it runs $\mathcal{A}$ as a subroutine. Just after $\mathcal{A}$ predicts coin $C_i$ for $1 \leq i < j$, $\mathcal{A}'$ immediately makes a sufficient number of reveal requests

to obtain $F(C_i)$, and hence $e_i$. $\mathcal{A}'$ stops $\mathcal{A}$ just after $\mathcal{A}$ makes its prediction $b_j$ for the value of $F(C_j)$, and then $\mathcal{A}'$ computes

$$\hat{b}_j = D(j; e_1, \ldots, e_{j-1}) \oplus b_j$$

as its prediction for $F(C_j)$ and halts. It is easy to see that $\hat{b}_j$ is correct with probability significantly better than 1/2.

Given the pseudo-random quality of the error vector, one can now easily derive a number of simple statistical properties. The only one we will need is this: for any $1 \leq m \leq q$, the probability that $\mathcal{A}$ correctly predicts the first $m$ coins is bounded by $2^{-m} + \epsilon$, where $\epsilon$ is a negligible function in the security parameter; we will need this property later in the efficiency analysis of the Byzantine agreement protocol.

### 3.4.4 Common coins from Threshold Signatures

An implementation of a coin-tossing scheme can be obtained from any non-interactive threshold signature scheme with the property that there is only one valid signature per message, such as the RSA-based scheme mentioned earlier [Sho00]. Then a cryptographic hash $H : \{0,1\}^* \to \{0,1\}$ can be applies on the signature to get the value of the coin. In the random oracle model, a coin predicting algorithm can simply be transfered into a signature forging algorithm. Any algorithm that predicts a coin with non-negligible advantage must have a non-negligible chance to query the random oracle at the right point. Simply observing the queries to the random oracle in this case gives us an advantage in guessing the input at this point, which is the threshold signature $\sigma$.

Using threshold signatures for the coins yields another advantage, as a party can prove it correctly tossed a coin to a party that did not toss that coin at all and did not see any coin shares whatsoever.

By the properties of the threshold signature scheme, the scheme now fits the definition below:

**Definition 3.2.** A *verifiable threshold coin tossing scheme* is a threshold coin tossing scheme with the following modifications:

– The *share combining* algorithm takes as input the name $C$ of a coin and $k$ valid shares of the coin, along with (perhaps) the verification keys, and (hopefully) outputs $F(C)$ and a witness $F_w(C)$.

– The *coin verification* algorithm takes as input the name $C$ of a coin, a value hopefully $F(C)$, a witness (hopefully $F_w(C)$) along with $VK$, and determines if the coin is valid.

The new security requirements are *robustness*, *unpredictability* and *non-forgeability Robustness.* There is a function $F(C)$ that maps names to $\{0,1\}$. It is computationally infeasible for an adversary to produce a name $C$ and $k$ valid shares of $C$ such that the output of the share combining algorithm is not $F(C)$.

*Unpredictability.* An adversary's *advantage* in the following game is negligible. The adversary interacts with the honest parties as above, and at the end of this interaction, he outputs a name $C$ that was submitted as a reveal request to fewer than $k-t$ honest parties, and a bit $b \in \{0,1\}$. The adversary's advantage in this game is defined to be the distance from $1/2$ of the probability that $F(C) = b$. Note that if the adversary actually corrupts $f < t$ parties, the relevant threshold is still $k-t$ and not $k-f$.

*Non-forgeability.* It is computationally infeasible for the adversary to output a valid coin on name $C$ of which less than $k-t$ honest parties have submitted coin shares.

In Section 3.5 we also present a direct implementation of a coin-tossing scheme based on the Diffie-Hellman problem.

## 3.5 A Diffie-Hellman Based Threshold Coin-Tossing Scheme

In this section, we present and analyze an $(n,k,t)$ threshold coin-tossing scheme based on the Diffie-Hellman problem. Let $G$ be a large finite cyclic group, e.g., $\mathbb{Z}_p^*$ with a large prime $p$. [1]

The scheme bases on the assumption that the *discrete logarithm* is a difficult problem, i.e., for a given group element $g \in G$, given $g^x$, it is difficult to find $x$; a slightly stronger assumption we will need later is that given for $g, h \in G$, given $g^x$ and $h^y$, one cannot even say if $x$ equals $y$.

In a nutshell, the scheme works by sharing some secret $x$ with a linear secret sharing scheme, such that $k$ parties can reconstruct their $x$ by pooling their shares, i.e., $k$ shares $x_{l_1}, \ldots, x_k$ are sufficient to compute $x$. Due to the linearity of the secret sharing scheme, this implies that the reconstruction can also take place in the exponent of a finite group, i.e., for some $g \in G$, the shares $g^{x_1}, \ldots, g^{x_k}$ can used to reconstruct $g^x$. Thus, we obtain a function that maps some public value (the group element $g$) onto some value $g^x$, which can only be computed if at least $k$ of the parties collaborate. This can be directly used as a common coin by simply hashing $g^x$ onto a binary value.

### 3.5.1 Basic Tools

**Shamir's Secret Sharing.** Our coin tossing scheme is based on Shamir's secret sharing scheme [Sha79]. The basic idea here is that a polynomial of degree $k$ is completely defined by $k+1$ values; given the evaluation of the polynomial at $k+1$ points, it can be evaluated at any other point. However, if only $k$ points are given, one has no information about what the evaluation of a some other point would be.

The secret space the scheme works on must be a subset of a finite field, e.g., a prime field $\mathbb{Z}_q$. The group $G$ would do perfectly for our purposes; however, to easily distinguish between exponents and mantissa, we keep shall use two different groups here.

In this $(n,k)$ secret sharing scheme, a dealer shares a secret $x_0 \in \mathbb{Z}_q$ among $n$ parties as follows:

The dealer chooses a random polynomial $f(T)$ over $\mathbb{Z}_q$ of degree $k-1$, such that $f(0) = x_0$. Each player $P_i$ then receives its secret share $x_i = f(\pi(i))$, where $\pi(i)$ is a permutation on $\mathbb{Z}_q$; we can assume for simplicity that it is the identity function.

Now, for any set $S$ of $k$ distinct points in $\mathbb{Z}_q$, and any $j \in \mathbb{Z}_q$, there exist elements $\lambda^S_{i,j} \in \mathbb{Z}_q$ for $i \in S$, such that

$$\sum_{i \in S} f(i)\lambda^S_{i,j} = f(j).$$

[1]One sometimes defines protocols for a family $\mathbb{G} = \{G_\mathfrak{p}\}$, i.e., a set of finite cyclic groups indexed by $\mathfrak{p}$ [Bon98]. Then there is an *instance generator* that, taken a random integer, outputs some random $\mathfrak{p}$ and a generator $g$ of $G_\mathfrak{p}$. For simplicity, we assume a fixed $G$ in describing our scheme.

This holds in particular for $j = 0$.

The $\lambda$-values are independent of $f(T)$, and can be computed from the formulas for Lagrange interpolation.

To denote that some shares $x_1, \ldots, x_n$ share a secret $x_0$ in a $(n, k)$ threshold secret sharing scheme, we also write

$$(x_1, \ldots, x_n) \leftrightarrow^{(k,n)} x_0 \mod q.$$

### 3.5.2 The Scheme DPCC (Distributed Pseudorandom Common Coin)

At a high level, our scheme works as follows. The value of a coin $C$ is obtained by first hashing $C$ to obtain $\tilde{g} \in G$, then raising $\tilde{g}$ to a secret exponent $x_0 \in \mathbb{Z}_q$ to obtain $\tilde{g}_0 \in G$, and finally hashing $\tilde{g}_0$ to obtain the value $F(C) \in \{0, 1\}$. The secret exponent $x_0$ is distributed among the parties using Shamir's secret sharing scheme [Sha79]. Each party $P_i$ holds a share $x_i$ of $x_0$; its share of $F(C)$ is $\tilde{g}^{x_i}$, along with a "validity proof." Shares of coin $C$ can then be combined to obtain $\tilde{g}_0$ by interpolation "in the exponent."

In more detail, we need cryptographic hash functions

$$\begin{aligned} H &: \{0,1\}^* \to G; \\ H' &: G^6 \to \mathbb{Z}_q; \\ H'' &: G \to \{0,1\}. \end{aligned}$$

No specific requirements are made for these hash functions, but they will be modeled as random oracles in the analysis.

**The dealing phase.** In the dealing phase, the dealer selects group $G$, and thus implicitly $q$, and $k$ coefficients of a random polynomial $f(T)$ over $\mathbb{Z}_q$ of degree less than $k$ and a random generator $g$ of $G$. For $0 \leq i \leq n$, let $x_i = f(i)$ and $g_i = g^{x_i}$. Party $P_i$'s secret key $SK_i$ is $x_i$, and his verification key $VK_i$ is $g_i$. The global verification key $VK$ consists of a description of $G$ (which includes $q$) and $g$.

**Share generation.** For a coin with name $C \in \{0, 1\}^*$, we let $\tilde{g} = H(C)$, and $\tilde{g}_i = \tilde{g}^{x_i}$ for $0 \leq i \leq n$. The value of the coin is $F(C) = H''(\tilde{g}_0)$.

For a given coin $C$, party $P_i$'s share of the coin is $\tilde{g}_i$, together with a "validity proof," i.e., a proof that $\log_{\tilde{g}} \tilde{g}_i = \log_g g_i$. This proof is the well-known interactive proof of equality of discrete logarithms (see [CEvdGP87, CP92]), collapsed into a non-interactive proof using the Fiat-Shamir heuristic [FS87]. A valid proof is a pair $(c, z) \in \mathbb{Z}_q \times \mathbb{Z}_q$, such that

$$c = H'(g, g_i, h, \tilde{g}, \tilde{g}_i, \tilde{h}), \tag{3.1}$$

where

$$h = g^z / g_i^c \text{ and } \tilde{h} = \tilde{g}^z / \tilde{g}_i^c.$$

Party $P_i$ computes such a proof by choosing $s \in \mathbb{Z}_q$ at random, computing $h = g^s$, $\tilde{h} = \tilde{g}^s$, and obtaining $c$ as in (3.1) and $z = s + x_i c$.

**Share combination.** Using Shamirs secret sharing scheme [Sha79], for any set $S$ of $k$ distinct points in $\mathbb{Z}_q$, and any $j \in \mathbb{Z}_q$, there exist elements $\lambda^S_{i,j} \in \mathbb{Z}_q$ for $i \in S$, such that

$$\sum_{i \in S} f(i) \lambda^S_{i,j} = f(j).$$

Combining shares essentially applies the same formula in the exponent of $g$. To combine a set of valid shares $\{\tilde{g}_i : i \in S\}$, one simply computes

$$\tilde{g}_0 = \tilde{g}^{x_0} = \tilde{g}^{\sum_{i \in S} f(i) \lambda^S_{i,0}} = \prod_{i \in S} \tilde{g}^{x_i \cdot \lambda^S_{i,0}} = \prod_{i \in S} \tilde{g}_i^{\lambda^S_{i,0}}.$$

The value of the coin is then computed as $H''(\tilde{g}_0)$.

### 3.5.3 Security Analysis

To analyze this scheme, we need to consider two intractability assumptions.

**Definition 3.3.** For $g, g_0, \hat{g} \in G$, define $\mathit{DH}(g, g_0, \hat{g})$ to be $\hat{g}_0 = \hat{g}^{x_0}$, provided that $g_0 = g^{x_0}$.

Also, define $\mathit{DHP}(g, g_0, \hat{g}, \hat{g}_0)$ to be 1 if $\hat{g}_0 = \mathit{DH}(g, g_0, \hat{g})$, and 0 otherwise.

The Computational Diffie-Hellman (CDH) assumption [Bon98] is the assumption that *DH* is hard to compute—that is, there is no efficient, probabilistic algorithm that computes *DH* correctly (with negligible error probability) on a random input.

The Decisional Diffie-Hellman (DDH) assumption is the assumption that *DHP* is hard to compute—that is, there is no efficient, probabilistic algorithm that computes *DHP* correctly (with negligible error probability) on a random input.

With these assumptions, we now state the following theorem:

**Theorem 3.2.** *In the random oracle model, the coin-tossing scheme DPCC is secure under the CDH assumption, if $k = t + 1$, and under the DDH assumption if $k > t + 1$.*

*Proof.* We need to show *robustness* and *unpredictability.*

The robustness of the scheme follows from the soundness of the interactive proof of equality of discrete logarithms [CP92], and the fact that in the random oracle model, the challenges $c$ are the output of the random oracle $H'$.

To prove unpredictability, we assume we have an adversary $\mathcal{A}$ that can predict a coin with non-negligible probability, and show how to use this adversary to efficiently compute *DH* (if $k = t+1$) or *DHP* (if $k > t+1$).

To this end, we design a simulator that provides the adversary with a view that is indistinguishable from a real run of the protocol; however, the simulator embeds an instance of the Diffie-Hellman (or the Decisional Diffie Hellman) problem into the simulated view. If the adversary can break the unpredictability property or distinguish the simulated view from a real protocol run, its output is sufficient to have an advantage in solving the embeded problem.

We first make a few simplifying assumptions:

– $\mathcal{A}$ corrupts parties $P_{k-t}, \ldots, P_{k-1}$; this does not restrict it at all, as we could give the parties numbers only after the adversary has chosen its corruptions. This assumption is made only to make our notations simpler.

– Before $\mathcal{A}$ requests the share of a coin or predicts a coin, it has already evaluated $H$ at that coin's name. This does not restrict the adversary at all; honest parties can w.l.o.g. query the oracle before giving out a coin share. Furthermore, if the adversary does not query the random oracle before guessing the coin, it will guess a completely random coin whose value does not help it at all.

– $\mathcal{A}$ evaluates $H$ successively at distinct points $C_1, \ldots, C_l$, where $l$ is a bound that is fixed for a given adversary and security parameter.

We denote the "target" coin, which $\mathcal{A}$ attempts to predict, by $\hat{C}$, and we let $\hat{g} = H(\hat{C})$, and $\hat{g}_i = \hat{g}^{x_i}$ for $0 \leq i \leq n$.

We may guess that $\hat{C}$ is equal to $C_s$, where $s$ is randomly chosen from $\{1, \ldots, l\}$. Should $\mathcal{A}$ make $k - t$ requests to reveal shares of $\hat{C}$, we simply stop the game. As $s$ is randomly chosen, the expected number of games we have to play until we guess the right target coin is $l$. Therefore, this assumption decreases the advantage of our construction relative to $\mathcal{A}$ by a factor of $l$.

By the assumption that the adversary queries the random oracle $H$ at $C$ before requesting the first coin shares, we obtain $\hat{C}$ and recognize it as such (i.e., as the $s$th coin name in the list of queries) before the adversary demands any coin shares.

*Case 1:* $k = t + 1$. Here is how we use this adversary to compute *DH*. To break the scheme, it is sufficient to construct an algorithm that on random inputs $g, g_0, \hat{g} \in G$, outputs a list of group elements that contains $\hat{g}_0 = DH(g, g_0, \hat{g})$ with non-negligible probability.

We simulate the $\mathcal{A}$'s interaction with the coin-tossing scheme as follows. By our simplifying assumption, the adversary corrupts parties $P_1, \ldots, P_t$. As the notation suggests, we use the given value $g$ in the global verification key. We choose $x_1, \ldots, x_t \in \mathbb{Z}_q$ at random, set $S = \{0, 1, \ldots, t\}$, compute $g_i = g^{x_i}$ for $1 \leq i \leq t$, and let for $t + 1 \leq i \leq n$

$$g_i = \prod_{j=0}^{k-1} g_j^{\lambda^S_{j,i}},$$

i.e., we compute the verification keys $VK_i$ of the honest parties from the shares (more precisely, from the verification keys derived from the shares) of the corrupted parties and the input value $g_0$.

The distibution of these keys is exactly the same the dealer would create by choosing a the random polynomial $f(T)$.

In the random oracle model, the adversary explicitly queries the random oracles $H, H', H''$. The simulator we are building is responsible for the operation of these oracles—it sees the queries made by the adversary, and is free to respond as it wishes so long as its responses are consistent and correctly distributed.

If $\mathcal{A}$ queries $H$ at point $\hat{C}$ (whatever $\hat{C}$ turns out to be), i.e., at the target coin, we use the given $\hat{g}$ as the output value of $H$.

For all other queries $C \neq \hat{C}$, we choose $r \in \mathbb{Z}_q$ at random and compute $\tilde{g} = g^r$.

The simulator uses this $\tilde{g}$ as the value of $H$ at $C$. We then compute the shares $\tilde{g}_i = g_i^r$ for $t + 1 \leq i \leq n$. As $r \in \mathbb{Z}_q$ is random, $g^r$ is a random group element of $G$, which implies that the $g^r$ chosen here have the same distribution as the output of $H$ has.

The validity proofs can be simulated using standard zero-knowledge techniques [GMR89], using the random oracle to make them non-intercative [BR95].

For the target coin $\hat{C}$, we only need the shares of the corrupted parties we computed before, and we never have to compute any shares for honest parties, since $k = t + 1$. When the adversary terminates, we simply output the list of queries made by the adversary to the oracle $H''$.

It is easily verified that the above simulation is nearly perfect: the adversary's view has precisely the same distribution as in the actual interaction (but there is actually a negligible probability that the zero-knowledge simulations fail).

Observe that because the adversary has a non-negligible advantage in predicting the value of the coin $\hat{C}$, he must evaluate $H''$ at the corresponding point $\hat{g}_0$ with non-negligible probability. Therefore, it must have a non-negligible chance to guess $\hat{g}_0$, which completes the proof of Theorem 3.2 for Case 1.

*Case 2:* $k > t + 1$. The above simulation does not work in this case because we would have to simulate the shares of the coin $\hat{C}$ from up to $k - t - 1 > 0$ honest parties. Moreover, we cannot view these honest parties as fixed: the adversary may adaptively select which honest parties contribute shares of the target coin. More precisely, the simulator has to decide at the setup for which $k - t - 1$ honest parties it selects the corresponding $x_i$ and computes the verification keys $VK_i$, and for which $n - (k - 1)$ parties the verification keys are derived from those keys and the value $g_0$. To use a guess for the target coin to compute $\hat{g}^{x_0}$, the simulator has to supply $\mathcal{A}$ with $k - t - 1$ coin shares $\hat{g}_i^x$ from honest parties. It can only compute these for the parties selected at the beginning while computing the verification keys; the adversary is free however to request the shares from any party it chooses. Thus, the probability that the simulator guesses right vanishes exponentially in $k - t$, which implies that the number of times the simulator has to restart the game are exponential in $k - t$. Thus, unless $k - t$ is bounded by the logarithm of the security parameter $\kappa$, the simulator does not run in polynomial time anymore and is thus worthless.

To deal with this problem, we take a dual approach. Instead of the real shares, the simulator uses some random values. If the adversary can't distinguish them from the real shares, we can use the simulator as above. If the adversary can, i.e., it can break the scheme in the real world but not in the simulation, then we can use this adversary to distinguish a set of valid shares from a set of random values. This can be transformed into computing *DHP*, i.e., the adversary would violate the Decisional Diffie-Hellman assumption.

It is sufficient [Sta96, NR97] to construct a statistical test that distinguishes between the following two distributions:

**D:** the set of tuples

$$(g, g_0, \ldots, g_{k-t-1}, \hat{g}, \hat{g}_0, \ldots, \hat{g}_{k-t-1}),$$

where $g, g_0, \ldots, g_{k-t-1} \in G$ are random, and $\hat{g} = g^r, \hat{g}_0 = g_0^r, \ldots, \hat{g}_{k-t-1} = g_{k-t-1}^r$ for randomly chosen $r \in \mathbb{Z}_q$; and

**R:** the set of tuples

$$(g, g_0, \ldots, g_{k-t-1}, \hat{g}, \hat{g}_0, \ldots, \hat{g}_{k-t-1}),$$

where $g, g_0, \ldots, g_{k-t-1}, \hat{g}_0, \ldots, \hat{g}_{k-t-1} \in G$ are random.

The first set of tuples corresponds to a real run of the protocol; the probability distribution of those tuples and the ones generated in a protocol run is identically.

The second set of tuples is the one we can fake with our simulator. The probability distribution of this tuple differs from the one generated in a protocol run. Every adversary that can find the difference (i.e., which can break the unpredictability property of our scheme, but not help the simulator solve the DH), can itself be used to solve the DDH. This is sufficient for our proof: any adversary that can break the unpredictability property of the coin can be used one way or the other to solve a problem which is assumed to be hard.

Our statistical test works as follows. Let

$$(g, g_0, \ldots, g_{k-t-1}, \hat{g}, \hat{g}_0, \ldots, \hat{g}_{k-t-1})$$

be the input "test" tuple. We simulate the adversary's interaction with the coin-tossing scheme as follows. By our simplifying assumption, the adversary corrupts $P_{k-t}, \ldots, P_{k-1}$. As the notation suggests, we simulate the dealer by using the given $g$ in the global verification key, and $g_1, \ldots, g_{k-t-1}$ in the local verification keys for $P_1, \ldots, P_{k-t-1}$. We choose the secret keys $x_{k-t}, \ldots, x_{k-1} \in \mathbb{Z}_q$ at random and set $S = \{0, 1, \ldots, k-1\}$; for $k-t \leq i \leq k-1$, compute $g_i = g^{x_i}$, and for $k \leq i \leq n$, let

$$g_i = \prod_{j=0}^{k-1} g_j^{\lambda_{j,i}^S}.$$

Also, we will use the given $\hat{g}$ as the output of $H$ at $\hat{C}$, and the given $\hat{g}_1, \ldots, \hat{g}_{k-t-1}$ as the corresponding shares of $\hat{C}$ from parties $P_1, \ldots, P_{k-t-1}$. We will use the given $\hat{g}_0$ to compute the shares of $\hat{C}$ from the other honest parties as follows: for $k-t \leq i \leq k-1$, set $\hat{g}_i = \hat{g}^{x_i}$, and for $k \leq i \leq n$, compute

$$\hat{g}_i = \prod_{j=0}^{k-1} \hat{g}_j^{\lambda_{j,i}^S}.$$

Whenever the adversary requests a share of $\hat{C}$ for an honest party $P_i$, we give the adversary $\hat{g}_i$ as computed above.

We reveal the shares of a coin $C \neq \hat{C}$ just as in Case 1: we choose $r \in \mathbb{Z}_q$ at random, and compute $\tilde{g} = g^r$ and $\tilde{g}_i = g_i^r$ for all $1 \leq i < k-t$ and $k \leq i \leq n$. Like in Case 1, the distribution is perfect.

For both target and non-target coins, we construct simulated proofs of correctness just as in Case 1.

At the end of the adversary's interaction, when the adversary makes a prediction $b \in \{0, 1\}$ for the value of coin $\hat{C}$, we output $X = 1$ if $b = H''(\hat{g}_0)$, and $X = 0$ otherwise.

We claim that this algorithm is an effective statistical test distinguishing **D** from **R**.

Observe that if the test tuple comes from **D**, the above simulation is nearly perfect (the only imperfection lying in the zero knowledge simulations, which fail with negligible probability), and so the probability that $X = 1$ is essentially the adversary's advantage, which differs from $1/2$ by a non-negligible amount.

Therefore, it will suffice to show that if the test tuple comes from **R**, the probability that $X = 1$ differs from $1/2$ by a negligible amount. But this follows from the observation that for any sequence of distinct indices $i_1, \ldots, i_{k-t-1}$ belonging to honest parties, the group elements

$$\hat{g}_0, \hat{g}_{i_1}, \ldots, \hat{g}_{i_{k-t-1}}$$

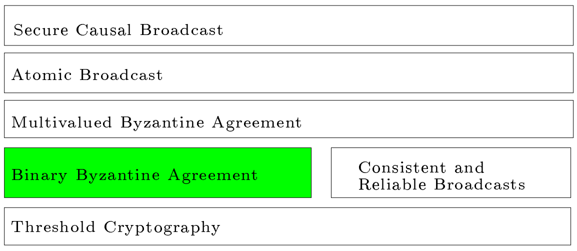
are independent and uniformly distributed. Thus, after revealing any $k - t - 1$ of the "shares" $\hat{g}_i$ belonging to honest parties, then conditioning on the adversary's view, the value of $\hat{g}_0$ is still random, and hence the probability that $X = 1$ in this case is essentially $1/2$.

□

This completes the proof of Theorem 3.2 for Case 2. Note that in this proof, we do not need to model $H''$ as a random oracle—we only need the property that for random $\hat{g}_0 \in G$, $H''(\hat{g}_0)$ has a nearly uniform distribution. For example, using the Entropy Smoothing Theorem [Lub96, Chapter 8], one could implement $H''$ as the inner product of the bit representation of $\hat{g}_0$ with a random bit string (chosen once and for all by the dealer). Also note that using the same proof technique, one could prove the unpredictability property using the threshold $k - f$ instead of $k - t$, where $f$ is the actual number of corrupted parties.

# Chapter 4

# Byzantine Agreement


Secure Causal Broadcast
Atomic Broadcast
Multivalued Byzantine Agreement
Binary Byzantine Agreement
Consistent and Reliable Broadcasts
Threshold Cryptography

This chapter applies our model to the well–known problem of Byzantine agreement. We provide a protocol that solves Byzantine agreement using the cryptographic tools presented in the previous chapter. Several modifications to the original protocol are introduced to increase flexibility and performance and make it more suitable for a real world implementation. Finally, an optimistic protocol is presented that combines the performance of timed protocols with the robustness of asynchronous ones.

## 4.1 Introduction

The (binary) Byzantine agreement problem is one of the fundamental problems in distributed fault-tolerant computing. In this problem, there are $n$ communicating parties, at most $t$ of which are corrupted. The goal is that all honest (i.e., uncorrupted) parties agree on one of two values, despite the malicious behavior of the corrupted parties.

Fischer, Lynch, and Paterson (FLP) [FLP85] have shown that no deterministic protocol can guarantee agreement even against benign failures in the asynchronous setting. The only way to overcome this limitation in the fully asynchronous model is to use randomized protocols, as suggested by Rabin [Rab83] and Ben-Or [BO83].

They assume a *common coin*, a random source observable by all participants but unpredictable for an adversary; this abstraction is used in most subsequent protocols for the asynchronous model.[1] Building on the cryptographic common coin and threshold signatures (see Chapter 3), we present protocol ABBA that solves Byzantine agreement in constant expected time with a communication complexity quadratic in the number of participants, which is essentially optimal.

To keep the essential ideas behind ABBA visible, the protocol has been kept relatively simple. Thus, in spite of the theoretical optimality of ABBA, many optimizations can be made to make it more practical, more flexible, faster and robust against some message loss. Section 4.7 presents protocol BASIL, which is

[1]In the literature of distributed systems, a common coin is sometimes called a *random oracle*. This is another unfortunate collision of vocabulary and has nothing to do with the cryptographic random oracle model

the fully optimized version of ABBA. A prototype of this protocol has also been implemented to prove the practicability of our approach not only on paper, but also provide real measurements.

To further optimize the performance, and to be able to compete with protocols that work in weaker models (like timed models and the crash failure model), we use the optimistic aspiration (see Section 2.6.2) to produce a combined deterministic/randomized protocol. The deterministic protocol is used to improve performance, while the randomized protocol is used as a backup to guarantee that the protocol terminates correctly.

**Related Work.** Since the problem of Byzantine agreement has been defined in 20 years ago [MPL80], a countless number of problem variations, models and approaches has appeared in the literature. For a general overview, see the survey of the early Byzantine era by Chor and Dwork [CD89] and the more recent account by Berman and Garay [BG93].

A fundamental result in this area is the impossibility result of Fischer, Lynch, and Paterson [FLP85] that rules out the existence of a deterministic protocol. This result led to the development of probabilistic protocols. The protocols of Rabin [Rab83] and Ben-Or [BO83] are the first probabilistic protocols to overcome this limitation. Bracha's protocol improves the resilience to the maximum $t < n/3$ [Bra84]. Numerous protocols (including ours) build on these [Tou84, FM97, BG93, CR93], but so far, all protocols are either inefficient or make additional assumptions on the environment (like a trusted dealer that is needed after a constant number of protocol invocations.

In mixed deterministic/randomized protocols, there are two different approaches in the literature. Goldreich and Petrank [GP90] designed a protocol for synchronous systems that uses randomization to increase performance. First a randomized protocol is started, terminating with an constant expected number of rounds. If it does not terminate within a given number of rounds, it is stopped and a deterministic (synchronous) protocol takes over. This way, the combined protocol has the high (expected) performance of a randomized protocol, while guaranteeing an upper bound on the running time. Later, Bar-Noy et al. used the same approach to combine three algorithms into one that is faster than the single algorithms [BNDDS92]. Zamsky [Zam96] optimized and combined both approaches to achieve optimality in both average and worst case.

Naturally, this approach works only in synchronous systems, as it relies on the deterministic protocol to work correctly.

In the failure-detector world, combined protocols are usually used to increase the robustness of the overall protocol. Dolev and Malkhi [DM94] use a combined randomized/ failure detector protocol to increase the resilience of crash failure protocols. Aguilera and Toueg [AT96] interleave a randomized and a failure detector protocol to guarantee the correctness of the protocol if either the failure detector or the common coin subprotocol does not satisfy its specifications, also in the crash failure model. Malkhi and Reiter [MR97] present a hybrid protocol for Byzantine failures along the lines of [AT96]. Again, the motivation is more to tolerate failure of one component (either the common coin or the failure detector), rather than optimizing for performance. Their protocol requires a rather strong failure detector, which requires properties that are rather difficult to meet in a practical system.

## 4.2 Problem Statement

We now define the operation and requirements of a Byzantine agreement protocol, in the context of our basic system model described in Chapter 2. There are $n$ parties, $P_1, \ldots, P_n$, and the adversary may corrupt up to $t$ of them, where $3t < n$.

As mentioned in Chapter 2, we want an agreement protocol that can be used to implement a *transaction processing service*. To this end, we assume that each decision to be made is associated with a unique transaction identifier *ID*, chosen by the adversary. If the adversary chooses the same transaction identifier twice, an honest party will treat the corresponding transactions as the same and simply ignore the surplus messages.

After initializing the protocol instance $ID$, the adversary may deliver a message to $P_i$ of the form

$$(ID, \texttt{in}, \texttt{propose}, V_i),$$

where $V_i$ in $\{0, 1\}$ is the initial value of party $P_i$.

We do not require the adversary to activate all parties on all transactions; however, we will define the problem in a way that termination of a certain transaction is only required if all honest parties have been activated.

Upon decision, a party outputs a message of the form

$$(ID, \texttt{out}, \texttt{decide}, \textit{decision-value}),$$

where *decision-value* $\in \{0, 1\}$. In the latter case, we say that $P_i$ *decides decision-value* for $ID$. We require that $P_i$ makes a decision for a given $ID$ at most once. However, the adversary may continue to deliver messages involving $ID$ after $P_i$ has made a decision for $ID$, and $P_i$ might still react on these messages.

The three basic properties that an agreement protocol must satisfy are *agreement*, *termination*, and *validity*.

**Agreement.** Any two honest parties that decide a value for a particular $ID$ must decide the same value. More precisely, it is computationally *infeasible* for an adversary to make two honest parties decide on different values.

**Validity.** If all honest parties have been activated on a given $ID$ with the same initial value, then all honest parties that decide must decide this value.

This is the usual definition of validity in the literature. In Sections 4.9.1 and 6.2, we discuss weaker notions of validity that may be more appropriate for particular applications. For instance, initial values may come with validating data (e.g., a digital signature) that establishes the "validity" of a value in a particular context. One could then simply require that an honest party may only decide on a value for which it has the accompanying validating data—even if all honest parties start with 0, they may still decide on 1 if they obtain the corresponding validating data for 1 during the agreement protocol.

**Termination.** The traditional approach in the literature is to assume that all messages between honest parties are "eventually" delivered, and then to define the termination condition to be that all honest parties "eventually" decide (with probability 1). As discussed in Section 2.4, in formalizing these definitions, one considers infinite runs of a protocol; however, in the computationally bounded setting, this simply does not work.

According to the general transformation in Section 2.4, we present here a workable definition in our setting that captures the intuition that *to the extent* the adversary delivers messages among honest parties, the honest parties *quickly* decide. Although the intuition is fairly clear, one has to be careful with the details. For us, *termination* consists of two conditions: *deadlock freeness* and *efficiency*.

*Deadlock freeness.* It is infeasible for the adversary to create a situation where for some $ID$ there are some honest parties who are not decided, yet all honest parties have been activated on $ID$, and all messages relating to this $ID$ generated by honest parties have been delivered.

*Efficiency.* For every $ID$, the communication complexity for $ID$ is probabilistically polynomially bounded.

The *deadlock freeness* property rules out trivial protocols that never decide and never generate any messages to be delivered. The *efficiency* property ensures timely convergence, provided the adversary delivers all messages between honest parties.

## 4.3 The Protocol ABBA (Asynchronous Binary Byzantine Agreement)

### 4.3.1 Protocol ABBA

We now present our protocol ABBA, which stands for Asynchronous Binary Byzantine Agreement; this protocol is also described in [CKS00]. As usual there are $n$ parties $P_1, \ldots, P_n$, up to $t, t < \frac{n}{3}$ of which may be corrupted by the adversary.

The protocol uses an $(n, n-t, t)$ threshold signature scheme (see Section 3.3), as well as an $(n, n-t, t)$ threshold coin-tossing scheme (see Section 3.4). Let $F(C)$ denote the value of coin with name $C$, where $C$ is an arbitrary bitstring.

For a given transaction identifier $ID$, party $P_i$ is initialized with an initial value $V_i \in \{0, 1\}$, and the protocol proceeds in asynchronous protocol rounds $r = 1, 2, \ldots$, which are defined by the protocol logic. Each round contains four basic steps that are sketched below:

0. Each party sends its initial value along with a corresponding signature share to all other parties. On receiving $n - t$ such votes, each party combines the signature shares of the value with the simple majority (i.e., at least $t + 1$ votes) to a threshold signature. This value will be the value used in the first pre-vote. To implement this, we shall need an $(n, t + 1, t)$ threshold signature scheme in addition to the threshold signature scheme used later in the protocol. Note that this step is not necessary if the input values are generated by some authenticated source or a weaker form of validity (see Section 4.9.1) suffices.

1. Each party casts a *pre-vote* for a value $b \in \{0, 1\}$. These pre-votes must be *justified* by an appropriate threshold signature, and must additionally be accompanied by a valid signature share on the pre-vote message. In round one the value of this vote is determined by step 1; later, the value of the pre–vote is either the (unique) value of a main vote from step 4 in the previous round, or the value of the coin if all main votes received are abstain.

2. After collecting $n - t$ valid pre-votes, each party casts a *main-vote* $v \in \{0, 1, \texttt{abstain}\}$. As with pre-votes, these main-votes must be justified by an appropriate threshold signature, and must be accompanied by a valid signature share on an appropriate message. The value of this vote is 0 (or 1) if all received pre-votes where for 0 (or 1), or `abstain` if pre-votes for different values have been received.

3. After collecting $n - t$ valid main-votes, each party examines these votes. If all votes are for a value $b \in \{0, 1\}$, then the party *decides* $b$ for $ID$, but continues to participate in the protocol for one more round. Otherwise, the party simply proceeds to the next round.

4. The value of coin $\text{¢}_r = F(ID, r)$ is revealed, which may be used in steps 1–3.

We next introduce some notation and concepts used in the description of the protocol below.

Recall that a message from $P_i$ to $P_j$ has the form $(ID, i, j, payload)$, so that in specifying a message, we will only specify the *payload*; the values of $ID$, $i$, and $j$ are implied from the context.

Each vote message has to contain a *justification*, which consists of threshold signatures on collected votes from previous votings.

In round $r = 1$, party $P_i$'s pre-vote is the majority of the "pre-processing votes". There must be at least $t + 1$ votes for the same value $b \in \{0, 1\}$ (although this $b$ might not be unique if $n > 3t + 1$). For the justification, a party selects $t + 1$ such votes, and combines the signature shares to obtain an $(n, t + 1, t)$ threshold signature on the string

$$(\texttt{pre-process}, ID, b).$$

In rounds $r > 1$, a pre-vote for $b$ may be justified in two ways:

– either with a threshold signature on the string

$$(\texttt{pre-vote}, ID, r-1, b);$$

we call this a *hard* pre-vote for $b$;

– or with a threshold signature on the string

$$(\texttt{main-vote}, ID, r-1, \texttt{abstain})$$

for the pre-vote $b = F(ID, r)$; we call this a *soft* pre-vote for $b$.

Intuitively, a hard pre-vote expresses $P_i$'s preference for $b$ based on evidence for preference $b$ in round $r-1$, whereas a soft pre-vote is just a vote for the current value of the coin, based evidence of conflicting votes in round $r-1$. The threshold signatures are obtained from the computations in previous rounds (see the description of step 4 below). We assume that the justification indicates whether the pre-vote is hard or soft.

A main-vote $v$ is one of the values $\{0, 1, \texttt{abstain}\}$ and, like pre-votes, accompanied by a *justification*. The main-vote is justified as follows.

– If among the $n-t$ pre-votes just collected by $P_i$ there is a pre-vote for 0 and a pre-vote for 1, then $P_i$'s main-vote $v$ for round $r$ is `abstain`. The justification for this main-vote consists of the justifications for the two conflicting pre-votes.

– Otherwise, $P_i$ has collected $n-t$ justified pre-votes for some $b \in \{0, 1\}$, and since each of these comes with a valid signature share on the string

$$(\texttt{pre-vote}, ID, r, b),$$

party $P_i$ can combine these shares to obtain a valid threshold signature on this string. Party $P_i$'s main-vote $v$ in this case is $b$, and its justification is this threshold signature.

Figure 4.1 shows how the justifications are generated; verification is done similary.

```
Subprotocol Generate justification
  input: message $m$, round number $r$.
  if $r = 1$ and $m$ is a pre-vote for $\rho$,
        $J \leftarrow$ a $(t+1)$ threshold signature on pre-process messages for $\rho$.
  if $r > 1$ and $m$ is a hard pre-vote on $\rho$,
        $J \leftarrow$ a justification for a main-votes on $\rho$ from round $r-1$
  if $r > 1$ and $m$ is a soft pre-vote,
        $J \leftarrow$ a threshold signature on $n-t$ main votes for abstention in round $r-1$.
  if $m$ is a main-vote on $\rho$,
        $J \leftarrow$ a threshold signature on $n-t$ pre-votes on $\rho$ from round $r$
  if $m$ is a main vote on abstain,
        $J \leftarrow$ one justification on a pre-vote for 1 and one justification on a pre-vote for 0 in round $r$.
```

Figure 4.1: Subprotocol GenJust (Generate Justification) of ABBA

The complete protocol is shown in Figures 4.1 and 4.2.

### 4.3.2 Analysis

**Theorem 4.1.** *Assuming a secure threshold signature scheme, a secure threshold coin-tossing scheme, protocol ABBA solves asynchronous Byzantine agreement for $n > 3t$.*

```
Protocol ABBA for party $P_i$ with initial value $V_i$.
  upon receiving a message $(ID, \texttt{in}, \texttt{propose}, V_i)$,

        /* Step 0: Pre-Round. */

        Generate a signature share $sh_i^{\texttt{step0}}$ of the $(n, t+1, t)$ threshold signature on the string
                                    $(ID, \texttt{pre-process}, V_i)$
        and send to all parties a message of the form
                                    $(ID, \texttt{pre-process}, V_i, sh_i^{\texttt{step0}})$
        .
        Collect $n-t$ pre-processing messages. Let $b$ be the simple majority of the received pre-processing votes.
        let $r \leftarrow 1$
        repeat
            /* Step 1: Pre-Vote. */

            if $r > 1$, let
                $$b = \begin{cases} 0 & \text{if } P_i \text{ accepted a main vote } mv_{r-1,j} \text{ for 0,} \\ 1 & \text{if } P_i \text{ accepted a main vote } mv_{r-1,j} \text{ for 1,} \\ ¢_{r-1} & \text{if } P_i \text{ accepted } n-t \text{ abstentions.} \end{cases}$$
            Produce a signature-share $sh_i^{\texttt{pre}}$ on the string
                                    $(\texttt{pre-vote}, ID, r, b)$.
            Invoke subprotocol GenJust to produce the corresponding justification and send to all parties a
                message of the form
                                    $(\texttt{pre-vote}, r, b, \textit{justification}, sh_i^{\texttt{pre}})$.
            /* Step 2: Main-Vote. */

            wait for $n-t$ pre-vote messages $pv_{r,1}, \ldots, pvr, n-t$ are received and accepted. Let
                $$v = \begin{cases} 0 & \text{if } P_i \text{ accepted } n-t \text{ pre-votes } pv_{r,1}, \ldots, pvr, n-t \text{ on 0,} \\ 1 & \text{if } P_i \text{ accepted } n-t \text{ pre-votes } pv_{r,1}, \ldots, pvr, n-t \text{ on 1,} \\ \texttt{abstain} & \text{if } P_i \text{ accepted pre-votes for 0 and 1,} \end{cases}$$
            and produce the signature share $sh_i^{\texttt{main}}$ on the string
                                    $(\texttt{main-vote}, ID, r, v)$.
            Invoke subprotocol GenJust to produce the corresponding justification and send to all parties a
                message of the form
                                    $(\texttt{main-vote}, r, v, \textit{justification}, sh_i^{\texttt{main}})$.
            wait for $n-t$ main-votes $mv_{r,1}, \ldots, mv_{r,n-t}$ are received and accepted.
            /* Step 3: Check for decision. */

            on detecting $n-t$ main-votes for $\rho \in \{0,1\}$, output $(ID, \texttt{out}, \texttt{decide}, \rho)$
              Repeat steps 4 and 1–2 for one more round.
            /* Step 4: Common coin. */

            Generate a coin share $sh_i^{\texttt{coin}}$ of the coin $(ID, r)$, and send to all parties a message of the form
                                    $(\texttt{coin}, r, sh_i^{\texttt{coin}})$.
            Collect $n-t$ valid shares of the coin $(ID, r)$, and combine these shares to get the value $¢_r = F(ID, r) \in$
                $\{0, 1\}$.
            $r \leftarrow r + 1$
        until decided
```

Figure 4.2: Protocol ABBA (Asynchronous Binary Byzantine Agreement)

The rest of this section is devoted to the proof of this theorem. We have to show *validity*, *agreement*, and *termination*.

Step 0 in the protocol forces every party to obtain at least $t+1$ signature shares on the value it wants to propose in the first pre-vote. At least one of these must come of an honest party, thus a party can only pre-vote $\rho$ in the first pre-vote if it was the initial value from some honest party. If all honest parties were activated with $\rho$, then all acceptable pre-votes in the first round are for value $\rho$, thus all acceptable main-votes in that round are for $\rho$, as any other vote can not be justified. This results in a decision on $\rho$ in the first round by all honest parties that decide, and *validity* follows.

We fix a (given) $ID$. Consider the pre-votes cast by honest parties in round $r \geq 1$. Because $n > 3t$, there will be at most one value $b \in \{0,1\}$ that gathers at least $n-2t$ such pre-votes, and we define $\rho_r$ to be this value (if it exists), and otherwise we say that $\rho_r$ is undefined. We say that $\rho_r$ is defined at the point in the game at which sufficient pre-votes are cast.

**Lemma 4.2.** *For $r \geq 1$, the following holds (with all but negligible probability):*

*(a) if an honest party casts or accepts a main-vote of $b \in \{0,1\}$ in round $r$, then $\rho_r$ is defined and $\rho_r = b$;*

*(b) if an honest party casts or accepts a hard pre-vote for $b \in \{0,1\}$ in round $r+1$, then $\rho_r$ is defined and $\rho_r = b$;*

*(c) if an honest party casts or accepts a main-vote of* `abstain` *in round $r+1$, then $\rho_r$ is defined and $\rho_r = 1 - F(ID, r)$;*

*(d) if $r$ is the first round in which any honest party decides, then all honest parties that eventually decide, decide the same value in either round $r$ or $r+1$.*

*Proof.* To prove (a), suppose an honest party accepts a main-vote of $b \in \{0,1\}$ in round $r$. To be justified, this main-vote must be accompanied by a valid threshold signature on the message

$$(\texttt{pre-vote}, ID, r, b).$$

By the non-forgeability property of the signature scheme, this implies that at least $(n-t)-t = n-2t$ honest parties cast pre-votes for $b$. Thus, $\rho_r$ has been defined and is equal to $b$. That proves (a).

Part (b) now simply follows from the fact that a hard pre-vote for $b \in \{0,1\}$ in round $r+1$ is justified by the same threshold signature as the main-vote from round $r$ in part (a).

Now for part (c). A main vote of `abstain` in round $r+1$ must be accompanied by a justification for a pre-vote of 0 in round $r+1$ and a justification for pre-vote of 1 in round $r+1$. These pre-votes cannot both be soft pre-votes, and so one of these two pre-votes must be hard. It follows from (b) that this hard pre-vote must be for $\rho_r$, and hence the other pre-vote must be a soft pre-vote for $1-\rho_r$, and hence $F(ID, r) = 1 - \rho_r$. Part (c) now follows.

Now for part (d). Suppose some party $P_i$ decides $b \in \{0,1\}$ in some round $r$, and no party has decided in a previous round. Then in this round, $P_i$ accepted $n-t$ main-votes for $b$. By part (a), we must have $b = \rho_r$. So any other honest party who decides in round $r$ must also decide $\rho_r$.

Of the $n-t$ main-votes for $b$ that $P_i$ accepted, at least $n-2t$ came from honest parties who main-voted $b$, and since $n > 3t$, fewer than $(n-t)-t = n-2t$ signature shares on the message

$$(\texttt{main-vote}, r, \texttt{abstain})$$

have been or ever will be generated by honest parties. This in turn implies that a soft pre-vote in round $r+1$ cannot be justified. Thus, the only justifiable pre-votes in round $r+1$ are hard pre-votes, and by part (b), these must be hard pre-votes of $b$. Finally, this implies that the only justifiable main-votes in round $r+1$ are main-votes for $b$, and so all main-votes accepted by honest parties in round $r+1$ will be main-votes for $b$. □

*Agreement* follows from part (d). All that remains is *termination.* For this, we need to show *deadlock freeness* and *efficiency.*

*Deadlock freeness* is fairly straightforward. It is clear that honest parties will proceed from one round to the next, provided the adversary delivers all messages between the honest parties. The *deadlock freeness* property follows from this observation, along with part (d) of Lemma 4.2, and the fact that parties who decide play along for one more round.

All that remains is *efficiency.* Lemma 4.2 says that in a given round $r + 1$, for $r \geq 1$, the set of $n - t$ main-votes accepted by an honest party in step 4 does not contain votes for both 0 and 1. Also, an honest party will decide in this round unless it accepts at least one main-vote of `abstain`. But if it does accept an `abstain`, then $\rho_r = 1 - F(ID, r)$. The key to showing efficiency will be to show that *the value of* $\rho_r$ *is determined before the coin* $(ID, r)$ *is revealed.* Then the (unique) value of justifiable hard prevotes, which can be influenced by the adversary, is fixed before the (unique) value of the soft pre-votes, which depends on the coin and can not be influenced by the adversary is revealed. This results in a probability of $\frac{1}{2}$ that they are identical and all justifiable prevotes are identical, which causes the protocol to terminate.

The notion of $\rho_r$ being *determined* is different from the notion of $\rho$ being *defined* introduced above; if it is determined, then the value it will have once being defined is fixed, but it does not need to be defined yet. More precisely, by saying "$\rho_r$ is determined at a particular point in time," we mean the following: There is an efficient procedure $W$ that takes as input a transcript describing the adversary's interaction with the system up to the given point in time (i.e., the initialization values of the corrupt parties and all communication between parties), along with $ID$ and $r \geq 1$, and outputs $w \in \{0, 1, \texttt{?}\}$. Furthermore, if the output is $w \neq \texttt{?}$, then if $\rho_r$ ever becomes defined, it must be equal to $w$ (or at least, it should be computationally infeasible for an adversary to cause this not to happen).

By "the coin $(ID, r)$ is revealed at a particular point in time," we mean the point in time when an honest party generates the $(n - 2t)$-th share of the coin $(ID, r)$.

**Lemma 4.3.** *There is a function* $W$ *that determines* $\rho_r$*, as described above, such that for all* $r \geq 1$*, either* $\rho_r$ *is determined before coin* $(ID, r)$ *is revealed, or* $\rho_{r+1}$ *is determined before* $(ID, r + 1)$ *is revealed.*

*Proof.* Suppose an honest party $P_i$ is just about to generate the $(n - 2t)$-th share of coin $(ID, r)$ in step 4 of round $r$. Thus, there is a set $\mathcal{S}$ of at least $n - 2t$ honest parties who have also reached step 4 of round $r$; this set includes $P_i$, who is just about to release its share; all other members of $\mathcal{S}$ have already released their share. Almost all round $r + 1$ pre-votes for the parties in $\mathcal{S}$, as well as their justifications, are completely determined at this point, even if these votes have not actually been cast. The only exception are soft pre-votes, whose actual value is equal to $F(ID, r)$, which is not yet known.

If any party in $\mathcal{S}$ is going to cast a hard pre-vote for $b \in \{0, 1\}$, then by Part (b) of Lemma 4.2, $b$ is the only possible value for $\rho_r$. Thus, $\rho_r$ is already determined—in fact, it is already defined.

Otherwise, all parties in $\mathcal{S}$ are going to cast soft pre-votes, choosing the value $F(ID, r)$ as the value of their round $r+1$ pre-vote. It follows that the only possible value for $\rho_{r+1}$ is $F(ID, r)$. Therefore, immediately *after* $P_i$ reveals its share of coin $(ID, r)$, $\rho_{r+1}$ is determined. Moreover, the coin $(ID, r + 1)$ has not yet been revealed at this point, since fewer than $n - 2t$ honest parties have gone beyond step 2 of round $r + 1$. Thus, $\rho_{r+1}$ is determined before coin $(ID, r + 1)$ is revealed. □

This lemma, together with the unpredictability property of sequences of coins described in Section 3.4.3, implies that the probability that any honest party advances more than $2r + 1$ rounds is bounded by $2^{-r} + \epsilon$, where $\epsilon$ is negligible. *Efficiency* follows immediately. Note that to make the full reduction proof, we need to be able to explicitly "predict" (as in Section 3.4.3) the desired value of the coin $(1 - \rho_r)$ that would delay termination, which is why we defined the notion of "determining" $\rho_r$ as we did.

We remark that a party that has decided cannot completely stop the protocol, as it might still be needed to help other parties deciding. Thus, it has to wait for other parties to move on for one more round, which might never happen. We will discuss a more "decisive" termination in Section 4.4.1.

After the publication of this protocol, an automated verification of ABBA using model checking has been published by Kwiatkowska and Norman [KN01].

### 4.3.3 Performance

From Lemmas 4.2 and 4.3, we can conclude that ABBA has a probability of 0.5 of terminating every other round against a worst case adversary; thus, the expected number of protocol rounds is constant, and the expected communication complexity is in $O(n^2)$.

So far, the description and analysis of the protocol ABBA has concentrated on the main ideas of the protocol, and the protocol has been presented in a very basic form.

There are several improvements to the basic protocol to optimize various properties. The rest of this chapter will be devoted to optimize ABBA in various ways.

As we have quite some modifications to the basic protocol, all of which have some impact on the analysis, Section 4.7 will present the protocol BASIL, which is a version of ABBA with most modifications included. For completeness, this protocol will then again be fully analysed.

## 4.4 Loose Strings: Variations and Optimizations

Although we have striven to make our protocol as efficient as possible, we have omitted several optimizations in order to simplify the presentation.

This section collects some of the loose ends left open in the previous section, and deals with some aspects important for a real world implementation of ABBA.

### 4.4.1 Achieving Real Termination

If the first honest parties to decide make their decision in round $r$, there may be others who make their decision in round $r+1$. The "early deciders" play along for round $r+1$, which allows the "late deciders" to decide. However, the "late deciders" do not "know" they are "late," so they attempt to play along for round $r+2$. What happens is that in round $r+2$, the protocol will "fizzle out": the "late deciders" will simply end up waiting in step 2 for $n-t$ messages that never arrive. This "fizzling out" does indeed satisfy our technical definition of termination, and is perhaps adequate for some settings; however, a minor modification of the protocol allows parties to completely terminate after decision. We call this property *real termination.*

**Definition 4.1 (Real termination).** It is infeasible for the adversary to create a situation where for some *ID* there are some honest parties who are still executing a **wait for** operation, yet all honest parties have been activated on *ID*, and all messages relating to this *ID* generated by honest parties have been delivered.

When a party $P_i$ decides $b$ for *ID* in round $r$, it can combine the signature shares that it has on hand to construct a threshold signature on the message

$$(\texttt{main-vote}, ID, r, b).$$

It then sends this threshold signature to all parties and stops. Thus, $P_i$ can effectively erase all data in its internal state relevant to *ID*, and ignore all future incoming messages relating to *ID*. Any other party that is waiting for some other message, but instead receives the above threshold signature, can also decide $b$ for *ID*, send this signature to all parties, and then stop.

Without this modification, the threshold signatures on main-votes other than `abstain` are actually not used by the protocol, and could be deleted.

**Modification of the proof.** The proposed modification has no influence on the termination properties — parties may only terminate earlier — but might influence validity and agreement.

For validity and agreement, recall that a threshold signature on $n-t$ main-votes for $\rho$ can only be obtained if at least $t+1$ honest parties issued a signature share on a main vote for $\rho$. Thus, $t+1$ honest parties main-voted for $\rho$, which will cause every party that reaches the following pre-vote to get one of these votes and thus the only acceptable pre-vote $\rho$, forcing the decision value to be $\rho$. Thus, the modification just allows a party to decide a value it would have decided at most one round later anyway, thus validity and agreement are preserved.

Besides a cleaner termination and some memory savings, real termination has another advantage: the private key share of the threshold signature scheme is no longer needed once the final message is generated. This is important for proactive systems; these systems compute new shares and delete the old ones. This renders old shares that leaked to the adversary useless [CGHN97]

### 4.4.2 Using an $(n, t+1, t)$ Coin-Tossing Scheme

Instead of an $(n, n-t, t)$ coin-tossing scheme, one could use an $(n, t+1, t)$ coin-tossing scheme, provided that before a party releases its share of a coin, it sends an appropriate "ready" message to all parties, and waits for $n-t$ corresponding "ready" messages from other parties. These "ready" messages do not need to be signed—the authenticity of the messages is enough. This modification increases the communication complexity of the protocol; however, an $(n, t+1, t)$ coin can be implemented based on weaker intractability assumptions than an $(n, n-t, t)$ coin, i.e., Diffie-Hellman instead of decisional Diffie Hellman. This does not buy us much as long as the threshold signatures used are based on the stronger assumption, but there are more degrees of freedom for the signatures; instead of the scheme proposed in [Sho00], the signature scheme presented in Section 3.3.4 can be used to trade communication for computation, and for a smaller number of participants, sets of standard public key signatures can be appropriate as a threshold signature scheme. Finally, in a non-Byzantine setting with crash failures only, signatures can be omitted completely. If sets of signatures are used for threshold signatures, it is possible to combine the ready-message with the $(n, n-t, t)$ coin. Then it is possible to tolerate an additional $t$ passive corruptions or —in conjunction with the hybrid crash/Byzantine failure model— exclusion failures.

**Modification of the proof.** The only influence this modification has is on the time the adversary can predict a coin; thus, *validity* and *agreement* remain unaltered. The important point is to reprove lemma 4.3.

**Lemma 4.3.** There is a function $W$ that determines $\rho_r$, as described above, such that for all $r \geq 1$, either $\rho_r$ is determined before coin $(ID, r)$ is revealed, or $\rho_{r+1}$ is determined before $(ID, r+1)$ is revealed.

*Proof.* Suppose an honest party $P_i$ is just about to generate the first share of coin $(ID, r)$ in step 4 of round $r$. As such, there is a set $\mathcal{S}$ of at least $n-2t$ honest parties who have also send the corresponding ready message, i.e., who have received $n-t$ main votes in round $r$. Almost all round $r+1$ pre-votes for the parties in $\mathcal{S}$, as well as their justifications, are completely determined at this point, even if these votes have not actually been cast. The only exception are soft pre-votes, whose actual value is equal to $F(ID, r)$, which is not yet known.

If any party in $\mathcal{S}$ is going to cast a hard pre-vote for $b \in \{0, 1\}$, then by Part (b) of Lemma 4.2, $b$ is the only possible value for $\rho_r$. Thus, $\rho_r$ is already determined—in fact, it is already defined.

Otherwise, all parties in $\mathcal{S}$ are going to cast soft pre-votes, choosing the value $F(ID, r)$ as the value of their round $r+1$ pre-vote. It follows that the only possible value for $\rho_{r+1}$ is $F(ID, r)$. Therefore, immediately *after* $P_i$ reveals its share of coin $(ID, r)$, $\rho_{r+1}$ is determined. Moreover, the coin $(ID, r+1)$ has not yet been revealed at this point, since fewer than $n-2t$ honest parties have gone beyond step 2 of round $r+1$. Thus, $\rho_{r+1}$ is determined before coin $(ID, r+1)$ is revealed. □

### 4.4.3 Biased Agreement

For some applications, it is useful to have a Byzantine agreement protocol that is biased towards one value. One example is the multivalued Byzantine agreement in Chapter 6, where binary Byzantine agreement is used to decide whether a proposal should be accepted or rejected. The performance of this protocol benefits greatly from a Byzantine agreement protocol that, while maintaining the strong validity condition, tries to favor an accepting vote (in our case, a vote for 1) to a rejecting one.

We propose two modifications to the standard ABBA protocol to bias it towards one value; for simplicity assume this value is the value 1.

**Biased Coin**

To bias the coin, the coin tossing protocol is modified such that the first coin tossed in the protocol always yields the value 1; no coin message needs to be send or received for that coin. For all other rounds, the coin tossing remains unchanged. We call this coin a `1` biased coin.

**Lemma 4.4.** *Using a `1` biased coin, ABBA will output `1` if at least $t+1$ parties enter the first pre-vote voting for `1`.*

*Proof.* If at least $t+1$ parties enter the first pre-vote voting for 1, then all parties see at least one vote for 1 in their set of $n-t$ pre-votes. This makes it impossible to vote 0 in the first main vote; all valid main votes are either 1 or abstain. Thus, in the next rounds pre-vote, every party has seen either a main-vote for 1, or it will toss the coin, which also is one. In either case, all legal main votes in this round are votes for 1, causing all honest parties to decide 1 in that round. □

**Biased Pre-Protocol**

The way ABBA is presented, the validity condition ensures that an $n-t$ majority on some value suffices to guarantee this value is the outcome of ABBA. In case of a verifiable input, we can bias this threshold. Instead of taking the simple majority of the preprocessing votes as an input for the first pre-vote, the following rule is applied:
If at least one (verifiable) pre-processing vote for `1` was received, use `1` as the first pre-vote value. Otherwise, vote `0`.

**Lemma 4.5.** *If at least $t+1$ parties enter ABBA with initial vote `1`, then the outcome of ABBA is `1`.*

*Proof.* If at least $t+1$ parties enter ABBA voting `1`, then all parties see at least one initial `1` vote, and thus vote `1` in the first pre-vote. By the strong validity of ABBA, this is sufficient to ensure that 1 is the decision value. □

### 4.4.4 Parallel Byzantine Agreement

If an application requires to run several (say, $k$) independent instances of Byzantine agreement, it is possible to collapse several instances into one parallel protocol. This not only allows piggybacking, it also saves real work. For example, one common coin can be used for all $k$ protocols, and one signature can sign $k$ messages. It also decreases the overall round complexity, as the following lemma shows.

**Lemma 4.6.** *Let $X_1, X_2, ..., X_k$ be random variables such that for every $1 \leq i \leq k$, $Pr[X_i > j] = 2^{-j}$ If $Y = \max\{X_i\}$, then the expected value $Exp[Y] = O(\log k)$.*

A similar claim for independent variables has been stated by Ben-Or and El-Yaniv [BOEY91] and implicitely by Ben-Or, Kelmer and Rabin [BOKR94]. However, as our model allows an adversary to create dependencies between the instances, we need a stronger statement here.

*Proof.* Consider the following game: The adversary is given a set of $k$ pebbles. The game then goes in rounds. In each round $l$, the adversary can divide the pebbles two (possibly empty) heaps; then, a randomly chosen heap is removed from the game, and the next round is started, unti the adversary runs out of pebbles.

Each pebble corresponds to one random variable (or one instance of the Byzantine agreement protocol). For each pebble, the probability to be taken in round $j$ is $2^{-j}$. Thus, the round in which a pebble is taken away corresponds to the value of its assigned random variable. We now have to show that the expected time until the game ends is logarithmic in $k$.

It is easy to show that an adversary strategy exists such that a game with $k$ pebbels requires $O(\log(k))$ rounds; the adversary simply divides the instances into two groups of equal size, then every round half of the instances terminate, and after $\log(k)$ rounds only one instance is left.

We will now show that there is no strategy with which the adversary can play longer than logarithmic in $k$.

Let $f(k)$ be the expected number of rounds the adversary can play the game with $k$ pebbles using the optimal strategy.

We increase the number of pebbles to $k+1$, marking the additional pebble red. The adversary can now play the optimal strategy for $k+1$ pebbles, with the restriction that if one heap is larger than the other, the red pebble must be in the larger heap.

There are two cases:

– At some point, the red pebble is in a heap containing at least two pebbles that is removed from the game.

– The red pebble is one of the last two pebbles left.

In the former case, the red pebble did not buy the adversary anything. The same strategy would have resulted in the same number of rounds with $k$ pebbles, i.e., the adversary could not profit from the additional red pebble.

In the latter case, the same strategy could have been played without the red pebble until the last round. Thus, in this case the adversary gains not more than expected $f(1)$ rounds compared to the optimal strategy with $k$ pebbles.

If the adversary plays the optimal strategy, and the pebble is always in the remaining (larger) heap, the game has at least $f(k)$ rounds. In every round, the red pebble is lost with probability $\frac{1}{2}$. Therefore, the probability that case (2) holds is at most $2^{-f(k)}$, and the expected gain of the extra red pebble is at most $2^{-f(k)} \cdot f(1)$ rounds.

This implies the following equation:

$$f(k+1) \leq f(k) + 2^{-f(k)}$$

Taking both sides to the power of 2 and applying some transformations, we get

$$2^{f(k+1)} \leq 2^{f(k)+2^{-f(k)}}$$

$$2^{f(k+1)} \leq 2^{f(k)} \cdot 2^{2^{-f(k)}}$$

$$2^{f(k+1)} \leq 2^{f(k)} \cdot \sqrt[2^{f(k)}]{2}$$

We know that

$$lim_{x\to\infty}(1+x)^x = e,$$

which is strictly monotonic increasing and $(1+x)^x = 2$ for $x = 1$, thus $2 \leq (1+x)^x \leq e$ for $x \geq 1$. Using this inequality with $x = 2^{f(k)}$,

$$\sqrt[2^{f(k)}]{2} \leq (1+\frac{1}{2^{f(k)}}) \leq \sqrt[2^{f(k)}]{e} \text{ for } f(k) \geq 0.$$

Thus,

$$2^{f(k+1)} \leq 2^{f(k)} \cdot (1+\frac{1}{2^{f(k)}}) = 2^{f(k)} + 1.$$

This implies that $f(k)$ increases not faster than $\log(k)$ in $k$. With $f(1) = 1$ and $\log(1) = 0$ this finally leads us to

$$f(k) \leq \log(k) + 1.$$

□

For completeness, we will now show the modification to the proof for a generalized case, which corresponds more directly to [BOEY91].

**Lemma 4.7.** *Let $X_1, X_2, ..., X_k$ be random variables such that for every $1 \leq i \leq k$, $Pr[X_i > j] = \frac{q}{p}^j$ $(q, p \in \mathbb{N}, q < p)$. If $Y = \max\{X_i\}$, then the expected value $Exp[Y] = O(\log k)$.*

*Proof.* The proof is similar to the one above, with the difference that the adversary divides the pebbles into $p$ heaps, and a random subset of $q$ heaps are taken away each round – again, the probability for an individual pebble to survive $j$ rounds is $\frac{q}{p}^j$.

We repeat the game with the red pebble. We have to be a bit stricter on the two cases:

– At some point, there are at least two pebbles in the game, and the red pebble is in a heap that is removed from the game.

– The red pebble is the last pebble left.

In the first case, the adversary could have played the same strategy without the red pebble as well, as it is allowed to use empty heaps. In the second case, it won $f(1)$ rounds compared to the optimal strategy with $k$ pebbles. As the red pebble is removed with probability $\frac{q}{p}$ in any given round (provided it has not been removed earlier), the expected gain of the red pebble is $\frac{q}{p}^{f(k)} \cdot f(1)$ rounds. The same computation, replacing 2 with the value $\frac{p}{q}$, then implies

$$(\frac{p}{q})^{f(k+1)} \leq (\frac{p}{q})^{f(k)} + 1,$$

i.e., $f(k) \leq \log_{\frac{p}{q}}(k) + 1$. □

## 4.4.5 Saving Cryptographic Operations

As ABBA makes heavy use of public-key cryptography, it can easily become *compute-bound*, i.e., the bottleneck is not the network, but the computation power. In our test implementation, one agreement between four parties without signatures and with an optimistic coin (i.e., the first coins are generated by a normal pseudorandom number generator, and only coins generated in later rounds use distributed coin protocol) needed about 4 ms; generating a single signature on the same computers takes 9 ms.

Thus, an important point in the implementation is to optimize away cryptographic operations.

In every round, it is only possible to receive 2 different pre-votes (a hard and a soft one) and 2 different main-votes (one for a value and one for abstention). If a vote arrives that votes for a value which already has been verified, it is not necessary to check the validation information again; it is possible to legally vote for this value, and this is enough. Furthermore, if $P_i$ voted $\rho$ in some vote, it does not need to verify any incoming votes on $\rho$ anymore; it knows that $\rho$ is a legal vote.

Also, if somebody else's vote for $\rho$ has been received, $P_i$ does not need to generate a proof on $\rho$ itself; it is sufficient to include the received proof.

Thus, the cryptographic effort for one vote is massively reduced; it is sufficient to verify at most two threshold signatures per vote, and the generation might be omitted completely; thus the only cryptographic operations every party has to do in every round is one verification, the generation of signature shares and the common coin.

**Lemma 4.8.** *If a party $P_i$ receives one justified pre-vote message $m$ for some value $\rho$, then it is not necessary to verify the justifications of further pre-votes for $\rho$ in that round. Furthermore, the received justification received with $m$ can be used if $P_i$ itself sends a pre-vote for $\rho$ in that round.*

*The corresponding statement holds for main-votes.*

*Proof.* Suppose an honest party $P_i$ has received a justified pre-vote message $m$ for $\rho$, and a pre-vote message $m'$ for $\rho$ from another party.

There are two cases:

– The message $m'$ was sent by an honest party. Then it certainly does not matter if the justification is verified or not.

– The message $m'$ is sent by a dishonest party $P_j$. Now the adversary has seen $m$, and thus knows how a justified pre-vote for $\rho$ looks like. If $P_j$ would uses this justification for $m'$, it is capable to sent a justified pre-vote for $\rho$ that $P_i$ would accept. If corrupted party *can* send a justified pre-vote for $\rho$, there is no benefit for sending a pre-vote for $\rho$ with an invalid justification (other than using the space to include obscene messages). Thus, there is also no reason to verify if it actually did.

This argument can be reversed; if $P_i$ sends a pre-vote message $m''$ for $\rho$, the receivers can apply the same argumentation to convince themselves that the justification for $m$ is sufficient to justify $m''$. The same argumentation holds for all other votes messages. □

### 4.4.6 Streamlining ABBA

We now describe some minor optimizations, most of which lead to a more flexible, "pipelined" execution of the protocol steps or omit some steps altogether.

1. A party need not generate a share of the coin in round $r+1$ if it did not accept a main-vote of `abstain` in round $r$, as this ensures that no honest party can receive $n-t$ abstentions and thus use the coin.

2. A party need not wait for $n-t$ coin shares, unless it is going to cast a soft pre-vote, or unless it needs to later verify the justification of a soft pre-vote (it can always wait for them later if needed).

3. A party need not wait for $n-t$ pre-votes once it accepts two conflicting pre-votes, since then it is already in a position to cast a main-vote of `abstain`.

4. A party need not wait for $n-t$ main-votes if it has already accepted a main-vote for something other than `abstain`, since then it is already in a position to move to the next round; however, the decision condition should be checked before the end of the next round.

5. It is possible to collapse steps 4 and 1; however, some adjustments must be made to accommodate the threshold signature. If a party wants to make a hard pre-vote for $b$, he should generate signature shares on two messages that say "I pre-vote $b$ if the coin is 0" and "I pre-vote $b$ if the coin is 1." If a party wants to make a soft pre-vote, he should generate signature shares on two messages that say "I pre-vote 0 if the coin is 0" and "I pre-vote 1 if the coin is 1." This allows the parties to make soft pre-votes and reveal the coin concurrently, while also making it possible to combine both soft and hard pre-votes for the same value to construct the necessary main-vote justifications. This variation reduces the round and message complexity by a factor of 1/3, at the expense of somewhat higher computational and bit complexity; it also precludes variations (1) and (2) above.

6. In a real world system, an adversary is normally not as strong as assumed here; especially, it is very hard to obtain the level of network control we give the adversary on a real network like the Internet. Thus, it might be worthwhile to use a (predictable) common coin generated by a local, deterministic pseudorandom number generator for the first couple of rounds. If the protocol then reaches a high round number (for example, round ten), it switches to the distributed coin generation to guarantee termination. If the predictable coin is sufficient, this saves one message and some cryptographic operations per round, at the price that the decision is delayed a bit if the adversary is indeed as strong as our model assumes.

## 4.5 Losing and Ignoring messages

This section deals with the case that not only the parties, but also the interconnecting links might be corrupted. A corrupted link[2] might change, suppress or generate messages. By using message authentication, both changing and generating messages will be detected by the receiver, and the corresponding messages will simply be dropped. For this reason, we consider only links that lose messages in the following.

The mechanism used for this has other advantages; is increases the performance of ABBA and balances the load towards the fast participants. Furthermore, the modified protocol is optimal (modulo constants and the security parameter) in message– and communication complexity.

### 4.5.1 Shortcutting

In the basic protocol, the parties do not make full use of all information they get. We introduce the concept of "shortcutting" into the protocol to both optimize efficiency and deal with lost messages.

The main idea of shortcutting is that if some party reached round $r$, it is not necessary that anyone puts any work into rounds smaller than $r$. Thus, a party that receives a vote message from round $r$, while itself being in a previous round, can immediately jump to round $r$ as well.

The protocol ABBA is modified by adding an additional condition:

**Shortcutting.** If the last vote sent was in round $r$, and a (justified) vote-message for round $r' > r$ is received, broadcast this message as your own vote. Ignore all messages concerning rounds $< r'$.

For the pre-vote on the common coin, this is slightly more complicated; we need a coin that comes with a proof of correctness to apply the same strategy. Therefore, we do not use the common coin as introduced in Section 3.4, but use a deterministic threshold signature [Sho00] instead. A proof for a pre-vote for the common coin then also requires the coin-value and the proof that this coin value is correct. We will now show that ABBA with shortcutting still solves Byzantine Agreement.

**Theorem 4.9.** *Assuming a secure threshold signature scheme, a secure verifiable threshold coin-tossing scheme, and a secure message authentication code, protocol ABBA with shortcutting solves asynchronous Byzantine agreement for $n > 3t$.*

The shortcutting property changes the original protocol in two ways. Firstly, a party takes messages it receives from another party and sends them out as its own messages. Secondly, a party may not participate in all votes and may not issue all coin shares. The following two lemmas show that for both changes, if an adversary could break the changed protocol, it could be modified to an adversary that breaks the original ABBA protocol.

**Lemma 4.10.** *If an honest party broadcasts a received vote $v$ as its own vote due to shortcutting, then the adversary could make this party broadcast $v$ in the unmodified ABBA protocol as well.*

*Proof.* Suppose party $P_i$ receives (and accepts) a vote message $m$ in round $r$. Then $m$ contains a justification, consisting of:

– $n - t$ signatures on pre-votes from round $r - 1$ , if $m$ is a hard pre-vote

– $n - t$ signatures on abstaining main votes from round $r - 1$, if $m$ is a soft pre-vote

– $n - t$ signatures on pre-votes from round $r$, if $m$ is a main-vote for 0 or 1

[2] The term "corrupted link" is a bit sloppy; in our model, all links are corrupted by definition, as the adversary controls the entire network. Thus, the precise term would be that the adversary is allowed to drop a message between two honest participants. We nevertheless use the "corrupted link", just as we use the term "corrupted party" for a party that is impersonated by the adversary, as a simplification of language.

– $n - t$ signatures on pre-votes from round $r - 1$ and $n - t$ signatures on abstaining main votes from round $r - 1$, if $m$ is an abstaining main-vote.

At least $t + 1$ of these signatures have been issued by honest parties.

We will now argue that in all four cases, the adversary could have generated the input for $m$ in the normal ABBA protocol that would have caused $P_i$ to issue the same vote.

Suppose $m$ is a hard pre-vote for $\rho$. An honest party generates a hard pre-vote on accepting a main-vote for $\rho$ from round $r - 1$, which is not an abstention. As the justification of a hard pre-vote for $\rho$ in round $r$ is identical to the justification of a main-vote for $\rho$ in round $r - 1$, the adversaries ability to generate a justified hard pre-vote for round $r$ implies it can generate a justified main-vote on $\rho$ fro round $r - 1$. Thus, it is easy to arrange for a corrupted party to send $P_i$ such a main-vote and thus cause it to output a hard pre-vote for $\rho$.

Suppose $m$ is a soft pre-vote. An honest party generates a soft pre-vote on accepting $n - t$ abstaining main votes. If $m$ is justified, at least $t + 1$ honest parties have issued abstaining main vote for round $r - 1$, and all corrupted parties can easily justify one ( by using the same justification the honest parties did). Thus, if the adversary can give $P_i$ a justified soft pre-vote, it can also give $P_i$ $n - t$ abstaining main-votes from the previous round, causing it to generate a soft pre-vote as well. Note that at this point, we need a verifiable coin; it is not enough for $P_i$ to know it can send a soft pre-vote, it also has to be able to know the coin at this point to select the right value to vote for.

Suppose $m$ is a main vote for 0 or 1. An honest party generates a non-abstaining main-vote on accepting $n-t$ corresponding pre-votes from the same round. If the adversary can justify an non-abstaining main-vote, at least $t + 1$ honest parties must have issued the corresponding pre-votes, and thus — similar to the case of the hard pre-vote — it is straightforward to generate the input $P_i$ needs to output a non-abstaining main vote.

Finally, suppose $m$ is an abstaining main vote. The same argument as above holds: at least $t + 1$ honest parties must have issued the corresponding pre-votes from round $r - 1$, and $t + 1$ honest parties must have issued the corresponding abstaining main vote, thus it is easy for the adversary to generate the two conflicting pre-votes that cause $P_i$ to vote $m$.

Thus, the adversary does not gain anything at all; if $P_i$ sends a message in the modified protocol, the adversary can easily make it send the same message in the unmodified protocol. □

The second change to the protocol is that a party does not participate in all votings anymore.

**Lemma 4.11.** *Agreement, validity, efficiency and deadlock freeness are not violated because of messages that would be send in ABBA, but are not send due to shortcutting.*

*Proof.* Missing messages have no influence on agreement, validity or efficiency; if a missing message would cause the protocol to terminate with one of these conditions violated, the adversary could simulate this effect by delaying delivery of that vote until the violation occurred.

It is left to show that our modification does not violate deadlock freeness.

Suppose an adversary can deadlock ABBA with shortcutting, i.e., all messages are delivered and no honest party wants to send a message. Let $m$ be the latest vote-message generated by some honest party, i.e., either the main-vote with the highest round number or a pre-vote with a round number higher than any main-vote. As at least one honest party $P_i$ received this message. If an honest party has not yet participated in that vote will take $m$ as their own and re-broadcast it. Otherwise, it has sent out a message for vote $m$ was for. In either case, all honest parties participate in that vote, and thus the protocol moves on.

Finally, we have to show that omitted coin shares do not deadlock the protocol. This is relevant if a party wants to make a *soft pre-vote.* However, if any honest party $P_i$ skips round $r$ and thus does not issue the corresponding coin share, then it has received a vote message from round $r + 1$ or a later round and broadcasted it to all parties as its own vote. Any honest party that is stuck waiting for coin shares in round $r$ then will receive a vote message from round $r + 1$ or a later round, thus skipping round $r$, which makes the missing coin shares irrelevant. □

### 4.5.2 Link Failures

The fact that not all messages are needed can also be used to tolerate a certain extend of message loss.

The protocol still works as long as for every vote, there is at least one party that gets enough input messages (both vote messages and coin shares) to generate a vote message for the subsequent vote, and all honest parties that cannot generate such a message receive it from someone else.

The number of messages that actually can be lost depends highly on the protocol flow. In the best case, it suffices if $n-t$ of the $n^2$ messages in a vote arrive. In the worst case, $n-t$ dropped messages in one vote are sufficient to stop the protocol. This is an optimal number, as $n-t$ dropped messages per vote suffices to completely cut of one honest party.

### 4.5.3 Optimality of ABBA with shortcutting

The concept of shortcutting allows us to make a trivial modification to the protocol with which ABBA is optimal in both message– and communication complexity (modulo constants and the security parameter).

We first show a lower bound on the communication complexity.

**Lemma 4.12.** *No protocol can solve asynchronous Byzantine Agreement with an expected message complexity better than $O(n \cdot (t+1))$.*

*Proof.* Suppose that the expected message complexity of a protocol is not $O(n \cdot (t+1))$. Thus, during a run without corruptions, there is at least one party $P_i$ that receives less than $t+1$ messages, i.e., at most $t$ parties send a message to this party, independent on the message scheduling.

Now suppose a run with $t$ corruptions, where the corrupted parties are exactly the $t$ parties that send a message to $P_i$, while $P_i$ itself is honest.

In an asynchronous environment, a party can not distinguish between a slow and a corrupt party. Thus, any mechanism that causes any uncorrupt party to send a message to $P_i$ could be triggered by bad scheduling in a run with only honest parties, which contradicts the assumption.

Therefore, either $t$ corrupted parties can deadlock an honest party – it never receives any input – or the protocol does not have a better expected message complexity than $O(n \cdot (t+1))$.

□

To reach this complexity in ABBA, we distinguish $t+1$ parties as special parties, the *relays.* The honest parties then only talk to the relays. No messages are sent between two parties honest parties that are not relays. One of the relays is honest and always generates the next vote message and distributes it to all other honest parties; this is sufficient to still guarantee the functionality of ABBA (if no additional messages get lost), while the communication is reduced to $O(n \cdot (t+1))$ messages of constant size (modulo the security parameter).

This construction is theoretically optimal, but creates new bottlenecks and thus will probably decrease the performance in a real implementation. However, the normal implementation of ABBA with shortcutting adapts nicely to real-world performance issues and balances the load quite well; parties that are faster than others do the main work, while slower parties may skip some operations and thus have to perform less work.

### 4.5.4 Using negative Acknowledgments

Even though we can tolerate loss of many messages, it might happen in a real world scenario that too many messages get lost and the protocol deadlocks. We cannot deal with this problem in a timeless model, but by using timeouts we can implement a resending algorithm.

The standard way of message resending algorithms, as for example used in TCP/IP, is by use of positive acknowledgments. This basically means that every received message is acknowledged to the sender, and the sender keeps on resending until it gets the acknowledgment, or reports an error if it waited too long.

For our application, we do not need this kind of acknowledgment, as some message loss can be tolerated without problems; positive acknowledgment will only create useless overhead if messages that are no longer needed are resent.

The resending algorithm we propose is based on negative acknowledgments. Every party $P_i$ keeps a timer for every started transaction. This timer is reset every time progress is made, i.e., the party broadcasted a vote. If no progress is made for a too long time, the party broadcasts an SOS-message (and keeps on doing so until something happens). On receiving an SOS-message by $P_i$, a party $P_j$ resends its last vote-message to $P_i$. Thus, $P_i$ will be brought up to date, and it might skip some rounds if it is really behind.

This approach has several advantages. Firstly, no acknowledgments are needed, thus in the normal case, the resending mechanism creates no communication overhead. Second, the mechanism does not try to resent messages whose loss can be tolerated. Finally, a party that has been cut off for a while only needs the latest messages to keep up with the other parties, i.e., most messages are not resent.

## 4.6 General Hybrid Adversary Structures

In this section, we go beyond the traditional $t$– out-of-$n$ failures model. Our goal is to represent structures that exist in the real world — corruptions do not happen independently, and some failures are more likely to occur than others. We extend the Byzantine agreement protocol in two ways:

**Hybrid Failures.** Instead of only accepting Byzantine failures, we want to tolerate a mixture of Byzantine and crash failures. This allows the protocol to tolerate more overall failures. More precisely, if $c$ is the number of crash failures, and $b$ is the number of Byzantine failures, then the necessary and sufficient condition is $2c + 3b < n$.

**Adversary Structures.** Instead of tolerating a fixed number of failures, define sets of parties that may fail simultaneously. This does not increase the number of tolerable failures, but allows to model real world structures. For example, we can assign attributes to participants (e.g. the operating system and the geographical location), and then tolerate failure of all participants with a certain attribute value.

### 4.6.1 General Definitions

Let $\mathcal{P}$ be the set of all participants. An *adversary structure* $\mathcal{Z}$ is a monotone set of classes $(C, B)$ of subsets of $\mathcal{P}$ (i.e., $C, B \subset \mathcal{P}$) [FHM99]. The adversary structure $\mathcal{Z}$ satisfies the predicate $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$, if $\forall (B_1, C_1), (B_2, C_2), (B_3, C_3) \in \mathcal{Z} : \{B_1 \cup B_2 \cup B_3 \cup C_1 \cup C_2\} \neq \mathcal{P}$.

We will first show that an adversary structure $\mathcal{Z}$ that satisfies $Q^{(3,2)}(\mathcal{P}, \mathcal{Z})$ is necessary sufficient to solve Byzantine agreement.

**Theorem 4.13.** *To solve Byzantine agreement on an adversary structure $\mathcal{Z}$, a necessary and sufficient condition is that $\mathcal{Z}$ satisfies $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$.*

*Proof.* Sufficiency follows from the existence of a protocol as presented in Section 4.7. It remains to show necessity.

Given an adversary structure $\mathcal{Z}$, and some pairs $(B_1, C_1), (B_2, C_2), (B_3, C_3) \in \mathcal{Z}$, such that $\mathcal{P} = \{B_1 \cup B_2 \cup B_3 \cup C_1 \cup C_2\}$.

Fix two honest parties $P_1 \notin B_1 \cup C_1$ and $P_2 \notin B_2 \cup C_2$. Let $B_3$ be the set of Byzantine parties.

Now $P_1$ has to be able to terminate the protocol without ever having heard from any party in $B_1 \cup C_1$. Similarly, $P_2$ must be able to decide without ever having heard of any party in the set $B_2 \cup C_2$. Thus, the input-values $P_1$ sees are the values form the parties in the set

$$B_2 \cup B_3 \cup C_2,$$

while $P_2$ sees the input values of the parties in

$$B_1 \cup B_3 \cup C_1.$$

As all parties in $B_3$ are Byzantine and might send different input values to different parties, these sets are disjunct, and an adversary can easily cause disagreement. $\square$

This theorem encompasses several special cases for the non-hybrid model (where an adversary strucutes is a set of subsets of $\mathcal{P}$ rather than a set of classes), and the the threshold model:

- If only crash failures are possible, then it is necessary and sufficient that $\mathcal{Z}$ satisfies $Q^2(\mathcal{P}, \mathcal{Z})$, i.e., that no two sets in $\mathcal{Z}$ cover $\mathcal{P}$ (see Definition 2.5).
- If only Byzantine failures are possible, then it is necessary and sufficient that $\mathcal{Z}$ satisfies $Q^3(\mathcal{P}, \mathcal{Z})$, i.e., that no three sets in $\mathcal{Z}$ cover $\mathcal{P}$.
- If $\mathcal{Z}$ is homogeneous, i.e., all $\mathcal{Z}$ contains exactly all classes $(C, B)$ with $|C| = c, |B| = b$, then $2c+3b < n$ is a necessary and sufficient condition. This corresponds to the threshold model without adversary structures, where $b$ is the maximum number of Byzantine failures and $c$ is the maximum number of crash failures.

For arguing about protocols in the new setting, it is helpful to completely abstract away all thresholds. Instead, we define sets with certain properties that we need for the proofs. For example, if a line in a protocol states

**wait** for $n - t$ votes,

then the reason that we wait for exactly $n - t$ votes is that this number of votes has a certain property; for example, it is the highest number of votes we can wait for without risking a deadlock.

If we want to argue about complicated adversary structures, it is helpful to identify the properties of the thresholds that we need to prove the protocol correct, and thus argue in a more abstract way. Once we nailed down the properties of the individual thresholds we need to prove the correctness of the protocol, it is relatively easy to move a protocol from the simple threshold model to a more refined failure model. We start by defining sets that correspond to the different thresholds in the conventional model.

**Definition 4.2.** Let $\mathcal{Z}$ define an adversary structure that satisfies $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$. A set $\mathcal{A} \subset \mathcal{P}$ is called a

- *full set*, if there exists a class $(C, B) \in \mathcal{Z}$, such that $\mathcal{A} \supseteq \mathcal{P} \setminus (B \cup C)$.
- *big set*, if there exists two classes $(C_1, B_1)$ and $(C_2, B_2) \in \mathcal{Z}$, such that $\mathcal{A} \supseteq \mathcal{P} \setminus B_1 \cup C_1 \cup B_2$.
- *small set*, if there is no class $(C, B) \in \mathcal{Z}$ such that $B \supseteq \mathcal{A}$.

More intuitively, a $\mathcal{A}$ is a full set if all parties in $\mathcal{P} \setminus \mathcal{A}$ might be faulty; after receiving messages from a full set $\mathcal{A}$ of parties, a party can not expect to receive more messages. This corresponds to the threshold $n - t$ in our previous notation. Given a full set of parties, some of which are Byzantine, a *big set* is the set that remains when the Byzantine parties are taken away. This corresponds to the threshold $n - 2t$ in our conventional notation. Finally, a small set of parties is a set that contains at least one non-Byzantine party. This corresponds to the threshold of $t + 1$.

A full set of signatures on $m$ denotes signatures on $m$ issued by a full set of parties. Similarly, a full set of coin shares denotes coin shares issued by a full set of parties.

We will now prove some elementary properties of above sets.

**Lemma 4.14.** *The following statements hold:*

**P1.** *Any big set contains at least one non-Byzantine party.*

**P2.** *Two full sets always have at least one non-Byzantine party in common.*

**P3.** *Each full set of parties contains at least a big set of non-Byzantine parties.*

**P4.** *If a full set of parties does an action $X$, and no non-Byzantine party does both actions $X$ and $Y$, then no full set of parties does $Y$.*

**P5.** *Every big set is a small set.*

**P6.** *Each pair of a full set and a big set of parties intersects.*

*Proof.*

Suppose statement 1 does not hold. Then there is a big set $\mathcal{A}$ and a class $(C, B)$ such that $\mathcal{A} \subseteq \mathcal{B}$. By definition, there are two classes $(C_1, B_1)$ and $(C_2, B_2)$ such that $\mathcal{A} = \mathcal{P} \setminus (C_1 \cup B_1 \cup B_2)$. Thus, $\mathcal{P} \setminus (C_1 \cup B_1 \cup B_2) \subseteq B$, and therefore $\mathcal{P} \subseteq (C_1 \cup B_1 \cup B_2 \cup B)$. This contradicts the definition of a full set.

For statement 2, take two full sets $\mathcal{A}_1 = \mathcal{P} \setminus (B_1 \cup C_1)$ and $\mathcal{A}_2 = \mathcal{P} \setminus (B_2 \cup C_2)$. Thus,

$$\mathcal{A}_1 \cap \mathcal{A}_2 = \mathcal{P} \setminus (B_1 \cup C_1 \cup B_2 \cup C_2).$$

By the definition of $Q^{(3,2)}(\mathcal{P}, \mathcal{Z})$, the remaining set is larger than any $B_3$ that occurs in a class $(B_3, C_3) \in \mathcal{Z}$. Statement 3 follows directly from the definition of a full- and a big set. Statement 4 follows directly from the Statement 2, while statement 5 follows directly from statement 1. Finally, let $A \supseteq \mathcal{P} \setminus (B_1 \cup C_1)$ be a full set, where $(B_1, C_1) \in \mathcal{Z}$. If $A' = \mathcal{P} \setminus A \subseteq (B_1 \cup C_1)$ is a big set, then there exist $(B_2, C_2), (B_3, C_3) \in \mathcal{Z}$ such that $A' \supseteq \mathcal{P} \setminus (B_2 \cup B_3 \cup C_2)$, and thus $(B_1 \cup C_1) \supseteq \mathcal{P} \setminus (B_2 \cup B_3 \cup C_2)$, which implies $\mathcal{P} \subseteq (B_1 \cup B_2 \cup B_3 \cup C_1 \cup C_2)$, which contradicts $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$. In other words, if a full set of parties is removed from the set of all parties, the remaining set is not big, which proves the claim that each pair of a full and a big set intersects.

□

Note that in the conventional threshold model, if we assume the maximum number of corruptions (i.e., $n = 3t + 1$) we get $n - 2t = t + 1$, i.e., there is no difference between a *big set* and a *small set*.

### 4.6.2 Generalizing the Cryptographic Primitives

We also have to adapt the cryptographic primitives to the new model. While hybrid failures do not cause any difference outside of changing the thresholds, the adversary structures have quite some impact onto the common coin and the threshold signature scheme.

#### Threshold Coin-Tossing Scheme

We first define the generalized coin-tossing scheme. As in the threshold coin tossing scheme, there is a set $\mathcal{P}$ of $n$ parties, some of which may be corrupted according to an adversary structure $\mathcal{Z}$ that satisfies $Q^{(3,2)}(\mathcal{P}, \mathcal{Z})$. The parties hold shares of an unpredictable function $F$ mapping the name $C$ (which is an arbitrary bit string) of a coin to its value $F(C) \in \{0, 1\}$. The parties may generate shares of a particular coin.

Let $\mathcal{K}$ be the set of sets of parties that are sufficient reconstruct the coin. The conditions on $\mathcal{K}$ are that every full set must be in $\mathcal{K}$, and every set in $\mathcal{K}$ must be a small set. This corresponds to the statement $t < k \leq n - t$ in the threshold model, where $k$ is the number of shares needed to reconstruct the coin. Let $\mathcal{K}'$ be the set of sets that satisfy the following condition: $K \subseteq \mathcal{P} \in \mathcal{K}' \Rightarrow \exists (C, B) \in \mathcal{Z}$ such that $K \cup B \in \mathcal{K}$. We will sometimes use shares as an equivalent to the parties holding the shares, i.e., use the term "a set $K \in \mathcal{K}$ of shares".

Note that crash failures have no impact on the coin tossing scheme; the only effect that a crashing party has is that less honest parties generate shares. This is not relevant here; the coin tossing scheme is only responsible to protect the secrets and to combine a sufficient number of shares to a valid coin, not to ensure that a sufficient number of shares are generated.

Like in the threshold scheme, the dealer initializes the parties by generating secret key shares $SK_1, \ldots, SK_n$, and verification keys $VK, VK_1, \ldots, VK_n$.

After the dealing phase, the adversary submits *reveal requests* to the honest parties for coins of his choice. Upon such a request, party $P_i$ outputs a *coin share* for the given coin, which it computes using $SK_i$.

The definition of the *share verification* algorithm and the *share combining* is slightly modified:

- The *share verification* algorithm takes as input the name of a coin, a share of this coin from a party $P_i$, along with $VK$ and $VK_i$, and outputs **accept** or **reject**. If it outputs **accept**, we say that the share is valid.

- The *share combining* algorithm takes as input the name $C$ of a coin and a set $K \in \mathcal{K}$ of valid shares of the coin, along with (perhaps) the verification keys, and (hopefully) outputs $F(C)$.

Similar modifications are made to the security requirements:

*Robustness.* There is a function $F(C)$ that maps names to $\{0, 1\}$. It is computationally infeasible for an adversary to produce a name $C$ and a set $K \in \mathcal{K}$ of valid shares of $C$ such that the output of the share combining algorithm is not $F(C)$.

*Unpredictability.* An adversary's *advantage* in the following game is negligible. The adversary interacts with the honest parties as above, and at the end of this interaction, he outputs a name $C$ that was submitted as a reveal request to a set of non-Byzantine parties that is not in $\mathcal{K}'$, and a bit $b \in \{0, 1\}$. The adversary's advantage in this game is defined to be the distance from $1/2$ of the probability that $F(C) = b$.

To implement a common coin scheme, it is necessary to make more precise statements on the set $\mathcal{K}$. This is equivalent to the threshold case, where we finally have to say what the parameter $k$ is.

For our Byzantine agreement protocol, we need a common coin where all sets in $\mathcal{K}$ are full sets, i.e., a full set of shares is both necessary and sufficient to determine the coin value. We will call a coin tossing scheme with this property a *threshold coin tossing scheme over a full set.*

The most straightforward way to adapt our threshold common coin to adversary structures is to use several instances of the common coin protocol (using the same hash functions and the same secret $x$, but otherwise independent), one for each set of parties that is allowed to assemble the coin. For each set of size $n'$, the corresponding common coin then is a $(n', n', 0)$ threshold coin tossing scheme. The secret share of a party holds for the coin tossing scheme is the set of one share for each of these instances.

The algorithms work exactly like in the normal threshold coin tossing scheme, just several of them in parallel. Robustness and unpredictability follow from the robustness and unpredictability of the individual common coin protocols and the fact that they are – beside using the same secret – independent.

This scheme has the disadvantage that the number of sets that might be allowed to reconstruct the secret might be quite large – under most circumstances, its size (and thus the size of the coin shares) will grow exponentially with the number of parties. In our model, this is not allowed — unless we impose further restrictions on the number of participants, a set whose size is exponential in the number of participants violates the poly-time assumptions (besides imposing practical problems that might make those schemes unusable).

### Threshold Signatures

Like the coin tossing scheme, a generalized signature scheme is similar to the threshold signature scheme. During the initialization, a dealer generates a public key $PK$ along with secret key shares $SK_1, \ldots, SK_n$, a global share verification key $VK$, and local share verification keys $VK_1, \ldots, VK_n$. The initial state information for party $P_i$ consists of the secret key $SK_i$ along with the public key and *all* the verification keys. Let $\mathcal{K}$ be the set of sets of parties that are sufficient generate a signature. The conditions on $\mathcal{K}$ are that every full set must be in $\mathcal{K}$, and every set in $\mathcal{K}$ must be a small set. This corresponds to the statement $t < k \leq n - t$ in the threshold model, where $k$ is the number of shares needed to reconstruct the coin. Let $\mathcal{K}'$ be the set of sets that satisfy the following condition: $K \subseteq \mathcal{P} \in \mathcal{K}' \Rightarrow \exists (C, B) \in \mathcal{Z}$ such that $K \cup B \in \mathcal{K}$. We will sometimes use shares as an equivalent to the parties holding the shares, i.e., use the term "a set $K \in \mathcal{K}$ of shares". After the dealing phase, the adversary submits signing requests to the honest parties for messages of his choice. Upon such a request, party $P_i$ computes a *signature share* for the given message using $SK_i$. The three algorithms provided by the generalized signature scheme now look as follows:

- The *share verification* algorithm takes as input a message, a signature share on that message from a party $P_i$, along with $PK$, $VK$, and $VK_i$, and determines if the signature share is valid.

- The *share combining* algorithm takes as input a message and a set $K \in \mathcal{K}$ of valid signature shares on the message, along with the public key and (perhaps) the verification keys, and (hopefully) outputs a valid signature on the message.
- The *signature verification* algorithm takes as input a message and a signature, along with the public key. It outputs either **accept** or **reject**; if it outputs **accept**, we say that the signature is valid.

The two basic security requirements are modified are now

*Robustness.* It is computationally infeasible for an adversary to produce a set $K \in \mathcal{K}$ of valid signature shares such that the output of the share combining algorithm is not a valid signature.

*Non-forgeability.* It is computationally infeasible for the adversary to output a valid signature on a message that was submitted as a signing request to a set on non-Byzantine parties that is not in $\mathcal{K}'$.

To implement the generalized signatures, we can use a collection of standard signatures, which easily fits into the generalized scheme. Like the common coin, it might be possible to generate a generalized threshold signature scheme from [Sho00], but this lies beyond the scope of this thesis (and its author).

### 4.6.3 Byzantine Agreement and General Hybrid Adversary Structures

To incorporate hybrid adversary structures into our protocol, we first have to slightly alter the definition of Byzantine agreement. The reason is that we now have to deal with two different kinds of corruptions. First of all, we have to give the adversary the possibility to crash parties as described in Section 2.5.

While Byzantine parties are still free to do anything they want, the parties that may crash still have to play by the rules, i.e., they may not deadlock and if they do not crash prior to deciding, and a decision they make has to satisfy validity and agreement. Therefore, we argue about non-Byzantine party instead of honest parties; this term covers both correct and crashing parties.

The basic properties of Byzantine agreement now look as follows:

**Agreement.** Any two non-Byzantine parties that decide a value for a particular *ID* must decide the same value. More precisely, it is computationally *infeasible* for an adversary to make two non-Byzantine parties decide on different values.

**Validity.** If all non-Byzantine parties have been activated on a given *ID* with the same initial value, then all non-Byzantine parties that decide must decide this value.

**Deadlock freeness.** It is infeasible for the adversary to create a situation where for some *ID* there are some non-Byzantine parties who are neither decided nor crashed, yet all non-Byzantine parties have been activated on *ID*, and all messages relating to this *ID* generated by non-Byzantine parties have been delivered.

**Efficiency.** For every *ID*, the communication complexity for *ID* is probabilistically polynomially bounded.

It is relatively straightforward to modify ABBA and its analysis for this model; all occurrences of $n - t$ are replaced by a *full set*, the occurrences of $n - 2t$ changes into a *big set* and $t + 1$ becomes a *small set.* We do not do the full analysis of ABBA in on hybrid adversary structures here; instead, a variation of ABBA that includes some other optimization as well will be presented and fully analyzed Section 4.7.

## 4.7 The Combined Protocol BASIL

We now present our protocol BASIL[3] , which combines most optimizations mentioned in the previous sections into one protocol, and proof the correctness of the protocol. BASIL is formulated in terms of hybrid adversary structures as defined in Section 2.5.2 and used in Section 4.6.3. Thus, it requires a threshold signature scheme that works on these adversary structures as well; although there is good reason to believe that Shoup's threshold signature scheme [Sho00] can be extended into this model, there is no formal proof so far.

The reason why BASIL uses adversary structures nevertheless is that this model is the most general. The standard assumption that $t$ out of $n$ parties are corrupted can be modeled as an adversary structure,

[3]After Basil the Macedonian, Emperor of the Byzantine Empire from 867-886.

thus every statement made in for BASIL on adversary structures holds for the numeric threshold model as well. As the use of adversary structures has quite some impact on the analysis, it seemed necessary though to analyze one full of a Byzantine agreement protocol in this model.

We will therefore assume that generalized threshold signature schemes on an adversary structure $\mathcal{Z}$ in $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$ exists. We need two schemes that allow a set $\mathcal{X}$ of signature shares to be combined to a signature if and only if $\mathcal{X}$ is at least a *full set* or a *small set*, respectively (the latter one is used for the pre-processing signature). We call these signature schemes a *threshold signature scheme over a full set* and a *threshold signature scheme over a small set*. In the numeric threshold model, this corresponds to an $(n, n-t, t)$ and an $(n, t+1, t)$ threshold signature scheme [Sho00]; a verifiable common coin over a full set (i.e. a full set of shares is needed to reveal the coin) or a $(n, n-t, t)$ verifiable common coin can be based on the same scheme.

Recall that each vote message has to contain a *justification*, which consists of threshold signatures on collected votes from previous votings. These justifications show that the it was possible to generate this vote while following the protocol, and thus outrule most malicious behavior. BASIL allows parties to “steal” justifications. If $P_i$ wants to justify a main vote for 0, and it has already detected a justified main vote on 0 from $P_j$ in the same round, it does not need to generate its own justification; it is sufficient to take the one received from $P_j$, provided it is valid. The verifications are generated by protocol GenJust (Figure 4.3) and verified by protocol VerJust (Figure 4.4). In BASIL, all outgoing vote messages are justified, and all incoming vote messages are verified for a valid justification. For the sake of readability, we will not explicitly mention this in the protocol description.

Recall that in our notation, when a party *receives* a message it deletes it from the incoming buffer; this message can then not trigger any other actions. When a party *detects* a message, the message stays in the incoming buffer and can thus be received or detected later again. BASIL uses some messages to trigger several actions; for example, a main vote can trigger a pre vote, the generation of a coin share, or the decision. As we do not know in which order this happens, all messages are *detected* rather than received. To be fully correct, we would need to insert conditions on when a message can be deleted from the incoming buffer to prevent it from growing forever. For the sake of readability, and because this is an artifact created by our model with little impact on a real implementation, we ignore this matter here.

```
**Subprotocol Generate justification**
  input: message $m$, round number $r$.
  **if** $m$ is a pre (main) vote for $\rho \in \{0, 1, \texttt{abstain}\}$, and a valid pre (main) vote $pv$ ($mv$) on $\rho$ by $P_j$ was already
      detected or generated, $J \leftarrow$ the justification in $pv$ ($mv$).
  **else**
        **if** $r = 1$ and $m$ is a pre vote for $\rho$,
              $J \leftarrow$ threshold signature on a small set of pre-process messages for $\rho$.
        **if** $r > 1$ and $m$ is a hard pre vote on $\rho$,
              $J \leftarrow$ a justification for a main vote on each $\rho$ and abstention from round $r-1$
        **if** $r > 1$ and $m$ is a soft pre vote (i.e., a pre vote on the coin),
              $J \leftarrow$ a threshold signature a full set of main votes for abstention in round $r-1$, and the justification
                  for the used coin $¢_r$.
        **if** $m$ is a main vote on $\rho$,
              $J \leftarrow$ a threshold signature on a full set of pre votes on $\rho$ from round $r$
        **if** $m$ is a main vote on abstain,
              $J \leftarrow$ one justification on a pre vote for 1 and one justification on a pre vote for 0 in round $r$.
        **if** $m$ is a decision vote on $\rho$,
              $J \leftarrow$ a threshold signature on a full set main votes for $\rho$ from the same round.
```

Figure 4.3: Subprotocol GenJust (Generate Justification) of BASIL

The protocol and the proof will adapt the hybrid adversary structures. We will use the following sets, as defined in Definition 4.2:

Let $\mathcal{Z}$ define an adversary structure that satisfies $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$. A set $\mathcal{A} \subset \mathcal{P}$ is called a *full set*, if there

```
Subprotocol Verify justification
  input: message $m$, round number $r$, Justification $J$.
  if $m$ is a pre (main,decision) vote for $\rho \in \{0, 1, \texttt{abstain}\}$, and a valid pre (main,decision ) vote $pv$ $(mv, dv)$ on $\rho$
      was already verified or generated, return true
  else
      if $r = 1$ and $m$ is a pre vote for $\rho$,
          and $J$ is signature on a small set of pre-process messages for $\rho$,
      or if $r > 1$ and $m$ is a hard pre vote on $\rho$,
          and $J$ a valid justification for a main vote on each $\rho$ and abstention from round $r - 1$
      or if $r > 1$ and $m$ is a soft pre vote,
          and $J$ is a threshold signature on a full set of main votes for abstention in round $r - 1$,
      or if $m$ is a main vote on $\rho$,
          and $J$ is a threshold signature on a full set of pre votes on $\rho$ from round $r$,
      or if $m$ is a main vote on abstain,
          and $J$ is one justification on a pre vote for 1 and one justification on a pre vote for 0 in round $r$,
      or if $m$ is a decision vote on $\rho$,
          and $J$ is a threshold signature on a full set of main votes for $\rho$ from the same round,
      then return true,
      else return false.
```

Figure 4.4: Subprotocol VerJust (Verify Justification) of BASIL

exists a pair $(b, c) \in \mathcal{Z}$, such that $\mathcal{A} \supseteq \mathcal{P} \setminus \{b \cup c\}$. The set $\mathcal{A}$ is called a *big set*, if there exists two classes $(C_1, B_1)$ and $(C_2, B_2) \in \mathcal{Z}$, such that $\mathcal{A} \supseteq \mathcal{P} \setminus B_1 \cup C_1 \cup B_2$. The set $\mathcal{A}$ is called a *small set*, if there is no set $(c, b) \in \mathcal{Z}$ such that $b \supseteq \mathcal{A}$.

As BASIL has no sequential round structure, we have to be a bit more careful in correctly indexing the local variables. The variables are used by BASIL and their initial values are as follows:
$b_{r,\text{pre}} \leftarrow \perp$: the pre vote value in round $r$.
$b_{r,\text{main}} \leftarrow \perp$: the main vote value in round $r$.
$r_{max} \leftarrow \infty$. The round in which the last non-Byzantine party decides.
$r_{min} \leftarrow 0$. The last round finished by that party.
$¢_r \leftarrow \perp$: the coin value in round $r$.
The value of the coin for round $r$ is called $¢_r$. Initially, we set

$$¢_r \leftarrow \begin{cases} \text{pseudorandom} & r \leq 10 \\ \perp & r > 10 \end{cases}$$

The pre-initialization of the first ten coins with a (deterministic) pseudorandom value increases performance if the adversary does not have full control over the network – for the first ten rounds, no computation or communication is needed to determine the coin. The price for this is that if an adversary does have full control over the network, it can delay termination by ten rounds.

The complete protocol is shown in Figures 4.3, 4.4 and 4.5.

### 4.7.1 Analysis

**Theorem 4.15.** *Assuming secure threshold signature schemes over a small and over a full set, and a secure threshold coin-tossing scheme over a full set, protocol BASIL solves asynchronous Byzantine agreement for an adversary structure $\mathcal{Z}$ in $\mathcal{Q}^{(3,2)}(\mathcal{P}, \mathcal{Z})$.*

We have to show *validity*, *agreement*, *deadlock freeness* and *efficiency*, as defined in Section 4.6.3.

Recall that by Lemma 4.14 , the following properties hold:

**P1.** Any big set contains at least one non-Byzantine party.

```
**Protocol BASIL for party $P_i$ with initial value $V_i$.**
case
  /* Step 0: Pre-Processing */

  upon receiving a message $(ID, \texttt{in}, \texttt{propose}, V_i)$,

      Generate a signature share $sh_i^{\texttt{step0}}$ of the preprocessing threshold signature on the string
                                  $(ID, \texttt{pre-process}, V_i)$
      send to all parties a message of the form
                                  $(ID, \texttt{pre-process}, V_i, sh_i^{\texttt{step0}})$.
  upon detecting a full set of pre-processing messages, let $b_{1,\mathrm{pre}}$ be the simple majority of the detected pre-processing
    votes.
  /* Step 1: Pre-Vote. */

  upon detecting a pre vote $pv_{r,j}$ for $\rho$ such that $r_{min} < r \leq r_{\max}$, if $b_{r,\mathrm{pre}} = \perp$
    $b_{r,\mathrm{pre}} \leftarrow \rho$
  upon detecting a main vote $mv_{r-1,j}$ for $\sigma_{r,\mathrm{pre}} \in \{0,1\}$ and a main vote $mv_{r-1,j'}$ for abstention such that $r_{min} <$
    $r \leq r_{\max}$, if $b_{r,\mathrm{pre}} = \perp$
    $b_{r,\mathrm{pre}} \leftarrow \sigma_{r,\mathrm{pre}}$
  upon detecting a full set of main votes $mv_{r-1,j_1}, \ldots, mv_{r-1,j_l}$ for abstention such that $r_{min} < r \leq r_{\max}$, if
    $b_{r,\mathrm{pre}} = \perp$, and $¢_{r-1} \neq \perp$:
    $b_{r,\mathrm{pre}} \leftarrow ¢_{r-1}$
  upon $b_{r,\mathrm{pre}} \neq done$ and $b_{r,\mathrm{pre}} \neq \perp$
      Produce a signature-share $sh_i^{\texttt{pre}}$ on the string
                                  $(\texttt{pre vote}, ID, r, b_{r,\mathrm{pre}})$.
      Send to all parties a message of the form
                                  $(\texttt{pre vote}, r, b_{r,\mathrm{pre}}, justification, sh_i^{\texttt{pre}})$.
      $b_{r,\mathrm{pre}} \leftarrow done; r_{min} \leftarrow r - 1$;
  /* Step 2: Main-Vote. */

  upon detecting a main vote $mv_{r,j}$ for $\sigma_{r,\mathrm{main}}$, such that $r_{min} < r \leq r_{\max}$, if $b_{r,\mathrm{main}} = \perp$
    $b_{r,\mathrm{pre}} \leftarrow \sigma_{r,\mathrm{main}}$
  upon detecting a full set of pre votes $pv_{r,j_1}, \ldots, pv_{r,j_l}$ for $\sigma \in \{0,1\}$ such that $r_{min} < r \leq r_{\max}$, if $b_{r,\mathrm{main}} = \perp$
    $b_{r,\mathrm{pre}} \leftarrow \sigma$
  upon detecting one pre vote $pv_{r,j}$ for 0 and one pre vote $pv_{r,j'}$ for 1 such that $r_{min} < r \leq r_{\max}$, if $b_{r,\mathrm{main}} = \perp$
    $b_{r,\mathrm{pre}} \leftarrow \texttt{abstain}$.
  upon $b_{r,\mathrm{main}} \neq done$ and $b_{r,\mathrm{main}} \neq \perp$
      Produce the signature share $sh_i^{\texttt{main}}$ on the string
                                  $(\texttt{main vote}, ID, r, b_{r,\mathrm{main}})$.
      Send to all parties a message of the form
                                  $(\texttt{main vote}, r, v, justification, sh_i^{\texttt{main}})$.
      $b_{r,\mathrm{main}} \leftarrow done; r_{min} \leftarrow r$
  /* Step 3: Check for decision. */

  upon detecting a full set of main votes $mv_{r-1,j_1}, \ldots, mv_{r-1,j_l}$for$\rho \in \{0,1\}$, or upon detecting a decision
    message on $\rho$,
      output $(ID, \texttt{out}, \texttt{decide}, \rho)$
      set $r_{\max} \leftarrow r + 1$
      Send to all parties a message of the form
                                  $(\texttt{decision}, \rho, justification)$.
  /* Step 4: Common coin. */

  forall $r, r_{min} + 1 < r \leq r_{\max}$ and $r > 10$,
      upon detecting a full set of valid shares of the coin $(ID, r)$, such that $r_{min} + 1 < r \leq r_{\max}$ and $r > 10$,
        combine these shares to get the value $¢_r \leftarrow F(ID, r) \in \{0,1\}$.
      upon detecting a full set of main votes for round $r$, such that $r_{min} + 1 < r \leq r_{\max}$ and $r > 10$,

        generate a coin share $sh_i^{\texttt{coin}}$ of the coin $(ID, r)$, and send to all parties a message of the form
                                  $(\texttt{coin}, r, sh_i^{\texttt{coin}})$.
end case
```

Figure 4.5: Protocol BASIL (Byzantine Agreement)

**P2.** Two full sets always have at least one non-Byzantine party in common.

**P3.** Each full set of parties contains at least a big set of non-Byzantine parties.

**P4.** If a full set of parties does an action $X$, and no non-Byzantine party does both actions $X$ and $Y$, then no full set of parties does $Y$.

**P5.** Every big set is a small set.

**P6.** Each pair of a full set and a big set of parties intersects.

Step 0 in the protocol forces every party to obtain at least a small set of signature shares on the value it wants to propose in the first pre vote. At least one of these must come of a non-Byzantine party, thus a party can only pre vote $\rho$ in the first pre vote if it was the initial value from some non-Byzantine party. If all non-Byzantine parties were activated with $\rho$, then all acceptable pre votes in the first round are for value $\rho$, thus all acceptable main votes in that round are for $\rho$, as any other vote can not be justified. This results in a decision on $\rho$ in the first round by all non-Byzantine parties that decide, and *validity* follows.

We fix a (given) $ID$. Consider the pre votes cast in round $r \geq 1$. Because of P4, there will be at most one value $b \in \{0, 1\}$ that gathers a full set of pre votes, and we define $\rho_r$ to be this value (if it exists), and otherwise we say that $\rho_r$ is undefined. We say that $\rho_r$ is defined at the point in the game at which sufficient pre votes are cast.

**Lemma 4.16.** *For $r \geq 1$, the following holds (with all but negligible probability):*

*(a) if an non-Byzantine party casts or accepts a main vote of $b \in \{0, 1\}$ in round $r$, then $\rho_r$ is defined and $\rho_r = b$;*

*(b) if an non-Byzantine party casts or accepts a hard pre vote for $b \in \{0, 1\}$ in round $r + 1$, then $\rho_r$ is defined and $\rho_r = b$;*

*(c) if an non-Byzantine party casts or accepts a main vote of* `abstain` *in round $r + 1$, then $\rho_r$ is defined and $\rho_r = 1 - F(ID, r)$;*

*(d) if $r$ is the first round in which any non-Byzantine party decides, then no non-Byzantine party issues a message in round $r + 2$ or later.*

*Proof.* To prove (a), suppose an non-Byzantine party accepts a main vote of $b \in \{0, 1\}$ in round $r$. To be justified, this main vote must be accompanied by a valid threshold signature on the message

$$(\texttt{pre vote}, ID, r, b).$$

By the non-forgeability property of the signature scheme, this implies that a full set of signature shares on this message has been generated. As non-Byzantine parties only generate one share for either 0 or 1, and by P4, we can conclude that $\rho_r$ has been defined and is equal to $b$. That proves (a).

Part (b) now simply follows from the fact that a hard pre vote for $b \in \{0, 1\}$ in round $r + 1$ is justified by the same threshold signature as the main vote from round $r$ in part (a).

Now for part (c). A main vote of `abstain` in round $r + 1$ must be accompanied by a justification for a pre vote of 0 in round $r + 1$ and a justification for pre vote of 1 in round $r + 1$. These pre votes cannot both be soft pre votes, and so one of these two pre votes must be hard. It follows from (b) that this hard pre vote must be for $\rho_r$, and hence the other pre vote must be a soft pre vote for $1 - \rho_r$, and hence $F(ID, r) = 1 - \rho_r$. Part (c) now follows.

Now for part (d). Suppose some non-Byzantine party $P_i$ decides $b \in \{0, 1\}$ in some round $r$, and no party has decided in a previous round. Then in this round, $P_i$ accepted a full set of main votes for $b$. By part (a), we must have $b = \rho_r$. So any other non-Byzantine party who decides in round $r$ must also decide $\rho_r$.

To decide $b$ in round $r$, $P_i$ must have received either a full set of justified main votes or a threshold signature on a full set of main votes for $b \in \{0, 1\}$ in that round. To justify an soft pre vote in round $r + 1$, a full set of signature shares on the message

$$(\texttt{main vote}, r, \texttt{abstain})$$

is required. No non-Byzantine party issues a signature share on main votes for both abstain and $b \in \{0, 1\}$. Along with P4, this implies that a soft pre vote in round $r + 1$ cannot be justified. Thus, the only justifiable pre votes in round $r + 1$ are hard pre votes, and by part (b), these must be hard pre votes for $b$. This implies that the only justifiable main votes in round $r + 1$ are main votes for $b$, and so all main votes accepted by non-Byzantine parties in round $r + 1$ will be main votes for $b$. As (after the first round) every pre vote requires at least one justified main vote for `abstention`, this prevents any non-Byzantine party from casting or accepting a pre vote in round $r + 2$, and thus any later votes. □

*Agreement* follows from part (d). All that remains is to show *deadlock freeness* and *efficiency*.

Suppose the protocol deadlocks, i.e., there is at least one non-Byzantine party that is not decided, yet all messages have been delivered and no non-Byzantine party is active. As decided parties send a message that immediately makes the recipient decide as well, this implies that no non-Byzantine party has decided. Let $m$ be the last (in terms of the protocol logic) vote message send by an non-Byzantine party. Every party that detects such a message will use it as its own vote, provided it did not do the same or a later vote already. Thus, all non-Byzantine parties have send a vote message for that particular vote, and therefore detected a full set of votes. By inspection of the protocol, it is clear that every full set of valid vote message for a particular vote causes the recipient to issue the subsequent vote, provided it did not issue that or a later vote already. The only exception here is a soft pre vote, which cannot be issued unless the corresponding coin is revealed. Suppose all non-Byzantine parties try to issue a soft pre vote in some round $r$. Thus, $m$ is a main vote in round $r - 1$, and by above argument all non-Byzantine parties issue a main vote in round $r - 1$. This causes all non-Byzantine parties to detect a full set of main votes for round $r - 1$, and subsequently reveal their coin share. This in turn opens the coin, and the soft pre vote in round $r$ can be cast.

Therefore, a message for a vote later than $m$ must have been cast, which contradicts the assumption that $m$ is the last (in terms of the protocol logic) vote message. As every non-Byzantine party starts the protocol, and therefore sends some vote message, this is a contradiction and the protocol is deadlock free.

All that remains is *efficiency*. Lemma 4.16 states that in a given round $r + 1$, for $r \geq 1$, the full set of main votes accepted by an non-Byzantine party in step 4 does not contain votes for both 0 and 1. Also, an non-Byzantine party will decide in this round unless it accepts at least one main vote of `abstain`. But if it does accept an `abstain`, then $\rho_r = 1 - F(ID, r)$. The key to showing efficiency will be to show that *the value of $\rho_r$ is determined before the coin $(ID, r)$ is revealed.* Then the (unique) value of justifiable hard pre votes, which can be influenced by the adversary, is fixed before the (unique) value of the soft pre votes, which depends on the coin and can not be influenced by the adversary is revealed. This results in a probability of $\frac{1}{2}$ that they are identical and all justifiable pre votes are identical, which causes the protocol to terminate.

The notion of $\rho_r$ being *determined* is different from the notion of $\rho$ being *defined* introduced above; if it is determined, then the value it will have once being defined is fixed, but it does not need to be defined yet. More precisely, by saying "$\rho_r$ is determined at a particular point in time," we mean the following: There is an efficient procedure $W$ that takes as input a transcript describing the adversary's interaction with the system up to the given point in time (i.e., the initialization values of the corrupt parties and all communication between parties), along with $ID$ and $r \geq 1$, and outputs $w \in \{0, 1, ?\}$. Furthermore, if the output is $w \neq ?$, then if $\rho_r$ ever becomes defined, it must be equal to $w$ (or at least, it should be computationally infeasible for an adversary to cause this not to happen).

By "the coin $(ID, r)$ is revealed at a particular point in time," we mean the point in time when an non-Byzantine party generates a share of the coin $(ID, r)$ such that the shares generated by non-Byzantine parties and the shares of the Byzantine parties (to which the adversary has access) form a full set, i.e., are sufficient to compute the coin value.

**Lemma 4.17.** *There is a function $W$ that determines $\rho_r$, as described above, such that for all $r \geq 1$, either $\rho_r$ is determined before coin $(ID, r)$ is revealed, or $\rho_{r+1}$ is determined before $(ID, r + 1)$ is revealed.*

*Proof.* Suppose that a non-Byzantine party $P_i$ is just about to generate the last share of coin $(ID, r)$ in step 4 of round $r$ that is missing to reveal the coin. Thus, there is a big set $\mathcal{S}$ of non-Byzantine parties who have also reached step 4 of round $r$; this set includes $P_i$, who is just about to release its share; all other members of $\mathcal{S}$ have already released their share. Almost all round $r+1$ pre votes for the parties in $\mathcal{S}$, as well as their justifications, are completely determined at this point, even if these votes have not actually been cast. The only exception are soft pre votes, whose actual value is equal to $F(ID, r)$, which is not yet known.

If any party in $\mathcal{S}$ is going to cast a hard pre vote for $b \in \{0, 1\}$, then by Part (b) of Lemma 4.16, $b$ is the only possible value for $\rho_r$. Thus, $\rho_r$ is already determined—in fact, it is already defined.

Otherwise, all parties in $\mathcal{S}$ are going to either crash or cast soft pre votes, choosing the value $F(ID, r)$ as the value of their round $r+1$ pre vote.

By P6 and the definition of $\rho_{r+1}$, it follows that the only possible value for $\rho_{r+1}$ is $F(ID, r)$.

Therefore, immediately *after* $P_i$ reveals its share of coin $(ID, r)$, $\rho_{r+1}$ is determined. Moreover, if the coin $(ID, r+1)$ has been revealed at this point, then a full set of hard pre votes must have been cast in round $r+1$. Thus, $\rho_{r+1}$ is determined before coin $(ID, r+1)$ is revealed. □

Let $c$ be the number of rounds with a pseudorandom coin (which we set to ten here), plus two. Then this lemma, together with the unpredictability property of sequences of coins described in Section 3.4.3 and Lemma 4.16(d), implies that the probability that any non-Byzantine party advances more than $2r + c$ rounds is bounded by $2^{-r} + \epsilon$, where $\epsilon$ is negligible. *Efficiency* follows immediately, as every party issues at most one message per vote.

## 4.8 Implementation Issues

It seems that until now, all implementations of fault tolerant protocols used in practice have been done using timed algorithms. Randomized protocols, beside a general bias against randomization, where thought of as being inefficient and thus unpractical [Sch00].

To demonstrate the practicability of the randomized approach, a prototype implementation of a modified BASIL protocol has been done in the context of this thesis. As the prototype was intended as a proof of concept to show that the performance of randomized protocols can compete with timed protocols — not only on paper, but also in a real implementation — the prototype was optimized for throughput, leading to an architecture that may not very well support modern program paradigms and that might not integrate smoothly into a greater architecture.

Since then, other implementations of the protocols in this thesis have appeared [CP01, MGB01], with different priorities in mind.

### 4.8.1 Main Structure

The implementation consists of one single process that is responsible for all instances of Byzantine agreement; there is no separation between different transactions on a process or thread level, thus the protocol requires no context switching. As mentioned above, the prototype is a modified version of BASIL; the modifications both made it simpler to implement, and take into account differences between a real world computer and our abstract model. Also, the prototype does not use the adversary structures, but the numerical $t$ out of $n$ model. This is significantly easier to implement, and the increased flexibility of the adversary structures is not needed to demonstrate the performance of the protocol.

To avoid complex message buffering, handling of messages is implemented differently than in BASIL. Instead of keeping messages in an internal buffer until a threshold is exceeded, every message is immediately dealt with (if possible) and then discarded. To this end, the implementation keeps a number of counters for each transaction and each round for which a messages has been send/received in this transaction. These counters count the number of pre-votes for 0 and 1, the main votes, and collect the coin shares; the exact data structure for used for this is shown in Figure 4.6. The actions (i.e., sending messages) then depend on these counters. In processing every message at once, even messages belonging to uninitialized transactions are immediately dealt with (if possible). In this, the implementation violates the model of inter-protocol communication described in Section 2.7.1.

Whenever a message is received, its validity and need is verified. It is an important factor to sort out unnecessary messages before they cause any work. Before a message is evaluated, several checks on the validity of the message are made: First of all, senders that are known to be corrupted – for example, because they send a message with a valid message authentication and an invalid signature – are blacklisted, and their message is discarded. Then, it is verified if the message is still needed; if it belongs to a vote or a coin that already has been processed, the message is dropped without verification as well. After a message passed this these tests, the message authentication and finally the included justifications are verified (if needed). This way, the expensive tests are only done when all other tests succeeded; furthermore, it is only possible for the adversary to make a party verify one invalid justification per dishonest party during the lifetime of the protocol, which makes denial of service attacks a bit more difficult. The order of the checks ensures that the expensive operations are only made after the message passed the cheaper tests.

To keep the implementation simple, threshold signatures have been implemented as a collection of standard RSA signatures, and the simple, non-verifiable common coin from Section 3.4 has been used. This required some twisting of BASIL, as soft pre-votes cannot be verified unless the coin is known. This made the prototype less elegant, and less suitable for purposes beyond demonstrating the performance of the approach.

### 4.8.2 Low Level Communication

To hand control on the message-resending protocols to our protocol, the prototype uses UDP as a communication medium. This gives us full control over the resending mechanism, as opposed to TCP which tries to implement a reliable message delivery itself and thus introduces its own timing assumptions and timeouts.

Unfortunately, UPD might cause problems in a real setting. An overloaded Internet router is likely to drop UDP packets first, as UDP is normally used for applications with a high volume, but little importance of the single packets (like streaming video); a dropped TCP packet however will probably be resent and thus dropping it does not really help.

As it becomes more and more important in Internet applications to support differential services, i.e., to treat IP packets of different applications with different priorities [PDS01], this problem might simply fade away once the next generation of routers is installed.

The implementation allows to choose between various techniques to ensure eventual message delivery, namely positive acknowledgments, negative acknowledgments, optimized negative acknowledgments, and direct or buffered delivery.

**positive acknowledgments** use a technique similar to TCP/IP; every message is acknowledged by the receiver, and messages are resend with an exponential increasing delay until the acknowledgment is received.

**negative acknowledgments** put the responsibility to the receiver; as every party knows which messages to expect, the complain if the message in question is not received in time.

**optimized negative acknowledgments** make use of the fact that not all messages are needed. Instead of waiting for a particular message, a protocol waits to make progress. If no progress is made for a certain time, a complaint is send to all parties, which respond by resending the latest vote message they send.

**direct/buffered delivery** refers to a pre-buffering of the messages in a special sorted buffer. As our protocols might be busy with cryptographic operations and thus not process newly received messages, several messages might queue up by the time the party is ready to process new messages again. In direct delivery, this issue is ignored and messages are directly read out of the UDP-buffer. In buffered delivery, a special data-structure buffers all messages, and the protocol is then delivered the messages in an optimized order (i.e., a message that is likely to render all other messages unnecessary is processed first).

```
struct one_round_t { // Number of votes already received for certain values

int PreVotes0;
int PreVotes1;

int MainVotes0;
int MainVotes1;
int MainVotesX;
// List of coin_shares we got; NULL/-1, if share doesn't exist.

coin_number shares[MAX_PARTIES];

// How many coin shares do we have ?
int no_of_shares;

// Value of the coin; -1, if the coin is not known yet.
int coin;

//Do we still need Pre-Votes/Main-Votes/Coin-Votes for that round ?
// Set to false if the corresponding vote has been done.
// Note:
// 1.  If n-t abstaining main-votes are received, but the coin is not
// available, we wait until getting either a main-vote on (0/1)
// or the coin.  Thus, after n-t abstentions but not having a coin,
// MainVotesNeeded is still true.  This might change in later versions.
// 2.  We always need main-votes for the decision-condition, even if
// MainVotesNeeded is false.  Thus, be careful when to drop a main-vote

bool PreVotesNeeded;
bool MainVotesNeeded;
bool CoinsNeeded;

// Table that shows whose votes where received to avoid
// using the same guys vote twice.

bool PreVotesReceived[MAX_PARTIES];
bool MainVotesReceived[MAX_PARTIES];
bool CoinVotesReceived[MAX_PARTIES];

// Finally, we have to store received signatures somewhere for later use;
// note that we also store a valid proof once we received it/generated it.
// The proof storage is an array if size 2 because we might have proofs for
// both 0 and 1 under certain circumstances.
char signatures[2][MAX_PARTIES+1][SignLength];
bool signature_exists[2][ABSTENTION+1][MAX_PARTIES+1];
char proof[2][ABSTENTION+1][ProofLength];
bool proof_exists[2][ABSTENTION+1];
};
```

Figure 4.6: The main data structure of the prototype

One important test for our protocol was in the crash failure mode, where no digital signatures are needed (see below). In this setting, the bottleneck of our implementation was the network capacity; the extra messages send using positive acknowledgments where a significant slowdown. Negative acknowledgments where rarely to never actually needed, as our local network was quite reliable; thus, the optimization did not make a difference. Also, there was no significant difference between direct and buffered delivery (in fact, direct was even a bit faster). Apparently, the number of rounds the protocol run in average was too small for the buffering to have a significant effect.

In the Byzantine model, where public key signatures where used, the effect of the low-lewel communication was negligible and not measurable.

### 4.8.3 Performance and Scalability

So far, no non-randomized implementations of Byzantine agreement have appeared in the literature that deal with Byzantine faults[4], thus we can compare our protocol only against other protocols in the crash failure model.

The only implementation of an "asynchronous" Byzantine agreement protocol in the literature that I could find is the failure detector protocol by Guerraoui and Schiper [GS96].

The measurements they supply considers a system with 3 parties in a crash-failure model on a fast LAN with broadcast channels, i.e., sending a message to all parties involves only one message to be send. Guerraoui and Schiper achieve 2.3 msec per agreement using SPARCstation20 and a 100Mb/s FDDI Network.

Our implementation of BASIL run on three IBM PowerPC 43P, connected by a 10Mb/s TokenRing Network without broadcast. The performance measured varied between 3.9-4.5 msec per agreement

Naturally, the exact numbers cannot be compared; while we used slightly faster computers, Guerraoui and Schipper had a faster network connection, and in both cases (at least in ours) the performance tests where not designed carefully enough to yield precise enough numbers to compare performance down to the decimal of the constants.

However, the point of our implementation was not to make a detailed performance evaluation, but to show that the performance of our randomized protocol lies in the same order of magnitude as the failure detector approach, while providing more security due to the full asynchrony.

For the Byzantine setting, both our protocols and the protocols in the literature use a comparable amount of public key cryptography (i.e., threshold signatures); it can be assumed that the results roughly translate into this setting.

We have two different threshold signature schemes at hand, (see Section 3.3.4 and [Sho00]), based on communication and communication, respectively. They also can be used in combination, e.g., all pre-votes are signed with one scheme and all main-votes with the other one. This allows an implementation to find a good tradeoff between communication and computation and adapt nicely to any environment.

In scaling up to more participants, there is (theoretically) no limit for the number of participants involved. Practically, however, one has to keep in mind that it is inherent in the protocols that every party communicates with every other party (or, at least with $t + 1$ other parties).

Thus, the effort in computation and communication per party is linear in the number of participants.

The total communication effort is thus quadratic in the number of participants; when this starts posing a problem depends highly on the network used. If the network is the Internet, it might take a while until the communication effort caused by this protocol becomes significantly visible among the normal noise.

[4]It sounds a bit strange that nobody seems to have implemented such a fundamental protocol (or, at least not publicized it widely). I belief that the reason is that for the practitioners, the full Byzantine model is out of fashion since about ten years and people are only slowly moving back towards this model; furthermore, the most influential projects on fault tolerant computing aim higher and offer higher level primitives like atomic broadcast and virtual synchrony, on top of which Byzantine agreement could then be build — naturally, comparing the efficiency of our protocol with such an indirect implementation would be unfair and not yield any usable results

In an simulation done at Stanford University [MGB01], an implementation of the full Byzantine version of ABBA has been tested with up to 148 participants on an slow network. Their performance lies in the order of five minutes per agreement; the implementation was however not at all optimized for speed, as the use case they had in mind did not impose any significant timing restrictions.

There are no comparable numbers for failure detector protocols in the literature. The Ensemble system, which provides higher level protocols (comparable to the atomic broadcast in Chapter 8) that rely on failure detectors in the crash-failure model, is optimized for relatively small groups (i.e., 30 to 40 parties) and thus behaves "sort of thrashing" when scaled up too much [Bir01]. This illustrated another advantage of asynchronous protocols; timed protocols require a quite precise tuning to the environment and group size, as too high timeouts lead to unnecessary idle times, while to small timeouts may result in false alarms.

## 4.9 Optimistic Byzantine agreement

To guarantee the maximum possible degree of security, our protocols are designed for a very strong adversary – under normal circumstances, an adversary is far more limited than our model proposes.

In this section, we use the *optimistic aspiration* (see Section 2.6.2). This aspiration does not weaken the adversary at all; it still can do everything he could do before. However, we (optimistically) hope that most of the time, our protocols are not under attack and the environment is not behaving hostile. By first assuming a non-hostile environment, we can increase the performance of our protocols by several orders of magnitude; it fact, the optimistic protocol is as efficient as any synchronous protocol can be. If the environment turns out to be less friendly, the performance is slightly lower compared to non-optimistic protocols, but the security remains unharmed.

The optimistic Byzantine agreement protocol works in two phases; first, it tries to reach agreement using a unreliable, but fast synchronous protocol. It then tests the result, and if any inconsistencies are detected, the asynchronous, reliable fallback protocol is invoked. It is well possible that some parties decide within the optimistic part of the protocol, while others decide only in the fallback protocol; the protocol logic guarantees that agreement holds anyway.

In the optimistic case, i.e., network timing assumptions hold and no adversary exists, the protocol executes only the optimistic pre-protocol, and agreement is reached in optimal time. Furthermore, no expensive computation is needed. In a real system with somewhat predictable network behavior, the probability that a transaction falls into this case is quite high, and for efficiency considerations this case is the important one.

In the pessimistic case, the optimistic pre-protocol invokes the fully asynchronous Byzantine Agreement protocol (the *fallback protocol*) without endangering any liveness or safety goals. Furthermore, if few failures occur (i.e., a small number of traitors and network failures occur), the fallback protocol is invoked with all honest parties having the same start value. This causes most Byzantine Agreement protocols, in particular ours, to terminate quite fast.

We assume that we have access to a protocol *ABA* that solves Asynchronous Byzantine Agreement. The protocols ABBA (Figure 4.2) and BASIL (Figure 4.5 are suitable for these purposes. Also, if a multivalued fallback protocol is offered (as the one presented in Chapter 6), the optimistic protocol works on the multivalued domain as well.

As ABA is only invoked in the pessimistic case, its efficiency plays a secondary role; however, the more efficient the fallback protocol is, the more aggressive timeouts can be chosen for the optimistic protocol. The protocol ABBA (Figure 4.2) is suitable for these purposes. Also, if a multivalued fallback protocol is offered (as the one presented in Chapter 6, the optimistic protocol works on the multivalued domain as well.

### 4.9.1 Protocol start, validity and the origin of the input values

In the optimistic model, the way the protocol is started is an important and subtle matter. If different parties start the protocol asynchronously — which is well possible even in a well behaved, adversary free network — the protocol might immediately go pessimistic and the entire notion of an optimistic protocol does not make much sense.

We can assume a synchronous start to rid ourself of this problem, or have a well specified time at which parties receive their inputs; this is the most convenient approach, but — depending on the real-world setting — it can be either unrealistic or kill the performance gained by the new protocol, if the assumed asynchrony in the starting-times is much larger than the assumed asynchrony of the network.

Alternatively, it is possible to alter the definition of validity such that a party is capable to take over the initial value of some other party if it did not receive its own initial value in time.

The easiest way to allow for this is to replace validity by *weak validity*:

**Weak validity:** If all parties are honest, and all parties have been activated with on a given $ID$ with the same initial value, the all parties must decide this value.

In this definition, if only one party proposes some value, the protocol is allowed to decide that value. This means that parties that do not receive their own initial value in time (or not at all) can take over the initial value of some other party. It also means that if this value comes from a dishonest party, then the decision value might be something no honest party initially wanted.

In some settings, this definition is too weak; especially if the domain of input values goes beyond binary values (multivalued Byzantine agreement), the protocol might output complete nonsense. To prevent this from happening, we want to ensure that the proposed values make sense; for example, if some people want to agree on a movie to go to, the agreement value should indeed be a movie running in a nearby theater.

This leads us to the definition of *external validity*:

Let $Q_{ID}$ be a predicate that determines if an input value is valid or not. A protocol solves *validated Byzantine agreement* with predicate $Q_{ID}$ if it solves Byzantine agreement where the *validity* condition is replaced by the following condition:

**External Validity:** Any honest party that terminates for $ID$ decides $v$ validated by $\pi$ such that $Q_{ID}(v, \pi)$ holds.

If one reads our definitions of external validity by the letters of the law, there is a simple protocol that solves agreement without any communication; simply test all possible combinations of $(v, \pi)$ until one is found that satisfies the verification.

One could rule out this protocol by adding the weak validity condition in addition to the external validity. Our protocols would still satisfy this definition; furthermore, a combination of external and weak validity can be used to ensure validity in the traditional sense.

We ignore this issue, however, for the following reasons:

– In most reasonable settings, finding a matching pair $(v, \pi)$ by guessing is as difficult as forging a signature; this is infeasible by the cryptographic assumptions.

– If the decision value passes the verification, it is a good decision, even if nobody initially proposed it. So there is nothing wrong applying the trivial protocol if possible.

We will need this definition of validity again when we introduce a protocol for multivalued Byzantine agreement in Chapter 6. For now, we will assume a synchronous start for simplicity; the modifications needed to use external validity are shown later.

The activation of our optimistic protocol is as before. We assume access to an asynchronous protocol ABA (the *fallback protocol*) that solves Byzantine agreement. The protocol ABA must either satisfy the stronger notion of validity as defined in Section 4.2, i.e., if all honest parties start the protocol with the same initial value, then this is the decision value. Alternatively, this property can be simulated by a combination of weak validity and external validity. The mechanism for this is the same as used in the pre-phase in ABBA (Figure 4.2), i.e., a simple majority vote is performed and the main protocol is started with values that (provably) are the majority among $n - t$ input values.

Note that in the multivalued setting, it is possible that every party submits a different initial value to the majority vote, i.e., the threshold signature scheme in [Sho00] is not applicable. We do not know how to do the proofs of majority with communication complexity $O(n)$ in this case.

Optimistic-ABA works on the same set of input values as ABA. Especially, if ABA is multi-valued, then so is Optimistic-ABA. The efficiency of ABA is only relevant in the presence of an adversary or network misbehavior.

The optimistic protocol does not need to put any further restrictions on the adversary, e.g., bounded computation power or static corruption, as long as the adversary cannot authenticate as an honest party. However, the asynchronous Byzantine agreement protocol might require a more restricted adversary model (as ABBA and BASIL do).

### 4.9.2 Optimistic-ABA

We now present the optimistic protocol. Each party $P_i$, $1 \leq i \leq n$, gets an input value $v_i$ and a corresponding transaction identifier $ID$. The protocol outputs some decision value $\rho$ or invokes the fallback protocol ABA. It is possible that a party decides $\rho$ in the optimistic protocol and still invokes the fallback protocol. In this case, the decision of ABA is ignored. It is important to note that if an honest party decides $\rho$ in the optimistic part of the protocol, then the decision in ABA can only be $\rho$, as will be shown in the validity paragraph in the proof.

In a nutshell, *Optimistic-ABA* works in four phases:

**Simple Agreement:** A simple non-Byzantine agreement is invoked by every party broadcasting its preference and waiting for all other parties to answer.

**Commit and Check for Decision:** Every party commits to the value it perceives as the output of the simple agreement protocol. If it receives $n$ identical commitments from all parties, then it can *decide*.

**Decide and Hibernate:** A party that decided cannot completely terminate the protocol, as it is possible that some other party could not reach decision. Instead, decided parties hibernate; they remember their decision for this transaction, but do not do anything unless a complaint is received.

**Complain and Recover:** A party that gets either inconsistent or not enough answers broadcasts a complaint; on receiving such a complaint, an honest party starts the "pessimistic part" of the protocol, i.e., enters the fallback protocol ABA.

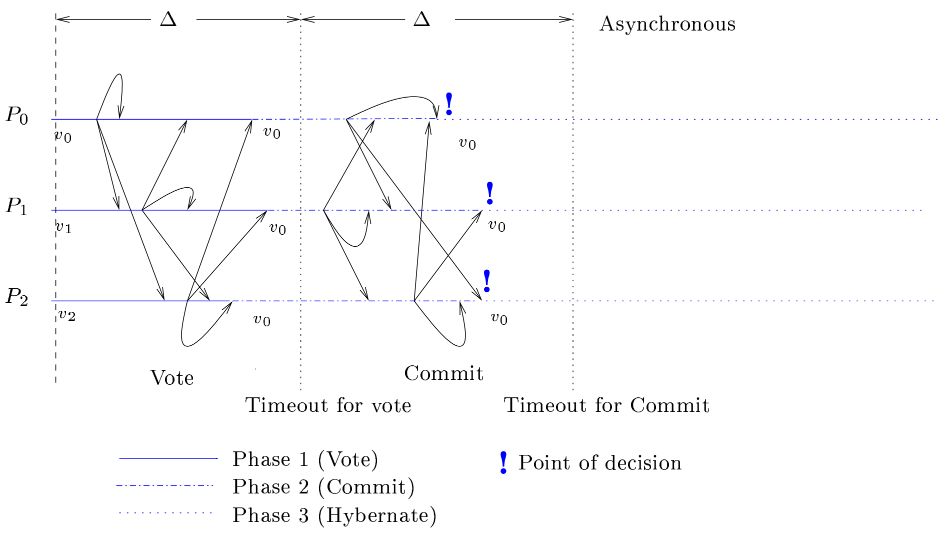


Figure 4.7: Normal flow of opt-ABBA

Figure 4.7 shows the message flow in the normal case, i.e., all parties decide optimistically and the pessimistic protocol is not invoked.

Figure 4.8 shows a situation where one message (indicated be the dotted line) is not sent. Party $P_2$ does not receive sufficiently many messages in phase two, and after time $2\Delta$ it could not make a decision. Thus, it invokes the pessimistic protocol, which also forces the already decided parties to wake up from hibernation and join the recovery phase.

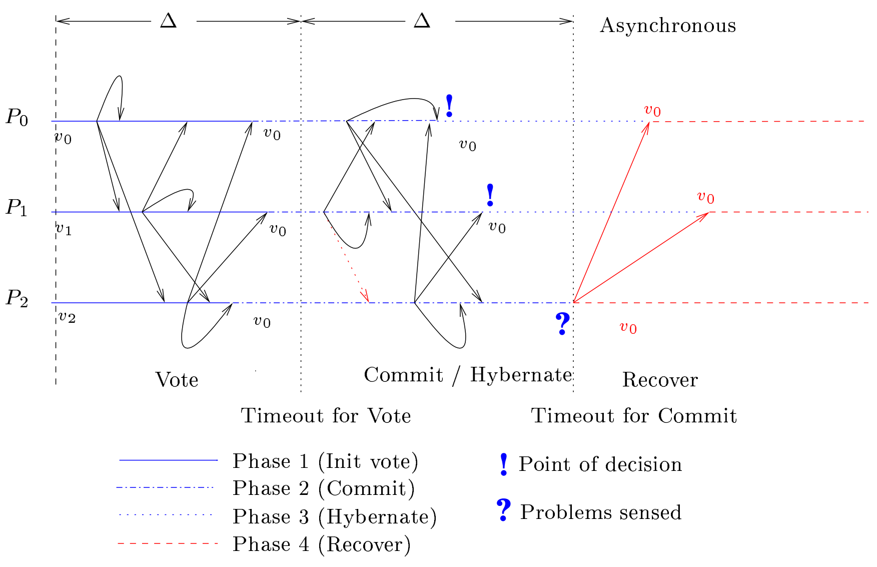


Figure 4.8: Pessimistic flow of opt-ABBA

Note that if one honest party decides $v_0$ optimistically, all honest parties committed to $v_0$ and thus will enter ABA with initial value $v_0$. This will cause ABA to decide $v_0$ as well, maintaining agreement between the optimistic and the pessimistic decisions.

The protocol is show in Figure 4.9.

## 4.9.3 Proof of Correctness

We are now going to show that the optimistic protocol solves the Byzantine agreement problem.

**Theorem 4.18.** *Given an asynchronous Byzantine agreement protocol ABA as defined above, if the adversary faithfully delivers all messages and timeout signals, the protocol Optimistic-ABA solves Byzantine agreement for $n > 3t$.*

*Proof.* To prove the correctness of our protocol, we have to show that the protocol satisfies validity, agreement and termination.

**Validity.** Suppose all honest parties start with $\rho$. All honest parties that send a commit message will commit to either the majority of the received init-votes or their own initial value, both being $\rho$.

This implies that no party can see more than $t$ commit-votes on something different from $\rho$. This makes it impossible for an honest party to decide anything but $\rho$ optimistically; every honest party either decides $\rho$ or sends the pessimism message.

Any honest party that enters the fallback protocol ABA will have the initial value $\rho$, as this is the simple majority of the received init-votes. If one honest party starts the fallback protocol, then it sent out a pessimism-message to all parties, and therefore all honest parties start the fallback protocol, provided the adversary delivers all messages. Thus, all honest parties start ABA with the same initial value, and by the validity of ABA this will be the decision value.

**Agreement.** Suppose one honest party decides $\rho$ before entering ABA. Then this party received $n$ commit-votes for $\rho$, $n - t$ of which have been issued by honest parties.

Case 1: If all honest parties decide before entering ABA, all decide $\rho$ and the protocol is finished. It still might happen that a corrupted party sends a `pessimism` message to some parties. This might cause them to continue and enter ABA, but the result is irrelevant and does not change their decision.

```
Protocol Optimistic-ABA for party $P_i$

/* Vote. */

wait for a message $(\texttt{in}, \texttt{opt-propose}, v_i)$,
    start timer
    send to all parties the message

        $(\texttt{init-vote}, v_i)$

/* Commit. */

wait for a timeout signal from the timer or $n$ init-votes, if no commit message has been sent by $P_i$,

    $v_i^{\texttt{commit}} \leftarrow \begin{cases} \text{simple majority of the init votes} & \text{,if } n \text{ votes where collected} \\ v_i & \text{,otherwise} \end{cases}$

    send to all parties the message

        $(\texttt{commit}, v_i^{\texttt{commit}})$.

    start timer

case /* Check for Decision. */

    upon receiving $n$ commitments for some value $\rho$
        stop timer
        output $(\texttt{out}, \texttt{decide}, \rho)$

    upon receiving a timeout signal from the timer, if a commit message has been sent,
        send to all parties the message

            $(\texttt{pessimism})$.

end case
wait for a pessimism message
    if $P_i$ did not broadcast a pessimism-message yet,
        send to all parties the message

            $(\texttt{pessimism})$.

    invoke ABA on transaction $ID$ with input value $v_i^{\texttt{commit}}$.
```

Figure 4.9: Optimistic-Byzantine Agreement (OPT-BA)

Case 2: Suppose that some honest party did not decide optimistically. This party will broadcast the `pessimism` message. All honest parties will, on receiving this message, enter ABA with the vote they committed to; if an honest party decided $\rho$, all honest parties committed to $\rho$, and thus enter ABA with $\rho$. By the validity condition of ABA, this guarantees agreement.

In the case that no honest parties decides before entering ABA, the agreement condition in ABA enforces agreement of the Optimistic-ABA protocol.

**Deadlock Freeness.** Any honest party that gets activated immediately starts a timer. If all timeout signals are delivered, the commit phase will be reached either by timeout or by receiving $n$ vote messages, After this, an honest party will either decide, or issue a pessimism message due to inconsistencies or a timeout of the timer. If a pessimism message is sent to all honest parties, deadlock freeness follows by the deadlock freeness of the fallback protocol ABA.

**Efficiency.** An honest party only sends two messages in addition to the fallback protocol ABA. Thus, efficiency follows from the efficiency of ABA.

□

Note that the optimistic protocol cannot entirely terminate after deciding; it is possible that it is woken up later and has to help other parties deciding, though its own decision can not change anymore. Depending on the system architecture, that might be a problem; if, for example, every transaction is a separate thread, this thread might never be completely terminated. If an implementation takes this into account, this problem is reduced to each party needing to store some information about each finished transaction, which it has to to anyway to avoid starting the same transaction twice.

### 4.9.4 Performance

The performance of the protocol has to be examined both in the optimistic and in the pessimistic case. Nevertheless, the optimistic case has more impact on the overall performance, assuming the timeouts are chosen carefully. It is not possible to make a combined analysis in our model, as this would require a more refined network model that is not oriented to the worst case, but on the average behaviour of a real network.

**The optimistic case.** Let $\Delta$ be the time the timer runs before timing out, and $d$ the longest delay a message has during the run of the protocol.

Suppose first that all parties are honest and all timing assumptions hold, i.e., the timer does not timeout. Then, all honest parties are decided after time $2d$. This is optimal, as two synchronous rounds is the optimum even for fully synchronous protocols [DRS90]

No public-key cryptography is needed, so the computing effort for all parties is quite low.
The message complexity is $n$ messages per party and phase, resulting in $2n^2$ messages altogether. As long as the initial values have a constant size, all messages have constant size, so the bit-complexity is $O(n^2)$ as well, with a relatively small constant — in fact, in a realistic communication like the Internet the message size will be dominated by information for the communication sublayer or the size of the agreement value.

So far, no asynchronous Byzantine agreement protocol can reach agreement with two rounds of communication. The best failure-detector implementations [Rei95, KMMS97, DS98] need at least four rounds. Furthermore, they require public key signatures, which are relatively expensive to generate.

**The pessimistic case.** In the pessimistic case, the protocol basically performs like the fallback protocol ABA, with an overhead created by the optimistic part. The time needed to detect the pessimistic case and start ABA is smaller than $2\Delta$. The number of messages is $3n^2$.
If one honest party decides optimistically, then all parties will enter ABA with the same initial value. If ABBA or BASIL is used as a fallback protocol, this triggers termination within one asynchronous round.

### 4.9.5 Asynchronous start

The model assumed so far is convenient to work with, but a synchronous start of all parties may not be realistic. We now describe the changes needed for an asynchronous start, assuming an external validity.

To make the algorithm work, the *init-vote* has to be modified. Any party that is aware that the protocol starts has to inform the other ones. We denote the verification information for a vote $v$ by $\Sigma_v$; if a party sees a vote for $v$ for the first time, it verifies the validity and only accepts the message if the verification was accepted.

```
1. Init-Vote
   upon receiving a message ($\texttt{in}, \texttt{opt-propose}, v_i, \Sigma_{v_i}$), or a message ($\texttt{init-vote}, v_i, \Sigma_{v_i}$) from $P_j$,
      start timer
      send to all parties the message ($\texttt{init-vote}, v_i, \Sigma_{v_i}$)
```

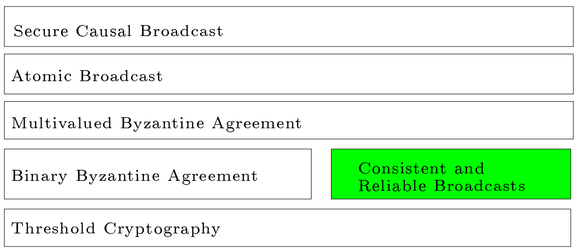
The modifications to the proof are straightforward; external validity is clearly satisfied by the fact that invalid proposals are simply ignored, while termination and agreement are not influenced by the modification; all that happens is that parties have another option on how to start the protocol.

This modification has some impact on the system model. The modified optimistic agreement protocol is the first protocol that can be activated without a specific start-message from the adversary. For a real implementation, this means the protocol has to be invoked before its input value is known, like it is the case for the broadcast protocols introduced in Chapter 5.

If the parties do not start synchronously, we need up to three synchronous rounds to reach agreement. Furthermore, the predicate for external validity might include the verification of a signature, which adds additional computing effort.

# Chapter 5

# Low Level Broadcasts


Secure Causal Broadcast
Atomic Broadcast
Multivalued Byzantine Agreement
Binary Byzantine Agreement
Consistent and Reliable Broadcasts
Threshold Cryptography

This chapter introduces a generic definition for broadcast protocols, as well as two broadcast primitives, *consistent broadcast* and *reliable broadcast* [CKPS01]. We also introduce the concept of a *verifiable* broadcast. Consistent and reliable broadcast do not need Byzantine agreement, and fully asynchronous solutions exist in the literature [Bra84, Rei94]. The protocols presented in this chapter are extensions and optimizations of those protocols.

## 5.1 The Generic Broadcast Model

In a nutshell, a generic[1] broadcast protocol allows one party (the sender) to send a message to the other parties (the receivers). In addition, a number of additional guarantees can be required, for example that a dishonest sender cannot make honest parties receive different messages in the same broadcast instance.

To distinguish between messages sent between parties to implement the broadcast protocol, and the messages broadcasted by it, the latter ones will from now on be referred as *payload messages* or *requests*. As in Byzantine agreement, different instances of the protocol have to be distinguished, and therefore each broadcast is parameterized by a tag *ID*. However, we must be a bit more careful here; if the tags are completely determined by the adversary, it could cause two honest parties to act as senders for the same broadcast, which can lead to unnecessary complications.

Therefore, we extend the syntax of the transaction identifier. In addition to the adversary generated tag *ID* it contains the identity of the sender, as well as a sender specific sequence number that is used to separate different broadcast instances.

Furthermore, we want to group instances of the broadcast protocol in a way that we get virtual "channels". This is not necessary for all broadcast primitives presented here, but important once the different instances of the broadcast protocol interfere with each other, as in the atomic broadcast introduced later.

[1] By generic, we mean that this part is identical for all broadcasts presented in this thesis. This is not to be confused with the *generic broadcast* defined by Pedone and Schiper [PS99], which is a generalization of atomic broadcast that takes into account message semantics.

Thus, the tag $ID$ consists of three parts: a channel identifier $ID_c$, the sender's identity $j$ and a sequence number $s$ that separates broadcasts by one party on a given channel. We restrict the adversary to activate broadcast tagged with $ID_c|j|s$ for party $P_i$ only if $i = j$ and at most once for every sequence number. These requirements are easily satisfied in practice by maintaining a message counter. Instances of broadcast are always identified by $ID = ID_c|j|s$.

A broadcast protocol is activated when the adversary delivers a message to $P_j$ of the form

$$(ID, \texttt{in}, \texttt{broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say $P_j$ *broadcasts* $m$ *tagged with* $ID$, or simply $P_j$ *broadcasts* $m$. Note that only $P_j$ is activated like this. The other parties are activated when they perform an explicit *open* action for instance $ID$ in their role as receivers; according to our convention, this occurs for instance when they **wait for** an output tagged with $ID$.

A party terminates a broadcast of $m$ tagged with $ID$ by generating an output message of the form

$$(ID, \texttt{out}, \texttt{deliver}, m).$$

In this case, we say $P_i$ *delivers* $m$ *tagged with* $ID$ (or *delivers* for brevity).

We say that all protocol messages which are generated by honest parties have tags with prefix $ID = ID_c|j|s$ are *associated* to the broadcast of $m$ by $P_j$ with sequence number $s$. Recall that this defines also the messages contributing to the communication complexity of the protocol instance $ID$. sec:liveness As described in Section 2.4, the standard *liveness* property is broken into two properties:

The *deadlock-freeness* property rules out trivial protocols that do not do anything useful (for example, a protocol that does nothing at all). One could use an equivalent, but simpler definition here, requiring that only the sender (and not all honest parties) deliver the message; but then one would have to modify this again to the present form for defining consistent broadcast below.

The *efficiency* property guarantees that whatever happens, happens quickly, i.e., with only a polynomial number of messages sent between honest parties. Together with deadlock-freeness this ensures liveness in the classical sense.

In the generic model, we only define one safety property, being *validity*, i.e., the output of the protocol is linked to the input.

**Definition 5.1.** A generic broadcast protocol satisfies the following conditions, except with negligible probability:

**Deadlock Freeness:** It is infeasible for the adversary to create a situation where for some $ID$, $m$ has been broadcasted by an honest party $P_j$, yet there there are honest parties that did not *deliver* $m$ tagged with $ID$, and all honest parties have been activated on $ID$, and all messages associated with $ID$ have been delivered.

**Efficiency:** For any $ID$, sender $j$, and sequence number $s$, the communication complexity of instance $ID$ is polynomially bounded.

**Validity:** For all $ID$, senders $j$, and sequence numbers $s$, every honest party *delivers* at most one message $m$ tagged with $ID$. Moreover, if all parties are honest, then $m$ was previously *broadcast* by $P_j$ with sequence number $s$.

Our definition of validity may seem weak, since our model assumes authenticated links and we could hope to get the guarantee in the second clause also with $t$ actually corrupted parties. Indeed, most reliable broadcast protocols implicitly also authenticate the sender of a message. It is possible to define the corresponding notion of an *authenticated* generic broadcast by replacing the validity condition above by the following.

**Authenticity:** For all $ID$, senders $j$, and sequence numbers $s$, every honest party *delivers* at most one message $m$ tagged with $ID$. Moreover, if $P_j$ is honest, then $m$ was previously *broadcast* by $P_j$ with sequence number $s$.

However, we will not use authenticity in the standard definitions below because only some of our protocols provide authenticity. In particular, the protocols for reliable and for consistent broadcast provide authenticity, but not the atomic broadcast protocol.

The property we call validity is sometimes called integrity in the literature [HT94]. The reason for this mismatch is that there is little consensus on a common notation in fault tolerant agreement– and broadcast protocols. While most of the literature on agreement protocols sees validity as the safety property that ensures that the output of the protocol depends on the input, in broadcast protocols the most common approach is to use the term "integrity" for this purpose and the term "validity" to ensure liveness (taking the role of "termination" in agreement protocols). As the notation is not very well defined anyway, our definitions mainly try to sustain an internal logic, i.e., properties with the same name (roughly) serve the same purpose in all definitions.

While above properties are common for all broadcasts presented here, additional properties are necessary to define useful broadcast services. In this chapter, we define three additional properties: verifiability, consistency, and reliability. In a nutshell,

**a verifiable** broadcast allows an honest party to prove that it terminated the broadcast protocol and received the payload message following the protocol;

**a consistent** broadcast guarantees that all honest parties that receive a payload message for one protocol instance $ID$ receive the same message;

**a reliable** broadcast guarantees that if one honest party delivers a payload message for one protocol instance $ID$, then all honest parties do.

In the following, we will describe how the individual properties can be implemented. To distinguish different kinds of broadcast, we will use a prefix, e.g., r-broadcast and r-deliver for a reliable broadcast.

## 5.2 Verifiable Broadcast

A party $P_i$ that has delivered a payload message may want to inform another party $P_j$ about this. Such information might be useful to $P_j$ if it has not yet delivered the message, but can exploit this knowledge somehow, in particular if $P_j$ is guaranteed to deliver the same message later. This is not always possible by default; in the original reliable broadcast protocol in [Bra84], this knowledge cannot be transferred in a verifiable way.

We formalize this property of a broadcast protocol here because it is useful in our applications, and call it *verifiability*. Informally, it means this: when $P_j$ claims that it is not yet in a state to deliver a particular payload message $m$, then $P_i$ can reply with a single protocol message and when $P_j$ processes this, it will deliver $m$ immediately and terminate the corresponding broadcast.

**Definition 5.2 (Verifiability).** A broadcast protocol is called *verifiable* if the following holds, except with negligible probability: When an honest party has delivered $m$ tagged with $ID$, then it can produce a single protocol message $M$ that it may send to other parties such that any other honest party will deliver $m$ tagged with $ID$ upon receiving $M$ (provided the other party has not already delivered $m$).

We call $M$ the message that *completes* the verifiable broadcast. This notion implies that there is a predicate $V_{ID}$ that the receiving party can apply to an arbitrary bit string for checking if it constitutes a message that completes a verifiable broadcast tagged with $ID$.

## 5.3 Consistent Broadcast

A *consistent broadcast* guarantees that if two honest parties receive a message, they receive the same message. Several protocols for consistent broadcast have been proposed by Reiter et al. [Rei94, MMR00]. The consistent broadcast provides no agreement on the set of messages delivered; some parties can be completely unaware of the existence of messages that other parties receive. It is however possible that the party learns about the existence of messages by higher level protocols, for example when executing the protocol VBA in Section 6.3.

### 5.3.1 Problem Statement

A consistent broadcast protocol is activated when the adversary delivers a message to $P_j$ of the form

$$(ID, \texttt{in}, \texttt{c-broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say $P_j$ *consistently broadcasts* $m$ *tagged with* $ID$.

A party terminates a consistent broadcast of $m$ tagged with $ID$ by generating an output message of the form

$$(ID, \texttt{out}, \texttt{c-deliver}, m).$$

In this case, we say $P_i$ *consistently delivers* $m$ *tagged with* $ID$. To distinguish consistent broadcast from other forms of broadcast, we will use the terms *c-broadcast* and *c-deliver*.

All protocol messages generated by honest parties and tagged with $ID$ are associated to the broadcast of $m$ by $P_j$ with sequence number $s$.

**Definition 5.3 (Consistent Broadcast).** A protocol for *consistent broadcast* is a generic broadcast that additionally satisfies the following condition except with negligible probability:

**Consistency:** If some honest party *c-delivers* $m$ tagged with $ID$ and another honest party *c-delivers* $m'$ tagged with $ID$, then $m = m'$.

In other words, consistent broadcast makes no provisions that two parties do deliver the payload message, but maintains consistency among the actually delivered messages with the same senders and sequence numbers.

### 5.3.2 A Protocol for Verifiable Consistent Broadcast

Protocol VCBC implements verifiable consistent broadcast and is described in Figure 5.1. It uses a non-interactive $(n, \lceil \frac{n+t+1}{2} \rceil, t)$-dual-threshold signature scheme $\mathcal{S}_1$ with verifiable shares according to Chapter 3. Recall that all messages are authenticated according to our basic system model.

The protocol is based on the "echo broadcast" of Reiter [Rei94], but uses a threshold signature to decrease the communication complexity. The idea behind it is that the sender broadcasts the message to all parties and hopes for $\lceil \frac{n+t+1}{2} \rceil$ parties to sign it as "witnesses" to guarantee consistency. The signature shares are then collected by the sender and combined to a threshold signature on the message; it then sends the signature all parties. After receiving the message together with a valid signature, a party delivers it immediately.

Because a party may forward the message and the signature to other parties, the protocol is also verifiable according to Definition 5.2. The corresponding interface is implemented by the `c-request` and `c-answer` messages, which are not otherwise used by the protocol.

The consistency property of the protocol is based on the following lemma.

**Lemma 5.1.** *For all senders $j$, sequence numbers $s$, and strings $ID$, it is infeasible for the adversary in Protocol* VCBC *to create valid $\mathcal{S}_1$-signatures on the strings $(ID, \texttt{c-ready}, m)$ and $(ID, \texttt{c-ready}, m')$ with $m \neq m'$.*

```
Protocol VCBC for party $P_i$ and tag $ID$
 Initialization:
         $\bar{m} \leftarrow \perp; \bar{\mu} \leftarrow \perp$
         $W_d \leftarrow \emptyset; r_d \leftarrow 0 \qquad (d \in \{0,1\}^{k'})$
 case

        upon receiving a message $(ID, \texttt{in}, \texttt{c-broadcast}, m)$:
            send $(\texttt{c-send}, m)$ to all parties
        upon receiving a message $(\texttt{c-send}, m)$ from $P_l$:
            if $j = l$ and $\bar{m} = \perp$ then
              $\bar{m} \leftarrow m$
              compute an $\mathcal{S}_1$-signature share $\nu$ on $(ID, \texttt{c-ready}, H(m))$
              send $(\texttt{c-ready}, H(m), \nu)$ to $P_j$
        upon receiving a message $(\texttt{c-ready}, d, \nu_l)$ from $P_l$ for the first time:
            if $i = j$ and $\nu_l$ is a valid $\mathcal{S}_1$-signature share then
              $W_d \leftarrow W_d \cup \{\nu_l\}$
              $r_d \leftarrow r_d + 1$
              if $r_d = \lceil \frac{n+t+1}{2} \rceil$ then
                combine the shares in $W_d$ to an $\mathcal{S}_1$-threshold signature $\mu$
                send $(\texttt{c-final}, d, \mu)$ to all parties
        upon receiving a message $(\texttt{c-final}, d, \mu)$:
            if $H(\bar{m}) = d$ and $\bar{\mu} = \perp$ and $\mu$ is a valid $\mathcal{S}_1$-signature then
              $\bar{\mu} \leftarrow \mu$
              output $(ID, \texttt{out}, \texttt{c-deliver}, \bar{m})$

        /* Implementation of verifiability property */
        upon receiving message $(\texttt{c-request})$ from $P_l$:
            if $\bar{\mu} \neq \perp$ then
              send $(\texttt{c-answer}, \bar{m}, \bar{\mu})$ to $P_l$
        upon receiving message $(\texttt{c-answer}, m, \mu)$ from $P_l$:
            if $\bar{\mu} = \perp$ and $\mu$ is a valid $\mathcal{S}_1$-signature on $(ID, \texttt{c-ready}, H(m))$ then
              $\bar{\mu} \leftarrow \mu$
              $\bar{m} \leftarrow m$
              output $(ID, \texttt{out}, \texttt{c-deliver}, \bar{m})$
 end case
```

Figure 5.1: VCBC for verifiable and authenticated consistent broadcast.

*Proof.* Suppose not. Then, assuming $\mathcal{S}_1$ is secure, there are at least $\lceil \frac{n+t+1}{2} \rceil - t$ signature shares from distinct honest parties on a message containing *ID* and $m$ and at least as many from honest parties on the message containing *ID* and $m'$. In total, there are $n + t + 1 - 2t = n - t + 1$ or more shares generated by honest parties containing *ID*. Since there are only $n - t$ honest parties, at least one honest party has signed two different messages with the same sender $j$ and sequence number $s$, which is impossible according to the protocol. □

**Theorem 5.2.** *Assuming $\mathcal{S}_1$ is a secure $(n, \lceil \frac{n+t+1}{2} \rceil, t)$-dual-threshold signature scheme, Protocol* VCBC *provides verifiable and authenticated consistent broadcast for $n > 3t$.*

*Proof. Deadlock Freeness* for an honest sender is obvious from the construction of the protocol since all honest parties generate a signature share on $m$ as soon as they receive an `c-send` message containing $m$. Since at least $\lceil \frac{n+t+1}{2} \rceil$ honest parties return them to the sender, it can combine them to a valid signature and *c-deliver* the message.

The *consistency* property follows directly from Lemma 5.1 because an honest party *c-delivers* a payload message only after verifying the corresponding threshold signature.

*Validity* follows directly from Lemma 5.1 together with the logic of the protocol, where $\bar{\mu} \neq \perp$ is used to represent the state in which $\bar{m}$ has already been *c-delivered.* The protocol provides also *authenticity* because honest parties process `c-send` messages only from the sender indicated by the message.

Finally, *efficiency* is straightforward to verify and *verifiability* is ensured by the `c-answer` protocol message, which is generated upon receiving a suitable `c-request`. □

The message complexity of Protocol VCBC is $O(n)$ and its bit complexity is $O(n(|m| + K))$, assuming the length of a threshold signature and a signature share is at most $K$ bits.

## 5.4 Reliable Broadcast

Reliable broadcast provides a way for a party to send a message to all other parties. It requires that all honest parties deliver the same set of messages and that this set includes all messages broadcast by honest parties, without guaranteeing anything about the order in which messages are delivered. In the context of arbitrary faults, reliable broadcast is also known as the *Byzantine generals problem* [LSP82].

### 5.4.1 Problem Statement

A reliable broadcast protocol is activated when the adversary delivers a message to $P_j$ of the form

$$(ID, \texttt{in}, \texttt{r-broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say $P_j$ *reliably broadcasts $m$ tagged with ID*.

A party terminates a reliable broadcast of $m$ tagged with *ID* by generating an output message of the form

$$(ID, \texttt{out}, \texttt{r-deliver}, m).$$

In this case, we say $P_i$ *reliably delivers $m$ tagged with ID*. To distinguish reliable broadcast from other forms of broadcast, we will use the terms *r-broadcast* and *r-deliver.*

All protocol messages generated by honest parties and tagged with *ID* are associated to the broadcast of $m$ by $P_j$ with sequence number $s$.

**Definition 5.4 (Reliable Broadcast).** A protocol for *reliable broadcast* is a generic broadcast that additionally satisfies the following conditions except with negligible probability:

**Consistency:** If some honest party *r-delivers* $m$ tagged with $ID$ and another honest party *r-delivers* $m'$ tagged with $ID$, then $m = m'$.

**Totality:** If some honest party *r-delivers* a message tagged with $ID$, then all honest parties *r-deliver* some message tagged with $ID$, provided all honest parties have been activated on $ID$ and the adversary delivers all associated messages.

### 5.4.2 A Protocol for Reliable Broadcast

```
Protocol RBC for party $P_i$ and tag $ID$
 Initialization:
        $\bar{m} \leftarrow \perp; \bar{d} \leftarrow \perp$
        $e_d \leftarrow 0; r_d \leftarrow 0$        $(d \in \{0,1\}^{k'})$
 case

        upon receiving message $(ID, \texttt{in}, \texttt{r-broadcast}, m)$:
            send $(\texttt{r-send}, m)$ to all parties
        upon receiving message $(\texttt{r-send}, m)$ from $P_l$:
            if $j = l$ and $\bar{m} = \perp$ then
              $\bar{m} \leftarrow m$
              send $(\texttt{r-echo}, H(m))$ to all parties
        upon receiving message $(\texttt{r-echo}, d)$ from $P_l$ for the first time:
            $e_d \leftarrow e_d + 1$
            if $e_d = n - t$ and $r_d \leq t$ then
              send $(\texttt{r-ready}, d)$ to all parties
        upon receiving message $(\texttt{r-ready}, d)$ from $P_l$ for the first time:
            $r_d \leftarrow r_d + 1$
            if $r_d = t + 1$ and $e_d < n - t$ then
              send $(\texttt{r-ready}, d)$ to all parties
            else if $r_d = 2t + 1$ then
              $\bar{d} \leftarrow d$
              if $H(\bar{m}) \neq d$ then
                send $(\texttt{r-request})$ to $P_1, \ldots, P_{2t+1}$
                wait for a message $(\texttt{r-answer}, m)$ such that $H(m) = \bar{d}$
                $\bar{m} \leftarrow m$
              output $(ID, \texttt{out}, \texttt{r-deliver}, \bar{m})$
        upon receiving message $(\texttt{r-request})$ from $P_l$ for the first time:
            if $\bar{m} \neq \perp$ then
              send $(\texttt{r-answer}, \bar{m})$ to $P_l$
 end case
```

Figure 5.2: RBC for authenticated reliable broadcast adopted from Bracha [Bra84].

A message-efficient reliable broadcast protocol, denoted RBC, is given in Figure 5.2; it results from a small modification of Bracha's reliable broadcast protocol [Bra84] to reduce the communication complexity.

Protocol RBC uses the hash of a payload message as a short, but unique representation for the potentially much longer message. The idea is that the payload is sent only once by the sender to all parties (similar to [Rei94]). When a party is ready to deliver a payload message but does not yet know it, it asks an arbitrary subset of $2t + 1$ parties for its contents and at least one of them will answer with the correct value.

In the description of the protocol, recall the global **wait for** condition for any message with a matching tag. Let $\perp$ denote a special value that cannot be broadcast. To implement the condition that a particular message from a party is processed only the first time it is received, one has to maintain the corresponding flags and counters, indexed by the contents of the message.

**Theorem 5.3.** *Assuming $H$ is a collision-free hash function, Protocol* RBC *provides authenticated reliable broadcast for $n > 3t$.*

*Proof. Deadlock Freeness* is clear for honest senders by inspection of the protocol because all parties receive the initial `r-send` message and also $2t+1$ `r-ready` messages from honest parties, provided all associated messages are delivered. It may not hold for faulty senders, though.

For *consistency*, suppose an honest party $P_i$ has *r-delivered* $m$ and another honest party $P_{i'}$ has *r-delivered* $m' \neq m$ with tag *ID*. Then $P_i$ must have received `r-ready` messages containing $d = H(m)$ from at least $t+1$ honest parties; the same holds for $P_{i'}$ with $d' = H(m')$. If $d = d'$, the adversary has created a collision in $H$. We assume no such collisions occur in the rest of the proof.

An honest party generates an `r-ready` message for $d$ only if it has received $n-t$ `r-echo` messages containing $d$ or $t+1$ `r-ready` messages already containing $d$. Thus, at least one honest party has sent an `r-ready` message containing $d$ upon receiving $n-t$ `r-echo` messages; at most $t$ of them are from corrupted parties. Similarly, some honest party must have received $n-t$ `r-echo` messages containing $d'$. Thus, there are at least $2(n-t) \geq n+t+1$ `r-echo` messages with tag *ID* and at least $n-t+1$ among them from honest parties. But no honest party generates more than one such message by the protocol.

To establish *totality*, note that if some honest $P_i$ delivers $\bar{m}$, then it has received the message $(\texttt{r-ready}, \bar{d})$ from $2t+1$ different parties. Therefore, at least $t+1$ honest parties have sent `r-ready` with *ID* and $\bar{d} = H(\bar{m})$, which will be received by all honest parties (assuming the adversary delivers all messages). Thus, all honest parties will send the corresponding `r-ready` message and any other party $P_l$ will receive $2t+1$ of them. If $P_l$ already knows $m'$ with $H(m') = \bar{d}$, it outputs that.

Otherwise, $P_l$ will send an `r-request` to $2t+1$ parties and wait for an `r-answer` satisfying $H(m') = \bar{d}$. Observe that there is at least one honest party who has sent an `r-ready` message containing $\bar{d}$ upon receiving $n-t$ corresponding `r-echo` messages. Thus, there are at least $n-2t$ honest parties who sent `r-echo` and know some $m'$ such that $H(m') = \bar{d}$. Sending the `r-request` to $2t+1$ parties ensures that at least one of them receives and answers it, provided all messages are delivered.

For *validity*, the uniqueness of the *r-delivered* message is clear from the protocol. If the sender $P_j$ of message with sequence number $s$ is honest, then at most $t$ parties will send `r-echo` messages for tag *ID* with $m' \neq m$. Thus, no uncorrupted party generates an `r-ready` message with $d$ different from $H(m)$ and no uncorrupted party outputs $m'$. Actually, the protocol also satisfies *authenticity* because honest parties process `r-send` messages only from the sender indicated by the `r-echo` message.

It is easy to see that the protocol satisfies *efficiency* for any sender. □

Note that collecting $n-t$ `r-echo` messages is needed for totality (because `r-request` messages are sent to only $2t+1$ parties), but for consistency alone, this could be relaxed to $\lceil \frac{n+t+1}{2} \rceil$ `r-echo` messages.

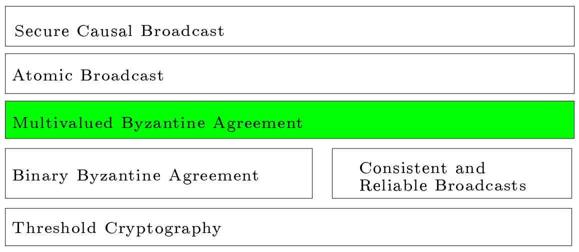
The message complexity of Protocol RBC is $O(n^2)$. If messages are delivered faithfully by a "benign" scheduler and no faults occur, then its communication complexity is only $O(n^2 k' + n|m|)$ for broadcasting a single message $m$, where $k'$ is the length of a hash value. However, the adversary can delay the `r-send` messages for some parties and increase the communication complexity. Since there are at most $t$ honest parties who issue an `r-request` by the argument above to establish totality, $m$ is transmitted $O(t^2)$ times and the overall communication complexity is $O(n^2 k' + n|m| + t^2|m|)$, or $O(n^2|m|)$ with maximal resilience.

Contrast this with the standard form of Bracha's broadcast that requires bit complexity in the order of $\Omega(n^2|m|)$, even in executions without faults. Under optimal circumstances, Protocol RBC needs to transmit $m$ only once per party in the system.

Protocol RBC could be made verifiable by adding a digital signature to the *r-ready* messages (this idea goes back to Pease, Shostak, and Lamport [PSL80]).

# Chapter 6

# Multivalued Byzantine Agreement


Secure Causal Broadcast
Atomic Broadcast
Multivalued Byzantine Agreement
Binary Byzantine Agreement
Consistent and Reliable Broadcasts
Threshold Cryptography

This chapter extends the Byzantine agreement protocol form Chapter 4 from binary agreement to agreement on arbitrary bit-strings. It has an expected message complexity that is quadratic in the number of parties and independent from the length of the decided values for all practical purposes. Contrary to earlier protocols, it never decides on a default value. This is achieved by using a new notion of "external" validity, which ensures that the final agreement value is useful for the context in which agreement must be reached.

## 6.1 Introduction

Randomized agreement protocols as described in Chapter 4 work only for *binary* decisions. In order to build distributed secure applications, however, this is not sufficient. One also needs agreement on values from large sets, which might even be determined dynamically by an application. This is the problem of *multi-valued* agreement. The "trivial" solution of agreeing bit-by-bit on the outcome clearly incurs an unreasonable overhead, while causing great difficulties in maintaining validity – it may satisfy weak definitions of validity, but it is quite difficult to agree on a meaningfull value like a digital signature.

In this chapter, we propose a *multi-valued Byzantine agreement* protocol with an *external* validity condition. External validity ensures that the final agreement value is acceptable to the particular application that requests agreement, and is key to the integration of the agreement primitive in higher-level protocols.

The usual notion of validity for Byzantine agreement requires that only if *all* honest parties propose the same value, this is also the agreement value. No particular outcome is guaranteed otherwise. Obviously, this still ensures that the agreement value was proposed by *some* honest party for the binary case. But it does not generalize to multi-valued Byzantine agreement, and indeed, previous protocols for multi-valued agreement [Rab83, TC84, Per85] may fall back to a default value in this case, and decide for a value that *no* honest party proposed.

External validity requires that the agreement value must be legal according to some verification, which guarantees its usefulness in the particular context. To ensure this, validated Byzantine agreement uses a global, polynomial-time computable predicate, known to all parties and determined by the particular higher-level application. Typically, a decision value will be validated through a digital signature.

Our multi-valued Byzantine agreement protocol invokes only a constant number of binary Byzantine agreement sub-protocols on average and achieves this by using a cryptographic common coin protocol in a novel way. It withstands the maximal possible corruption of up to one third of the parties and has expected quadratic message complexity (in the number of parties), which is essentially optimal.

**Related Work.** The problem of multi-valued agreement addressed in this chapter is related to *interactive consistency* [PSL80, Fis83], which requires agreement on a vector of $n$ values, one from each party. This notion is mostly used for environments with crash failures.

Validated Byzantine agreement is also related to the primitive of *agreement on a core set*, which was introduced by Ben-Or, Canetti, and Goldreich [BCG93] in the information-theoretic model with applications to secure asynchronous multi-party computation. The problem is to agree on a fixed-size common subset of all parties that satisfy a dynamic predicate; all honest parties will eventually satisfy the predicate. In contrast our work, the validity condition in this problem relies on the *local* views of all parties. Agreement on a core set can be turned into multi-valued agreement rather easily and vice versa. However, the most efficient implementation of agreement on a core set, by Ben-Or, Kelmer, and Rabin [BKR94], uses $n$ independent invocations of binary Byzantine agreement, and has much higher message and communication complexities than our protocol. Apparently, there is also an older, unpublished manuscript of Ben-Or and El-Yaniv [BOEY91] addressing both multi-valued agreement and interactive consistency.

## 6.2 Problem Statement

A validated Byzantine agreement is similar to the binary Byzantine agreement as described in section 4.2, but agreement can be on arbitary values. Furthermore, we need a mechanism to ensure that agreement values are sensible. To this end, we use a public predicate $Q_{ID}$ that takes as an input an agreement value $v$ and a proof $\pi$, and that is satisfied if $v_i$ is a valid agreement value, justified by $\pi$. For example, $v_i$ could be a document (that has to satisfy some syntactic rules). and $\pi$ could be a threshold signature confirming that at least $t+1$ parties have seen the document.

Twith differences on the activation and decision; after initializing the protocol instance $ID$, the adversary may deliver a message to $P_i$ of the form

$$(ID, \texttt{in}, \texttt{propose}, V_i, \pi_i),$$

where $V_i \in \{0,1\}^*$ is the initiall value of party $P_i$, and $\pi_i$ is the validator, i.e., if $P_i$ is an honest parties, then $Q_{ID}(V_i, \pi_i)$ holds.

Upon decision, a party outputs a message of the form

$$(ID, \texttt{out}, \texttt{decide}, \textit{decision-value}, wordvalidator),$$

where *decision-value*, *validator* $\in \{0,1\}^*$. We say that $P_i$ *decides decision-value* for $ID$ validated by *validator*.

**Definition 6.1 (Validated Byzantine Agreement).** A protocol solves the *validated Byzantine agreement* problem with predicate $Q_{ID}$ if it satisfies the following conditions except with negligible probability:

**External Validity:** Any honest party that terminates for $ID$ decides $v$ validated by $\pi$ such that $Q_{ID}(v, \pi)$ holds.

**Agreement:** If some honest party decides $v$ for $ID$, then any honest party that terminates decides $v$ for $ID$.

**Deadlock Freeness:** If all honest parties have been activated on $ID$ and all associated messages have been delivered, then all honest parties have decided for $ID$.

**Efficiency:** For every $ID$, the communication complexity for $ID$ is probabilistically polynomially bounded.

In other words, honest parties may propose all different values and the decision value may have been proposed by a corrupted party, as long as honest parties obtain the corresponding validation during the protocol. The standard definition of binary Byzantine agreement is obtained by adding one round of pre-processing and by requiring that the decision bit comes with the signatures of at least $t+1$ parties. Note that agreement, deadlock-freeness, and efficiency are the same as in the definition of ordinary, binary Byzantine agreement in the cryptographic model.

### 6.2.1 Biased Validity

Another variation of the validity condition is that an application may prefer one decision value over others. Such an agreement protocol may be *biased* and *always* output the preferred value in cases where other values would have been valid as well.

For our main protocol, we will need an implementation of a binary validated agreement protocol that is biased towards 1. Its purpose will be to detect whether there is a validation for 1, so it suffices to guarantee termination with output 1 if $t+1$ honest parties know the corresponding information at the outset. A *binary validated Byzantine agreement protocol biased towards 1* is a protocol for validated Byzantine agreement on values in $\{0,1\}$ such that the following condition holds:

**Biased External Validity:** If at least $t+1$ honest parties propose 1, then any honest party that terminates for $ID$ decides 1.

### 6.2.2 Implementing Binary Validated Byzantine Agreement

Binary asynchronous Byzantine agreement protocols can easily be adapted to external validity. For example, in the protocol in Chapter 4 one has to "justify" the pre-votes of round 1 with a valid $\pi$. The logic of the protocol guarantees that either a decision is reached immediately or the validations for 0 and for 1 are seen by all parties in the first two rounds.

Furthermore, the protocol can be biased towards 1 as shown in Section 4.4.3

## 6.3 The Protocol for Multi-Valued Agreement

We describe two related protocols for multi-valued validated Byzantine agreement: Protocol VBA, described in this section, needs $O(n)$ rounds and invokes $O(n)$ binary agreement sub-protocols; this can be improved to a constant expected number of rounds, resulting in Protocol VBAconst, which is described the next section.

The basic idea of the validated agreement protocol is that every party proposes its value as a candidate value for the final result. One party whose proposal satisfies the validation predicate is then selected in a sequence of binary Byzantine agreement protocols and this value becomes the final decision value. More precisely, the protocol consists of the following steps (see Figure 6.1).

**Echoing the proposal (lines 1–4):** Each party $P_i$ *c-broadcasts* the value that it proposes to all other parties using verifiable authenticated consistent broadcast. This ensures that all honest parties obtain the same proposal value for any particular party, even if the sender is corrupted. Then $P_i$ waits until it has received $n-t$ proposals satisfying $Q_{ID}$ before entering the agreement loop.

**Agreement loop (lines 5–20):** One party is chosen after another, according to a fixed permutation $\Pi$ of $\{1,\ldots,n\}$. Let $a$ denote the index of the party selected in the current round ($P_a$ is called the "candidate"). Each party $P_i$ carries out the following steps for $P_a$:

1. Send a `v-vote` message to all parties containing 1 if $P_i$ has received $P_a$'s proposal (including the proposal in the vote) and 0 otherwise (lines 6–11).

2. Wait for $n - t$ `v-vote` messages, but do not count votes indicating 1 unless a valid proposal from $P_a$ has been received—either directly or included in the `v-vote` message (lines 12–13).
3. Run a *binary* validated Byzantine agreement biased towards 1 to determine whether $P_a$ has properly broadcast a valid proposal. Vote 1 if $P_i$ has received a valid proposal from $P_a$ and validate this by the protocol message that completes the verifiable broadcast of $P_a$'s proposal. Otherwise, if $P_i$ has received $n - t$ `v-vote` messages containing 0, vote 0; no validation data is needed here. If the agreement decides 1, exit from the loop (lines 14–20).

**Delivering the chosen proposal (lines 21–24):** If $P_i$ has not yet *c-delivered* the broadcast by the selected candidate, obtain the proposal from the validation returned by the Byzantine agreement.

The full protocol is shown in Figure 6.1.

**Protocol VBA for party $P_i$, tag $ID$, and validation predicate $Q_{ID}$**
**Let** $V_{ID|a}(v, \rho)$ be the following predicate:

$$V_{ID|a}(v,\rho) \equiv (v = 0) \textbf{ or } \Big(v = 1 \textbf{ and } \rho \text{ completes the verifiable authenticated c-broadcast of a message } (\texttt{v-echo}, w_a, \pi_a) \text{ with tag } ID|a|0 \text{ such that } Q_{ID}(w_a, \pi_a) \text{ holds}\Big)$$

```
upon receiving message $(ID, \texttt{in}, \texttt{v-propose}, w, \pi)$:
 1: verifiably c-broadcast message $(\texttt{v-echo}, w, \pi)$ tagged with $ID|\texttt{vcbc}|i|0$
 2: $w_j \leftarrow \bot; \pi_j \leftarrow \bot \quad (1 \le j \le n)$
 3: wait for $n - t$ messages $(\texttt{v-echo}, w_j, \pi_j)$ to be c-delivered with tag $ID|\texttt{vcbc}|j|0$
      from distinct $P_j$ such that $Q_{ID}(w_j, \pi_j)$ holds
 4: $l \leftarrow 0$
 5: repeat
 6:    $l \leftarrow l + 1; a \leftarrow \Pi(l)$
 7:    if $w_a = \bot$ then
 8:       send the message $(\texttt{v-vote}, a, 0, \bot)$ to all parties
 9:    else
10:       let $\rho$ be the message that completes the c-broadcast with tag $ID|\texttt{vcbc}|a|0$
11:       send the message $(\texttt{v-vote}, a, 1, \rho)$ to all parties
12:    $u_j \leftarrow \bot; r_j \leftarrow \bot \quad (1 \le j \le n)$
13:    wait for $n - t$ messages $(\texttt{v-vote}, a, u_j, \rho_j)$ from distinct $P_j$ such
         that $V_{ID|a}(u_j, \rho_j)$ holds
14:    if there is some $u_j = 1$ then
15:       $v \leftarrow 1;\ \rho \leftarrow \rho_j$
16:    else
17:       $v \leftarrow 0;\ \rho \leftarrow \bot$
18:    propose $v$ validated by $\rho$ for $ID|a$ in binary validated Byzantine agreement
         biased towards 1, with predicate $V_{ID|a}$
19:    wait for the agreement protocol to decide some $b$ validated by $\sigma$ for $ID|a$
20: until $b = 1$
21: if $w_a = \bot$ then
22:    use $\sigma$ to complete the verifiable authenticated c-broadcast with tag $ID|\texttt{vcbc}|a|0$
         and c-deliver $(\texttt{v-echo}, w_a, \pi_a)$
23: output $(ID, \texttt{out}, \texttt{v-decide}, w_a, \pi_a)$
24: halt
```

Figure 6.1: Protocol VBA for multi-valued validated Byzantine agreement.

An obvious optimization of Protocol VBA is based on the observation that in most cases, adding $P_a$'s proposal in $\rho$ to a `v-vote` message is not necessary. If this is omitted, then the code for $P_i$ to receive `v-vote` messages has to be modified as follows. If a `v-vote` from $P_j$ indicates 1 but $P_i$ has not yet received $P_a$'s proposal, ignore the vote and ask $P_j$ to supply $P_a$'s proposal (by sending it the message $(ID|\texttt{vcbc}|a|0, \texttt{c-request})$). The `v-vote` by $P_j$ is only taken into account after $(\texttt{v-echo}, w_a, \pi_a)$ has been *c-delivered* with tag $ID|\texttt{vcbc}|a|0$

such that $Q_{ID}(w_a, \pi_a)$ holds; however, it may still be that enough votes indicating 0 from other parties are received before that.

**Lemma 6.1.** *In Protocol* VBA*, the adversary can cause at most* $2t$ *iterations of the agreement loop.*

*Proof.* The proof works by counting the total number $A$ of `v-vote` messages containing 0 that are generated by honest parties (over all iterations of the agreement loop).

Since every honest party has received a valid proposal from $n - t$ parties in the `v-echo` broadcasts, it will generate `v-vote` messages containing 0 for at most $t$ proposing parties. Thus, $A \leq t(n - t)$.

Note that for the binary Byzantine agreement protocol to decide 0 for a particular $a$ and to cause one more iteration of the loop, at least $n - 2t$ honest parties must propose 0 for the binary agreement (otherwise, there would be $t + 1$ or more honest parties proposing 1 and the binary agreement protocol would terminate with 1, as it is biased towards 1). Since honest parties only propose 0 if they have received $n - t$ `v-vote` messages containing 0, there must be at least $n - 2t$ honest parties who have generated a `v-vote` message containing 0 in this iteration.

Let $R$ denote the number of iterations of the loop where the binary agreement protocol decides 0. From the preceding argument, we have $A \geq R(n - 2t)$.

Combining these two bounds on $A$, we obtain $R(n - 2t) \leq (n - t)t$, or equivalently,

$$R \leq t + \frac{t^2}{n - 2t}.$$

Using $n - 2t \geq t + 1$, this can be simplified to $R \leq t + \frac{t^2}{t+1}$ and further to $R < 2t$. Thus, the binary agreement decides 1 at the latest in iteration $R + 1$ of the loop and the lemma follows. □

**Theorem 6.2.** *Given a protocol for biased binary validated Byzantine agreement and a protocol for verifiable authenticated consistent broadcast, Protocol* VBA *provides multi-valued validated Byzantine agreement for* $n > 3t$*.*

*Proof.* We have to establish *external validity*, *agreement*, *deadlock-freeness*, and *efficiency*.

*External validity* follows because every honest party that proposes 1 in the agreement on party $P_a$ has verified that $Q_{ID}$ holds for $w_a$ and $\pi_a$. Thus, by the standard validity condition for the binary Byzantine agreement, the decision is 0 if $Q_{ID}$ does not hold.

For *agreement*, note that the properties of the *binary* validated Byzantine agreement protocol ensure that all parties terminate the loop with the same $a$. By the consistency property of consistent broadcast, all honest parties obtain the same values $w_a$ and $\pi_a$ from the broadcast tagged with $ID|\texttt{vcbc}|a|0$. Thus, they output the same $w_a$.

*Deadlock Freeness* holds by inspection of the protocol.

*Efficiency* follows from the composition property of communication complexity statistic together with Lemma 6.1 because there are at most $2t$ binary agreement sub-protocols invoked for a particular $ID$. □

The message complexity of Protocol VBA is $O(tn^2)$ if Protocol VCBC is used for verifiable consistent broadcast and the binary validated Byzantine agreement is implemented according to Section 6.2.2.

If all parties propose $v$ and $\pi$ that are together no longer than $L$ bits, the communication complexity in the above case is $O(n^2(tK + L))$, assuming the length of a threshold signature and a signature share is at most $K$ bits. For a constant fraction of corrupted parties, however, both values are cubic in $n$. We can improve the message complexity to $O(n^2 \log n)$ by replacing the **for**–loop in the protocol by a parallel Byzantine agreement on all candidates as described in section 4.4.4. As the next section will show, a better solution is to use randomization again. This way, the expected message complexity can be reduced to a quadratic expression in $n$.

## 6.4 A Constant-round Protocol for Multi-Valued Agreement

In this section we present Protocol VBAconst, which is an improvement of the protocol in the previous section that guarantees termination within a constant expected number of rounds. The drawback of Protocol VBA above is that the adversary knows the order $\Pi$ in which the parties search for an acceptable candidate, i.e., one that has broadcast a valid proposal. Although at least one third of all parties are guaranteed to be accepted, as shown above, the adversary can choose the corruptions and schedule messages such that none of them is examined early in the agreement loop.

The remedy for this problem is to choose $\Pi$ randomly during the protocol *after* making sure that enough parties are already committed to their votes on the candidates. This is achieved in two steps. First, one round of commitment exchanges is added before the agreement loop. Each party must commit to the votes that it will cast by broadcasting the identities of the $n-t$ parties from which it has received valid `v-echo` messages (using authenticated consistent broadcast). Honest parties will later only accept `v-vote` messages that are consistent with the commitments made before. The second step is to determine the permutation $\Pi$ using a threshold coin-tossing scheme that outputs a random, unpredictable value after enough votes are committed. Taken together, these steps ensure that the fraction of parties which are guaranteed to be accepted are distributed randomly in $\Pi$, causing termination in a constant expected number of rounds.

The details of Protocol VBAconst are described in Figure 6.2 as modifications to Protocol VBA.

**Protocol VBAconst for party $P_i$, tag $ID$, and validation predicate $Q_{ID}$**

Modify Protocol VBA for party $P_i$, tag $ID$, and validation predicate $Q_{ID}$ as follows:

1. Initialize and distribute the shares for an $(n, t+1)$-threshold coin-tossing scheme $\mathcal{C}_1$ with $k''$-bit outputs during system setup. Recall that this defines a pseudorandom function $F$. Let $G$ be a pseudorandom generator.
2. Include the following instructions between lines 3 and 4 of Protocol VBA, before entering the agreement loop:

```
1: $c_j \leftarrow \begin{cases} 1 & \text{if } w_j \neq \perp \\ 0 & \text{otherwise} \end{cases} \qquad (1 \le j \le n)$
2: $C \leftarrow [c_1, \ldots, c_n]$
3: c-broadcast the message $(\texttt{v-commit}, C)$ tagged with $ID|\texttt{cbc}|i|0$
4: $C_j \leftarrow \perp \qquad (1 \le j \le n)$
5: wait for $n-t$ messages $(\texttt{v-commit}, C_j)$ to be c-delivered with tag $ID|\texttt{cbc}|j|0$
       such that at least $n-t$ entries in $C_j$ are 1
6: generate a coin share $\gamma$ of the coin $ID|\texttt{vba}$ and send the message $(\texttt{v-coin}, \gamma)$
       to all parties
7: wait for $t+1$ v-coin messages containing shares of the coin $ID|\texttt{vba}$ and
       combine these to get the value $S = F(ID|\texttt{vba}) \in \{0,1\}^{k''}$
8: choose a random permutation $\Pi$, using the pseudorandom generator $G$ with seed $S$.
```

3. Modify the condition for accepting `v-vote` messages (line 13) inside the agreement loop such that $(\texttt{v-vote}, a, 0, \perp)$ from $P_j$ is accepted only if $C_j$ is known and $C_j[a] = 0$. (This involves also waiting for additional messages $(\texttt{v-commit}, C_j)$ to be *c-delivered* as above.)

Figure 6.2: Protocol VBAconst for multi-valued validated Byzantine agreement.

To analyze the protocol, we consider the state of the system at the point in time when the first honest party $P_i$ reveals its coin share. The crucial observation is that $n-t$ "early committing" parties are committed to their 0-votes at this point because $P_i$ has delivered the corresponding broadcasts. We are now going to investigate the number of candidates that can be rejected by the adversary, by making the binary Byzantine agreement decide 0, and the number of iterations of the agreement loop.

**Lemma 6.3.** *Let $\mathcal{A} \subseteq \{1, \ldots, n\}$ denote the set of parties that garner* less *than $n - 2t$ commitments to 0-votes from the early committers, and suppose $\Pi$ is an ideal, random permutation of $\{1, \ldots, n\}$. Then, except with negligible probability,*

1. *for every $a \in \mathcal{A}$, the binary agreement protocol on $ID|a$ will decide 1;*

*2.* $|\mathcal{A}| > n - 2t$*;*

*3. there exists a constant* $\beta > 1$ *such that for all* $f \geq 1$*,*

$$\Pr\Big[\big(\Pi(1) \notin \mathcal{A}\big) \wedge \cdots \wedge \big(\Pi(f) \notin \mathcal{A}\big)\Big] \;\leq\; \beta^{-f}.$$

*Proof.* In order for the binary agreement on $ID|a$ to decide 0, there must be some honest party who proposes 0. By the instructions for computing $v$, it must have received $n-t$ `v-vote` messages containing 0 that are consistent with the commitments made by their issuers. But since there are only $n$ distinct parties, at least $n-2t$ of those 0-votes must come from early committers, which is not the case for any $a \in \mathcal{A}$. This proves the first claim.

To establish the second claim, let $A$ denote the total number of commitments to 0-votes cast by early committers. Since every early committer may commit to voting 0 for at most $t$ parties, we have $A \leq t(n-t)$. On the other hand, observe that $A \geq (n - |\mathcal{A}|)(n-2t)$ by the definition of $\mathcal{A}$.

Observe that these bounds on $A$ are the same as in Lemma 6.1 with $R = n - |\mathcal{A}|$. Using the same argument, it follows $|\mathcal{A}| > n - 2t$.

The third claim follows now because $|\mathcal{A}|$ is at least a constant fraction of $n$ and thus, there is a constant $\beta > 1$ such that $\Pr[\Pi(i) \notin \mathcal{A}] \leq 1/\beta$ for all $1 \leq i \leq f$. Since the probability of the $f$ first elements of $\Pi$ jointly satisfying the condition is no larger than for $f$ independently and uniformly chosen values, we obtain

$$\Pr\Big[\big(\Pi(1) \notin \mathcal{A}\big) \wedge \cdots \wedge \big(\Pi(f) \notin \mathcal{A}\big)\Big] \;\leq\; \beta^{-f}.$$

□

**Lemma 6.4.** *Assuming* $\mathcal{C}_1$ *is a secure threshold coin-tossing scheme and* $G$ *is a pseudorandom generator, there is a constant* $\beta > 1$ *such that for all* $f \geq 1$*, the probability of any honest party performing* $f$ *or more iterations of the agreement loop is at most* $\beta^{-f} + \epsilon$*, where* $\epsilon$ *is negligible.*

*Proof.* This can be shown by a standard hybrid argument, where one makes a series of small modifications to transform an idealized system into the real system, argues that each change affects the adversary only with negligible probability, and then concludes that the real system behaves just like the idealized system with all but negligible probability.

The "hybrid systems" are defined by running the system

(1) with a truly random permutation $\Pi$,

(2) with the output of $G$ replaced by truly random bits, and $\Pi$ computed from that,

(3) with $F(ID|\texttt{vba})$ replaced by a random bit string, but $G$ being a pseudorandom generator according to the protocol, and $\Pi$ computed from the output of $G$,

(4) with $F$, $G$, and $\Pi$ computed according to the protocol.

In all cases, we define a statistical test by letting the adversary run the system until the first honest party is about to release its share of the coin $ID|\texttt{vba}$, and then $F$, $G$, and $\Pi$ are determined. Note that the set of early committers is defined and the set $\mathcal{A}$ (of Lemma 6.3) can be computed at this point. The statistical test simply outputs 0 if $\Pi(i) \notin \mathcal{A}$ for all $1 \leq i \leq f$ and 1 otherwise.

We now analyze the behavior of the statistical test.

Case (1) above corresponds to the idealized system in Lemma 6.3, which implies that the test outputs 0 at most with probability $\beta^{-f}$.

In case (2) above, the permutation is generated from truly random bits with uniform distribution. This can be done using an algorithm that always terminates in a polynomial number of steps such that the output permutation is statistically close to a random permutation. The behavior of any polynomial-time adversary will not be changed by this, except with negligible probability.

Cases (2) and (3) above can be mapped to the definition of a pseudorandom generator. But if $G$ is secure, the statistical test will not be able to distinguish between them with more than negligible probability.

Finally, the difference between (3) and (4) corresponds to game C1–C4 in the definition of the coin $F$. Assuming $F$ is pseudorandom, this cannot induce more than a negligible difference in the behavior of the statistical test.

In conclusion, we obtain that no polynomial-time statistical test can distinguish between (1) and (4) and therefore the conclusions of Lemma 6.3 apply also to the real protocol except with negligible probability. Since honest parties go through more than $f$ iterations of the agreement loop only if the first $f$ elements of $\Pi$ are not in $\mathcal{A}$, this probability is at most $\beta^{-f}$ plus some negligible quantity. □

**Theorem 6.5.** *Given a protocol for biased binary validated Byzantine agreement and a protocol for verifiable consistent broadcast, Protocol* VBAconst *provides multi-valued validated Byzantine agreement for $n > 3t$ and invokes a constant expected number of binary Byzantine agreement sub-protocols.*

*Proof.* Since we have not changed the way in which binary agreement sub-protocols are invoked from Protocol VBA, we only have to show *deadlock-freeness* and *efficiency* for the modified protocol.

*Deadlock Freeness* holds because all $n - t$ honest parties broadcast correctly constructed commitments and therefore, enough valid `v-commit` and `v-vote` messages are guaranteed to be received in line 13 of the original protocol.

*Efficiency* follows from the composition property of probabilistically bounded communication complexity together with Lemma 6.4 above, because honest parties generate a probabilistically polynomially bounded number of messages of bounded size in each iteration of the agreement loop. □

The expected message complexity of Protocol VBAconst is $O(n^2)$ if Protocol VCBC is used for consistent verifiable broadcast and the binary validated Byzantine agreement is implemented according to Section 6.2.2. If all parties propose $v$ and $\pi$ that are together no longer than $L$ bits, the expected communication complexity in the above case is $O(n^3 + n^2(K + L))$, assuming a digital signature is $K$ bits. The $n^3$-term, which results from broadcasting the commitments, has actually a very small hidden constant because the commitments can be represented as bit vectors.

For a constant fraction of corrupted parties, the message complexity is quadratic in $n$ and essentially optimal. We do not know whether the communication complexity can be lowered to a quadratic expression in $n$ as well.

# Chapter 7

# Interlude: Relative Efficiency and Fairness

All protocols presented and analyzed so far had the property that protocol instances are independent from each other; to manage several parallel transactions, all we have to do is to keep them from interfering with each other.

For the problems addressed next, this is not possible anymore. In Atomic and Secure Causal Broadcast primitives, protocol instances must interfere by definition; the very purpose of these protocols is to get different instances (i.e., different broadcasted messages) organized.

This changes have some impact on the way we see protocols. It is not possible to define which protocol instance a certain implementation message belongs to; any assignment would be arbitrary. Furthermore, we loose an important flow control mechanism. It is relatively easy to combine independent (efficient) protocols in a way that the combined protocol maintains efficiency, and to prevent the adversary from starting an arbitrary number of protocol instances without ever terminating one. If the protocol instances can not be separated, the entire flow control has to be done manually.

## Relative Efficiency

If protocol instances interfere with each other, the efficiency has to be defined from a more global viewpoint. Instead of examining single protocol instances, we focus on the group of protocol instances that may interfere.

By the way we define broadcasts, this grouping comes quite naturally. All broadcasts are sent on a virtual broadcast channel, identified by a tag $ID_c$. For our protocols, the virtual channels are independent, i.e., interference is possible only for messages send on the same channel.

We say that a protocol message is *associated* to the channel with tag $ID_c$ if and only if the message is generated by an honest party and tagged with $ID_c$ or with a tag $ID_c|\ldots$ starting with $ID_c$. In particular, this encompasses all messages of the protocol with tag $ID_c$ generated by honest parties and all messages associated to basic broadcast and Byzantine agreement sub-protocols invoked.

Recall the definition of efficiency used before. A protocol is *efficient*, if the number of messages sent between honest parties (i.e., the communication complexity) for any protocol instance is polynomially bounded.

For the new notion of efficiency, the *relative efficiency*, we have to relate the communication complexity to the number of terminated protocol instances.

**Definition 7.1 (Relative Efficiency).** For a particular group of protocol instances with a tag with prefix $ID_c$, let $X$ denote its communication complexity and let $Y$ be the maximum number of distinct transactions that have been *terminated* by some honest party with a tag with prefix $ID_c$. Then the random variable $X/(Y+1)$ is probabilistically polynomially bounded.

Adding 1 to the divisor covers the state until the first protocol instance is terminated.

The efficiency condition counts only the transactions terminated by the "fastest" honest party. This party will usually be synchronized with at least $n-2t-1$ other honest parties, but it seems impossible to synchronize it with the "slowest" honest party. Moreover, there seems to be no easy way to provide a fixed bound on a suitable statistic (such as communication complexity) until *all* honest parties have delivered a particular payload. This is because the adversary can always drive the system forward with only $n-2t$ honest parties and leave the others behind. The "fast" parties might generate an a priori unbounded amount of work until the "slow" ones finally terminate a particular protocol instance, if at all.

## Fairness

Besides guaranteeing relative efficiency, on regarding the relation between several protocol instances one also would like to reach some sort of fairness, i.e., a protocol instance is executed "quickly" after a sufficient number of honest parties started participating in it; quickly in this sense means relative to other protocol instances that might be started later.

To define fairness, we first need a reference point, i.e., a definition when a protocol instance is considered active. Obviously, it does not make sense to include protocol instances into this definition that less than $t+1$ parties have started; as the adversary determines when an honest party first becomes aware of some transaction, and up to $t$ parties can be completely cut of, no useful guarantees can be given in this case.

Furthermore, we have to consider that a party can have a number of unfinished transactions going by the time a new transaction is started; for the definition of fairness, these transactions must be taken into account.

For this reason, we define "queues" of honest parties. We say that a transaction $T$ is *in the queue of a party $P_i$* if $P_i$ has send a start-message with tag $ID$ for $T$, but not send an end-message $m$ tagged with $ID$. The *system queue* contains any transaction that is in the queue of *some* honest party, but has not yet been terminated by *any* honest party.

**Fairness:** Fix a particular protocol instance with tag $ID$. For any transaction $T$ consider the system at the time when the $(t+1)$st honest party starts a transaction with tag $ID$. Let $S_T$ denote the number of distinct transactions terminated by honest parties up to that time, and let $V_T$ denote the total number of distinct unterminated transactions in the queues of those $t+1$ parties who have started a transaction $x$ (all with tag $ID$). Suppose the adversary causes some honest party to terminate $x$ as the $W_T$-th transaction. Then the random variable $(W_T - S_T)/V_T$ is polynomially bounded.

Note that $V_T \geq 1$ by definition. One could define a weaker version of fairness and start counting only if $k$ honest parties *broadcast* a request for $t+1 \leq k \leq n-t$. Also note that by using the term "time" iwe do not imply the existence of global time in our model; the time used here is purely defined by the adversary delivering messages.

refers to a global time, which is unknown to the participants and only used for definitions and the analysis.

## Protocol Composition

For the composition of protocols, one might have to be a bit careful. In a combined protocol, the definition of *terminating a transaction* depends exclusively on the highest level protocol. Even if all individual protocols satisfy relative efficiency, the combined protocol may not, for example if the adversary manages to invoke many instances of a low-level protocol without allowing progress for the corresponding instances of the higher level protocol.

For combined protocol, the definition for relative efficiency goes as follows:

**Definition 7.2 (Relative Efficiency for composed protocols).** For a particular group of protocol instance with a tag with prefix $ID_c$, let $X$ denote its communication complexity and let $Y$ be the maximum number of distinct transactions that have been *terminated* by some honest party *by the highest level protocol* that uses the transaction with tag $ID$. Then the random variable $X/(Y+1)$ is probabilistically polynomially bounded.

To ensure this, we need a mechanism to prevent the low level protocol to be invoked without synchronizing with the terminated transactions of the higher level protocol that defines termination of the whole protocol stack.

One mechanism is to not allow the low level protocol do put any work into a transaction unless specifically requested by the high level protocol; architecturally, one could see every instance of the low level protocol as a procedure call that has to be done by the high level protocol.

This approach works well for protocols that can not be invoked by a single party, like agreement protocols. It also works for the low-level broadcast protocols, where a party can just “expect” one broadcast from each other party, and serial numbers can be used to ensure that all parties participate in the same protocol instances.

For complex broadcast protocols like an atomic broadcast, however, this causes some problem.

A broadcast is started by a single party (the broadcaster), and the other parties do not necessarily expect a broadcast of this party; thus, one (potentially corrupted) party can start a complicated broadcast protocol.

One could add a pre-protocol that announces every broadcast to come; as this announcement has to be reliable, i.e., there must be an agreement on the expected broadcasts, this approach can be quite difficult.

Our approach goes a slight different ways; instead of telling a lower level protocol which broadcasts to expect, the higher level protocol only tells the lower level protocols how many instances it is allowed to process. When this number is exceeded, the lower level protocol refuses to perform work until the higher level protocol allows it to process new transactions.

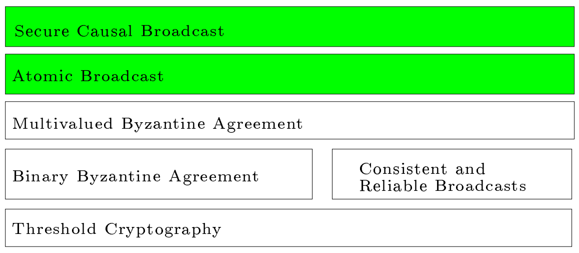
This is implemented by the higher level protocol sending an *acknowledgment* for every instance it is ready to process. The lower level protocol blocks after each broadcast it delivers, until it receives an acknowledgment. It is possible though that a higher level protocol sends a certain number of acknowledgments in advance. This allows the lower level protocol to run ahead for a well determined number of transactions and thus smoothes the flow of the composed protocol.

In practice, the *a-delivery* operation could be implemented by a blocking upcall to the higher-level protocol. In terms of the formal model, an acknowledgment is modeled as an input message $(ID, \texttt{in}, \texttt{acknowledge})$ from the adversary. We will say that the adversary *generates acknowledgments* if it acknowledges every *delivered* message.

The protocols presented in the next chapter will incorporate this approach. Note that they do not need acknowledgments to invoke the lower level Byzantine agreement protocols, as it is not possible to generate any work (according to our definition in Section 2.3) for honest parties on bogus agreement protocols.

# Chapter 8

# Atomic and Secure Causal Broadcast


Secure Causal Broadcast
Atomic Broadcast
Multivalued Byzantine Agreement
Binary Byzantine Agreement
Consistent and Reliable Broadcasts
Threshold Cryptography

In this chapter, we will use the protocols developed so far to build *atomic broadcast* protocols. We will present both a normal and an optimistic protocol. Finally, the technical part of this thesis will end with a secure causal broadcast protocol, which adds confidentiality and causal order to atomic broadcast.

## 8.1 Introduction

Atomic Broadcast is a fundamental building block in fault tolerant distributed computing. By ordering broadcast requests in such a way that two broadcast requests are received in the same order by all honest recipients, a synchronization mechanism is provided that deals with many of the most problematic aspects of asynchronous networks.

Atomic broadcast guarantees a total order on messages such that honest parties deliver all messages with a common tag in the same order. It is well known that protocols for atomic broadcast are considerably more expensive than those for reliable broadcast because even in the crash-fault model, atomic broadcast is equivalent to consensus [CT96] and cannot be solved by deterministic protocols. The atomic broadcast protocol given here builds directly on multi-valued validated Byzantine agreement from the last chapter.

Secure causal atomic broadcast (SC-ABC) is a useful protocol for building secure applications that use state machine replication in a Byzantine setting. It provides atomic broadcast, and also guarantees that the payload messages arrive in an order that maintains "input causality," a notion introduced by Reiter and Birman [RB94]. Informally, input causality ensures that a Byzantine adversary may not ask the system to deliver any payload message that depends in a meaningful way on a yet undelivered payload sent by an honest client. This is very useful for delivering client requests to a distributed service in applications that require the contents of a request to remain secret until the system processes it. Input causality is related to the standard causal order (going back to Lamport [Lam78]), which is a useful safety property for distributed systems with crash failures, but is actually not well defined in the Byzantine model [HT93].

**Related Work.** There is a rich literature on ordering broadcast channels, including several implementations and a broad theoretical basis. However, most work in the literature is done in the crash-failure model;

much less work has been done in the Byzantine failure model.

Rampart [Rei94] and SecureRing [KMMS98] directly transfer crash-failure protocols into the Byzantine world by using a modified failure detector along with digital signatures. The disadvantage of this approach is that it is relatively expensive, as a large number of expensive cryptographic operations need to be computed. Furthermore, there are attacks on the failure detector [ACBMT95] that can violate the safety of these protocols.

The BFS system by Castro and Liskov [CL99b] addresses these problems. As already mentioned, they only require timing assumptions to guarantee liveness, while the safety properties of the protocol hold regardless of timing issues. A similar approach is taken by Doudou *et al.* [DGG00], but their protocol is described and analyzed in terms of a Byzantine failure detector.

While both [CL99b] and [DGG00] still rely extensively on expensive public-key cryptographic operations, the extension of BFS in [CL99a, Cas00] relies much less on public-key cryptography.

Secure causal broadcast goes one step further and ensures that if a party has seen a broadcast message $m_1$ and subsequently sends another broadcast $m_2$, then $m_1$ will arrive before $m_2$ does.

This goes back to Lamport [Lam78], and has been well studied in the crash failure model; in the Byzantine model, however, the problem has not been well defined [HT93].

## 8.2 Atomic Broadcast

### 8.2.1 Problem Statement

Atomic broadcast ensures that all messages broadcast with the same tag *ID* are delivered in the same order by honest parties. The total order of atomic broadcast yields an implicit labeling of all messages. Assuming some honest party has atomically delivered $s$ distinct messages, the global sequence of the first $s$ delivered messages is well-defined. Thus, an explicit sequence number is not needed. Since the sender of a payload message is not necessarily identifiable (without requiring explicit authenticity instead of validity), the sender name is also omitted, and an unstructured tag *ID* suffices.

An atomic broadcast is activated when the adversary delivers an input message to $P_i$ of the form

$$(ID, \texttt{in}, \texttt{a-broadcast}, m),$$

where $m \in \{0,1\}^*$. When this occurs, we say $P_i$ *atomically broadcasts* $m$ *with tag* $ID$. "Activation" here refers only to the broadcast of a particular payload message; the broadcast channel *ID* must be opened before the first such request.

A party terminates an atomic broadcast of a particular payload by generating an output message of the form

$$(ID, \texttt{out}, \texttt{a-deliver}, m).$$

In this case, we say $P_i$ *atomically delivers* $m$ *with tag* $ID$. To distinguish atomic broadcast from other forms of broadcast, we will also use the terms *a-broadcast* and *a-deliver*.

For the composition of atomic broadcast with other protocols, we need a synchronized output mode, where *a-delivering* a payload may block the protocol and prevent it from delivering more payloads until the consumer is ready to accept them. We introduce an acknowledgment mechanism for output messages for this purpose, i.e., the adversary should *acknowledge* every *a-delivered* payload message to the delivering party. In practice, the *a-delivery* operation could be implemented by a blocking upcall to the higher-level protocol. In terms of the formal model, an acknowledgment is modeled as an input message $(ID, \texttt{in}, \texttt{a-acknowledge})$ from the adversary. When a party receives such a message, it means that its most recently *a-delivered* payload message with tag *ID* has been *acknowledged*. We will say that the adversary *generates acknowledgments* if it acknowledges every *a-delivered* message.

Again, the adversary must not request an *a-broadcast* of the same payload message from any particular party more than once for each *ID* (however, several parties may *a-broadcast* the same message).

Atomic broadcast protocols should be *fair* so that a payload message $m$ is scheduled and delivered within a reasonable (polynomial) number of steps after it is *a-broadcast* by an honest party. But since the adversary may delay the sender arbitrarily and *a-deliver* an a priori unbounded number of messages among the remaining honest parties, we can only provide such a guarantee when at least $t+1$ honest parties become "aware" of $m$. Our definition of fairness requires actually that only after $t+1$ honest parties have *a-broadcast* some payload, it is guaranteed to be delivered within a reasonable number of steps. It can be interpreted as a termination condition for the broadcast of a particular payload $m$. This is also the reason for allowing multiple parties to *a-broadcast* the same payload message—a client application might be able to satisfy this precondition through external means and achieve guaranteed fair delivery in this way.

**Definition 8.1 (Atomic Broadcast).** A protocol for *atomic broadcast* satisfies the following conditions except with negligible probability:

**Deadlock-Freeness:** If an honest party has *a-broadcast* $m$ tagged with $ID$, then it *a-delivers* $m$ tagged with $ID$, provided the adversary opens channel $ID$ for all honest parties, delivers all associated messages, and generates acknowledgments.

**Agreement:** If some honest party has *a-delivered* $m$ tagged with $ID$, then all honest parties *a-deliver* $m$ tagged with $ID$, provided the adversary opens channel $ID$ for all honest parties, delivers all associated messages, and generates acknowledgments for every party that has not yet *a-delivered* $m$ tagged with $ID$.

**Total Order:** Suppose an honest party $P_i$ has *a-delivered* $m_1, \ldots, m_s$ with tag $ID$, a distinct honest party $P_j$ has *a-delivered* $m'_1, \ldots, m'_{s'}$ with tag $ID$, and $s \leq s'$. Then $m_l = m'_l$ for $1 \leq l \leq s$.

**Validity:** For all $ID$, senders $j$, and sequence numbers $s$, every honest party *delivers* at most one message $m$ tagged with $ID|j|s$. Moreover, if all parties are honest, then $m$ was previously *broadcast* by $P_j$ with sequence number $s$.

**Fairness:** Fix a particular protocol instance with tag $ID$. For any $m$ consider the system at the time when the $(t+1)$st honest party *a-broadcasts* $m$ with tag $ID$. Let $S_m$ denote the number of distinct messages *a-delivered* by honest parties up to that time, and let $V_m$ denote the total number of distinct payload messages in the queues of those $t+1$ parties who have *a-broadcast* $m$ (all with tag $ID$). Suppose the adversary causes some honest party to *a-deliver* $m$ as the $W_m$-th message. Then the random variable $(W_m - S_m)/V_m$ is polynomially bounded.

**Relative Efficiency:** For a particular group of protocol instances with tag $ID$, let $X$ denote its communication complexity and let $Y$ be the maximum number of distinct payload messages that have been *a-delivered* by some honest party with tag $ID$. Then the random variable $X/(Y+1)$ is probabilistically polynomially bounded.

Some remarks on the above definition:

1. Deadlock-Freeness, agreement, and validity are analogous to reliable broadcast; only total order and fairness are new. Deadlock-Freeness ensures liveness of a protocol and rules out trivially empty protocols. It is stated in the canonical form (only the sender should *a-deliver* the message).

2. The agreement condition combines the consistency and totality of reliable broadcast; there is no need to distinguish these two aspects here, but they could also be separated. In particular, only totality requires that messages and acknowledgments are delivered.

### 8.2.2 A Protocol for Atomic Broadcast

We now present a protocol for atomic broadcast based on validated Byzantine agreement. Its overall structure is similar to the protocol of Hadzilacos and Toueg [HT93] for the crash-fault model, but we need to take additional measures to tolerate Byzantine faults.

Our Protocol ABC for atomic broadcast proceeds as follows. Each party maintains a FIFO queue of not yet *a-delivered* payload messages. Messages received to *a-broadcast* are appended to this queue whenever they are received. The protocol proceeds in asynchronous global rounds, where each round $r$ consists of the following steps:

1. Send the first payload message $w$ in the current queue to all parties, accompanied by a digital signature $\sigma$ in an `a-queue` message.

2. Collect the messages of $n - t$ distinct parties and store them in a vector $W$, store the corresponding signatures in a vector $S$, and propose $W$ for Byzantine agreement validated by $S$.

3. Perform multi-valued Byzantine agreement with validation of a vector $W = [w_1, \ldots, w_n]$ and proof $S = [\sigma_1, \ldots, \sigma_n]$ through the predicate $Q_{ID|\mathsf{abc}|r}(W, S)$ which is true if and only if for at least $n - t$ distinct indices $j$, the vector element $\sigma_j$ is a valid $\mathcal{S}$-signature on $(ID, \texttt{a-queue}, r, j, w_j)$ by $P_j$.

4. After deciding on a vector $V$ of messages, deliver the union of all payload messages in $V$ according to a deterministic order; proceed to the next round.

In order to ensure liveness of the protocol, there are actually two ways in which the parties move forward to the next round: when a party receives an *a-broadcast* input message (as stated above) and when a party receives an `a-queue` message of another party pertaining to the current round. If either of these two messages arrive and contain a yet undelivered payload message, and if the party has not yet sent its own `a-queue` message for the current round, then it enters the round by appending the payload to its queue and sending an `a-queue` message to all parties.

The detailed description of Protocol ABC is found below in Figure 8.1. The FIFO queue $q$ is an ordered list of values (initially empty). It is accessed using the operations *append*, *remove*, and *first*, where *append*$(q, m)$ inserts $m$ into $q$ at the end, *remove*$(q, m)$ removes $m$ from $q$ (if present), and *first*$(q)$ returns the first element in $q$. The operation $m \in q$ tests if an element $m$ is contained in $q$.

A party waiting at the beginning of a round simultaneously **waits for** `a-broadcast` and `a-queue` messages containing some $w \notin d$ in line 2. If it receives an *a-broadcast* request, the payload $m$ is appended to $q$. If only a suitable `a-queue` protocol message is received, the party makes $w$ its own message for the round, but does not append it to $q$. It should be clear from the protocol that no honest party is ever blocked waiting for some payload message to process if some honest party has *a-broadcast* one and all associated messages have been delivered.

The term $n - t$ in line 9 of the protocol and in the validation predicate $Q_{ID|\mathsf{abc}|r}$ could be replaced by any $f'$ between $t + 1$ and $n - t$ if the fairness condition is changed such that $f = n - f' + 1$ parties instead of $t + 1$ must have *a-broadcast* the message.

The protocol in Figure 8.1 is formulated using a single loop that runs forever after initialization; this is merely for syntactic convenience and can be implemented by decomposing the loop into the respective message handlers.

**Theorem 8.1.** *Given a protocol for multi-valued validated Byzantine agreement and assuming $\mathcal{S}$ is a secure signature scheme, Protocol* ABC *provides atomic broadcast for $n > 3t$.*

*Proof.* We first prove *deadlock-freeness.* Towards a contradiction, suppose that some honest party has *a-broadcast* a payload message $m$, but not *a-delivered* it and yet, all associated protocol messages and

```
Protocol ABC for party $P_i$ and tag $ID$
 Let $Q_{ID|\texttt{abc}|r}$ be the following predicate:

        $Q_{ID|\texttt{abc}|r}([w_1, \ldots, w_n], [\sigma_1, \ldots, \sigma_n]) \equiv \big($for at least $n - t$ distinct $j$, $\sigma_j$ is a valid
                                                    $\mathcal{S}$-signature by $P_j$ on $(ID, \texttt{a-queue}, r, j, w_j)$.$\big)$

 Initialization:
        $q \leftarrow []$           {FIFO queue of messages to a-broadcast}
        $d \leftarrow \emptyset$           {set of a-delivered messages}
        $r \leftarrow 0$           {current round}
 upon receiving message $(ID, \texttt{in}, \texttt{a-broadcast}, m)$:
        if $m \notin d$ and $m \notin q$ then
          $append(q, m)$
 loop forever
      1: $w_j \leftarrow \perp; \sigma_j \leftarrow \perp$      $(1 \le j \le n)$
      2: wait for $q \neq []$ or a message $(\texttt{a-queue}, r, l, w_l, \sigma_l)$ received from $P_l$
           such that $w_l \notin d$ and $\sigma_l$ is a valid signature from $P_l$
      3: if $q \neq []$ then
      4:    $w \leftarrow first(q)$
      5: else
      6:    $w \leftarrow w_l$
      7: compute a digital signature $\sigma$ on $(ID, \texttt{a-queue}, r, i, w)$
      8: send the message $(\texttt{a-queue}, r, i, w, \sigma)$ to all parties
      9: wait for $n - t$ messages $(\texttt{a-queue}, r, j, w_j, \sigma_j)$ such that $\sigma_j$ is a valid
           signature from $P_j$ (including the message from $P_l$ above)
     10: $W \leftarrow [w_1, \ldots, w_n]$; $S \leftarrow [\sigma_1, \ldots, \sigma_n]$
     11: propose $W$ validated by $S$ for multi-valued validated Byzantine agreement
           on $ID|\texttt{abc}|r$ with predicate $Q_{ID|\texttt{abc}|r}$
     12: wait for the validated Byzantine agreement protocol to decide some
           $V = [v_1, \ldots, v_n]$ for $ID|\texttt{abc}|r$
     13: $b \leftarrow \bigcup_{j=1}^{n} v_j$
     14: for $m \in (b \setminus d)$, in some deterministic order do
     15:    output $(ID, \texttt{out}, \texttt{a-deliver}, m)$
     16:    wait for an acknowledgment
     17:    $d \leftarrow d \cup \{m\}$
     18:    $remove(q, m)$
     19: $r \leftarrow r + 1$
 end loop
```

Figure 8.1: ABC for atomic broadcast using multi-valued validated Byzantine agreement.

acknowledgments have been delivered. Since the sender has *a-broadcast* but not *a-delivered* $m$, its queue contains at least $m$ and it can no longer be waiting in line 2. Thus, it has proceeded and sent `a-queue` messages to all parties in line 8. Since these have been delivered, every honest party has received an `a-queue` message containing $m \notin d$ and therefore has also entered the same round (by condition for waiting in line 2). Thus, all $n-t$ honest parties have sent valid `a-queue` messages and every honest party has received all of them and subsequently started and terminated Byzantine agreement. Since also the *a-delivered* payloads have been acknowledged, the sender must be waiting in line 2 with $q = []$. But then $m$ has been removed from $q$ and this occurs only if it was *a-delivered*, a contradiction.

We now establish *agreement*. Towards a contradiction, suppose that some honest $P_i$ has *a-delivered* a payload message $m$, but an honest $P_j$ has not *a-delivered* it and yet, all associated protocol messages have been delivered and acknowledgments have been generated for all parties who have not yet *a-delivered* $m$. Assume $P_i$ *a-delivered* $m$ in round $r$. Since no party who has not *a-delivered* $m$ is blocked waiting for messages or acknowledgments under these conditions, it is easy to see from inspection of the protocol and from the liveness condition of the Byzantine agreement sub-protocol that $P_j$ must have received all messages belonging to any round up to and including $r$. But then it cannot be waiting for an acknowledgment either—unless it has already *a-delivered* $m$.

The *total order* condition follows from the agreement property of the validated Byzantine agreement primitive since all honest parties decide on the same proposal and then *a-deliver* all payload messages contained in the proposal in a deterministic order. This implies also that the set $d$ of *a-delivered* messages is the same for all honest parties.

*Validity* is immediate from the protocol by induction on the construction of $d$, using the properties of Byzantine agreement. Even if corrupted parties include messages that have already been delivered, they are not delivered again.

To show *fairness*, consider the system when $W_m$ has been defined. We have to provide a polynomial bound on $(W_m - S_m)/V_m$, independent of the adversary. Note that the decided vector of payloads in every round contains $n-t$ values of which at least $n-2t \geq t+1$ are the first elements in the queues of honest parties. Thus, at least one of the initially $V_m$ elements in the respective $t+1$ queues has been *a-delivered* in every round henceforth. But there are at most $n$ distinct payloads that are *a-delivered* per round, which implies that $W_m - S_m$ is bounded by $nV_m$.

For *efficiency*, we have to relate the communication complexity of the protocol to the payload messages that were actually *a-delivered*. Note that honest parties generate messages only when they make progress in the round structure—either by sending an `a-queue` message or by invoking the Byzantine agreement sub-protocol. But an honest party enters the next round only if it is aware of some payload message that it has not yet *a-delivered*. Since at least one payload message from the system queue is delivered in every round, all protocol messages generated during that round can be related to that payload. There are a fixed polynomial number of protocol messages generated directly by the protocol in every round and the length of each one is at most $n$ times the length of a payload. The communication complexity of the Byzantine agreement sub-protocol is probabilistically polynomially bounded by its efficiency condition. Thus, the communication complexity per round is probabilistically polynomially bounded. □

The message complexity of Protocol ABC to broadcast one payload message $m$ is dominated by the number of messages in the multi-valued validated Byzantine agreement; the extra overhead for atomic broadcast is only $O(n^2)$ messages. The same holds for the communication complexity, but the proposed values have length $O(n(|m| + K))$, assuming digital signatures of length $K$ bits.

With Protocol VBAconst from Section 6.4, the total expected message complexity is $O(n^2)$ and the expected communication complexity is $O(n^3|m|)$ for an atomic broadcast of a single payload message.

### 8.2.3 Equivalence of Byzantine Agreement and Atomic Broadcast

For the sake of completeness, we state the equivalence of atomic broadcast to Byzantine agreement in the cryptographic model. It is the analogue to the equivalence between consensus and atomic broadcast in the crash-fault model shown by Chandra and Toueg [CT96].

**Corollary 8.2.** *(Binary) Byzantine agreement and atomic broadcast are equivalent in the basic system model Chapter 2, assuming a secure signature scheme and provided $n > 3t$.*

*Proof.* To implement Byzantine agreement from an atomic broadcast protocol, a party uses the following algorithm:

1. To propose $v \in \{0, 1\}$ for transaction $ID$, compute a digital signature $\sigma$ on $(ID, v)$ and *a-broadcast* the message $(ID, v, \sigma)$.

2. Wait for *a-delivery* of the first $2t+1$ messages of the form $(ID, v_j, \sigma_j)$ from distinct parties that contain valid signatures. Decide for the simple majority of all received values $v_j$.

The other direction follows from Theorems 6.2 and 8.1. □

Note that using an appropriately defined notion of *authenticated* atomic broadcast, this could also be implemented without the additional digital signatures in the reduction. However, Protocol ABC would have to be modified in order to provide authentication.

## 8.3 Optimistic Atomic Broadcast

We now present an optimistic atomic broadcast protocol [KS01]. The message and computational complexity of the optimistic part (and thus, the *amortized* message and computational complexity of the protocol) is essentially the same as that of a simple "Bracha broadcast."

Our protocol is inspired by the innovative work of Castro and Liskov [CL99b, CL99a, Cas00]. The optimistic phase is very "lightweight" — each request is processed using nothing more than a "Bracha broadcast" ([Bra84]) done by a leader. In particular, no public-key cryptography is used. The protocol is also quite stable; even if some number of parties, barring a designated leader, are corrupted, it remains in the optimistic part. If there are too many unexpected network delays, or the leader is corrupted, several parties may "time out," shifting the protocol into the pessimistic phase. The pessimistic phase builds on the multivalued Byzantine agreement protocol. It is still reasonably practical, although certainly not as efficient as the optimistic phase. The pessimistic phase cleans up any potential "mess" left by the current leader, after which the optimistic phase starts again with a new leader.

The optimistic phase of our protocol is essentially the same as that of Castro and Liskov. Therefore, we expect that in practice, our protocol is just as efficient as theirs. However, our pessimistic phase is quite different, and makes use of randomized Byzantine agreement as well as some additional public-key cryptographic operations. The pessimistic phase of Castro and Liskov makes use of public-key cryptography as well, and it is not clear if their pessimistic phase is significantly more or less efficient than ours — determining this would require some experimentation.

Castro and Liskov's pessimistic protocol is completely deterministic, and hence is subject to the FLP impossibility result. Indeed, although their protocol guarantees safety, it bases liveness on a timing assumption.

### 8.3.1 An Optimistic Protocol for Atomic Broadcast

The protocol operates in epochs, each epoch $e = 1, 2,$ etc., consisting of an optimistic and a pessimistic phase. In the optimistic phase, a designated leader is responsible to order incoming requests by assigning sequence numbers to them and initiating a reliable broadcast *a la* [Bra84]; the protocol only ensures that a dishonest leader cannot violate *total order.* If enough parties suspect that there is a threat to the deadlock-freeness or fairness of the protocol, they can shift the protocol into the pessimistic phase. The pessimistic phase cleans up any potential "mess" left by the current leader, after which the optimistic phase starts again with a new leader.

### 8.3.2 Overview and Optimistic Phase

In the optimistic phase of epoch $e$, when a party *a-broadcasts* a request $m$, it *initiates* the request by sending a message of the form $(ID, \texttt{initiate}, e, m)$ to the leader for epoch $e$. When the leader receives such a message, it *0-binds* a sequence number $s$ to $m$ by sending a message of the form $(ID, \texttt{0-bind}, e, m, s)$ to all parties. Sequence numbers start at zero in each epoch. Upon receiving a *0-binding* of $s$ to $m$, an honest party *1-binds* $s$ to $m$ by sending a message of the form $(ID, \texttt{1-bind}, e, m, s)$ to all parties. Upon receiving $n-t$ such *1-bindings* of $s$ to $m$, an honest party *2-binds* $s$ to $m$ by sending a message of the form $(ID, \texttt{2-bind}, e, m, s)$ to all parties. A party also *2-binds* $s$ to $m$ if it receives $t+1$ *2-bindings* of $s$ to $m$ — this has the effect of "amplifying" *2-bindings*, which is used to ensure *agreement.* Upon receiving $n-t$ such *2-bindings* of $s$ to $m$, an honest *a-delivers* $m$, provided all messages with lower sequence numbers were already delivered, enough acknowledgments have been received, and $m$ was not already *a-delivered.*

A party only sends or reacts to *0-*, *1-*, or *2-bindings* for sequence numbers $s$ in a "sliding window" $\{w, \ldots, w + \mathit{WinSize} - 1\}$, where $w$ is the number of requests already *a-delivered* in this epoch, and $\mathit{WinSize}$ is a parameter. Keeping the "action" bounded is necessary to ensure *efficiency* and *fairness.*

The number of requests that any party *initiates* but has not yet *a-delivered* is bounded by a parameter $\mathit{BufSize}$: a party will not *initiate* any more requests once this bound is reached. We denote by $\mathcal{I}$ the set of requests that have been *initiated* but not *a-delivered*, and we call this the *initiation queue.* If sufficient time passes without anything leaving the initiation queue, the party "times out" and *complains* to all other parties. These *complaints* are "amplified" analogously to the *2-bindings.* Upon receiving $n-t$ *complaints*, a party enters the pessimistic phase of the protocol. This strategy will ensure *deadlock-freeness.* Keeping the size of $\mathcal{I}$ bounded is necessary to ensure *efficiency* and *fairness.*

Also to ensure *fairness*, a party keeps track of the "age" of the requests in its initiation queue, and if it appears that the oldest request is being ignored, i.e., many other requests are being *a-delivered*, but not this one, then the party simply refuses to generate *1-bindings* until the problem clears up. If $t+1$ parties block in this way, they stop the remaining parties from making any progress in the optimistic phase, and thus, the pessimistic phase will be entered, where the fairness problem will ultimately be resolved.

We say that an honest party $P_i$ *commits* $s$ to $m$ in epoch $e$, if $m$ is the $s$th request (counting from 0) that it *a-delivered* in this epoch, optimistically or pessimistically.

Now the details. The state variables for party $P_i$ are as follows.

**Epoch number $e$.** The current epoch number, initially zero.

**Delivered $\mathcal{D}$.** All requests that have been *a-delivered* by $P_i$. It is required to ensure that requests are not *a-delivered* more than once; in practice, however, other mechanisms may be employed for this purpose. Initially, $\mathcal{D}$ is empty.

**Initiation Queue $\mathcal{I}$.** The queue of requests that $P_i$ *initiated* but not yet *a-delivered.* Its size is bounded by $\mathit{BufSize}$. Initially, $\mathcal{I}$ is empty.

**Window pointer $w$.** $w$ is the number of requests that have been *a-delivered* in this epoch. Initially, $w = 0$. The optimistic phase of the protocol only reacts to messages pertaining to requests whose sequence number lies in the "sliding window" $\{w, \ldots, w + \mathit{WinSize} - 1\}$. Here, $\mathit{WinSize}$ is a fixed system parameter.

**Echo index sets $\mathit{BIND}_1$ and $\mathit{BIND}_2$.** The sets sequence numbers which $P_i$ has *1-bound* or *2-bound*, respectively. Initially empty.

**Acknowledgment count $\mathit{acnt}$.** Counts the number of acknowledgments received for *a-delivered* requests. Initially zero.

**Complaint flag $\mathit{complained}$.** Set if $P_i$ has issued a complaint. Initially *false*.

**Initiation time $\mathit{it}(m)$.** For each $m \in \mathcal{I}$, $\mathit{it}(m)$ is equal to the value of $w$ at the point in time when $m$ was added to $\mathcal{I}$. Reset to zero across epoch boundaries. These variables are used in combination with a fixed parameter $\mathit{Thresh}$ to ensure *fairness.*

**Leader index $l$.** The index of the leader in the current epoch; we simply set $l = (e \bmod n) + 1$.

**Scheduled request set $\mathcal{SR}$.** Only maintained by the current leader. It contains the set of messages which have been assigned sequence numbers in this epoch. Initially, it is empty.

**Next available sequence number $scnt$.** Only maintained by the leader. Value of the next available sequence number. Initially, it is zero.

The protocol for party $P_i$ consists of two threads. The first is a trivial thread that simply counts acknowledgments and is shown in figure 8.2

```
/* Process Acknowledgments */

loop forever
        wait for an acknowledgment: increment $acnt$.
end loop
```

Figure 8.2: Acknowledgment processing of the protocol Opt-AB for optimistic atomic broadcast.

The main thread is described below in Figure 8.3.

### 8.3.3 Fully Asynchronous Recovery

The recovery protocol tidies up all requests that where initiated under a (potentially) faulty leader. We distinguish between three types of requests:

– Requests for which it can be guaranteed that they have been *a-delivered* by an honest party.

– Requests that potentially got *a-delivered* by an honest party.

– Requests for which it can be guaranteed that they have not been *a-delivered* by an honest party.

For the first two kinds of requests, an order of delivery might already be defined, and has to be preserved. The other requests have not been *a-delivered* at all, so the recovery protocol has complete freedom on how to order them. They can not be left to the next leader, however, as an adversary can always force this leader to be thrown out as well. To guarantee efficiency, the recovery procedure has to ensure that *some* request is *a-delivered* in every epoch. This is precisely the property that Castro and Liskov's protocol fails to achieve: in their protocol, without imposing additional timing assumptions, the adversary can cause the honest parties to generate an arbitrary amount of messages before a single request is *a-delivered.*

According to the three types of requests, the recovery protocol consists of three parts.

*Part 1.* In the first part, a watermark $\hat{s}_e$ is jointly computed such that all requests with sequence numbers up to this $\hat{s}_e$ have been *a-delivered* by *some* honest party. The watermark has the property that at least one honest party *a-delivered* request $\hat{s}_e$, and no honest party *a-delivered* a request higher than $\hat{s}_e + 2 \cdot \mathit{WinSize}$.

The watermark is determined as follows. When $P_i$ enters the pessimistic phase of the protocol, it sends out a signed statement to all parties that indicates its highest *2-bound* sequence number. Then, $P_i$ waits for $t + 1$ signatures on sequence numbers $s'$ such that $s'$ is greater than or equal to the highest sequence number $s$ that $P_i$ committed during the optimistic phase. Let us call such a set of signatures a *a strong consistent set of signatures for $s$*. Since $P_i$ already received $n - t$ *2-bindings* for $s$, it is assured that at least $t+1$ of these came from honest parties, and so it will eventually receive a strong consistent set of signatures for $s$. Any party that is presented with such a set of signatures can conclude the following: one of these signatures is from an honest party, therefore some honest party sent a *2-binding* for a sequence number at least $s$, and therefore, because of the logic of the sliding window, that honest party committed $(s - \mathit{WinSize})$ in its optimistic phase.

Once $P_i$ has obtained its own strong consistent set for $s$, it signs it and sends this signed strong consistent set to all parties, and collects a set $\mathcal{M}_i$ of $n - t$ signed strong consistent sets from other parties. Then $P_i$

```
**Protocol** OPT-ABC **for party** $P_i$ **and tag** $ID$
  **loop forever** ; **case**
      /* Initiate $m$. */

      **upon receiving** a message $(ID, \texttt{in}, \texttt{a-broadcast}, m)$ for some $m$ such that $m \notin \mathcal{I} \cup \mathcal{D}$ and $|\mathcal{I}| < BufSize$:
          Send the message $(ID, \texttt{initiate}, e, m)$ to the leader.
          Add $m$ to $\mathcal{I}$.
          Set $it(m) \leftarrow w$.
      /* 0-bind $scnt$ to $m$. */

      **upon receiving** a message $(ID, \texttt{initiate}, e, m)$ for some $m$, such that $i = l$ and $w \leq scnt < w + WinSize$
        and $m \notin \mathcal{D} \cup \mathcal{SR}$:
          Send the message $(ID, \texttt{0-bind}, e, m, scnt)$ to all parties.
          Increment $scnt$ and add $m$ to $\mathcal{SR}$.
      /* 1-bind $s$ to $m$. */

      **upon receiving** a message $(ID, \texttt{0-bind}, e, m, s)$ from the current leader for some $m, s$ such that $w \leq s <$
        $w + WinSize$ and $s \notin BIND_1$ and $((\mathcal{I} = \emptyset)$ or $(w \leq \min\{it(m) : m \in \mathcal{I}\} + Thresh))$:
          Send the message $(ID, \texttt{1-bind}, e, m, s)$ to all parties.
          Add $s$ to $BIND_1$.
      /* 2-bind $s$ to $m$. */

      **upon receiving** $n - t$ messages of the form $(ID, \texttt{1-bind}, e, m, s)$ from distinct parties that agree on $s$ and
        $m$, such that $w \leq s < w + WinSize$ and $s \notin BIND_2$:
          Send the message $(ID, \texttt{2-bind}, e, m, s)$ to all parties.
          Add $s$ to $BIND_2$.
      /* Amplify a 2-binding of $s$ to $m$. */

      **upon detecting** $t + 1$ messages of the form $(ID, \texttt{2-bind}, e, m, s)$ from distinct parties that agree on $s$ and
        $m$, such that $w \leq s < w + WinSize$ and $s \notin BIND_2$:
          Send the message $(ID, \texttt{2-bind}, e, m, s)$ to all parties.
          Add $s$ to $BIND_2$.
      /* Commit $s$ to $m$. */

      **upon receiving** $n - t$ messages of the form $(ID, \texttt{2-bind}, e, m, s)$ from distinct parties that agree on $s$ and
        $m$, such that $s = w$ and $acnt \geq |\mathcal{D}|$ and $m \notin \mathcal{D}$ and $s \in BIND_2$:
          Output $(ID, \texttt{out}, \texttt{a-deliver}, m)$.
          Increment $w$.
          Add $m$ to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).
          **stop timer**.
      /* Start timer. */

      **upon** (timer not running) and (not $complained$) and $(\mathcal{I} \neq \emptyset)$ and $(acnt \geq |\mathcal{D}|)$:
          **start timer**.
      /* Complain. */

      **upon timeout**:
          if not $complained$ then:
              Send the message $(ID, \texttt{complain}, e)$ to all parties.
              Set $complained \leftarrow true$.
      /* Amplify complaint. */

      **upon detecting** $t + 1$ messages $(ID, \texttt{complain}, e)$ from distinct parties, such that not $complained$:
          Send the message $(ID, \texttt{complain}, e)$ to all parties.
          Set $complained \leftarrow true$.
          **stop timer**.
      /* Go pessimistic. */

      **upon receiving** $n - t$ messages $(ID, \texttt{complain}, e)$ from distinct parties, such that $complained$:
          Execute the procedure $Recover$ below.
  **end case** ; **end loop**
```

Figure 8.3: Main thread of the protocol Opt-AB for optimistic atomic broadcast.

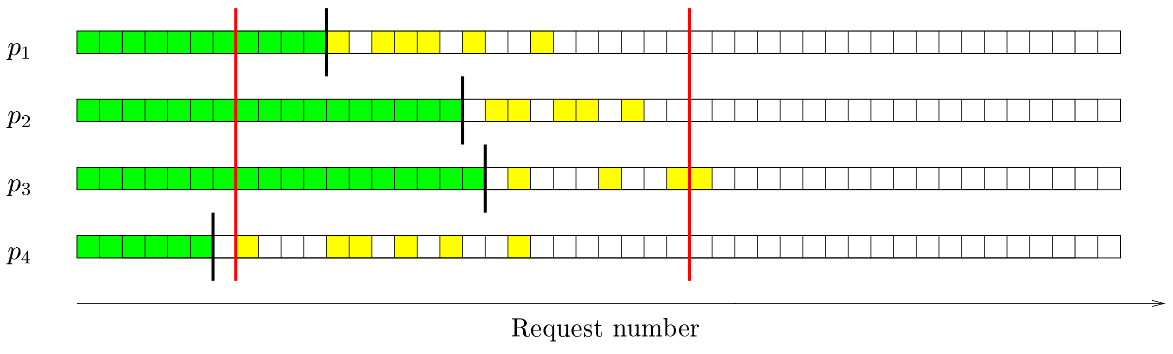
runs a multivalued Byzantine agreement protocol with input $\mathcal{M}_i$, obtaining a common set $\mathcal{M}$ of $n-t$ signed strong consistent sets. The watermark computed as $\hat{s}_e = (\tilde{s} - \mathit{WinSize})$, where $\tilde{s}$ is the maximum sequence number $\tilde{s}$ for which $\mathcal{M}$ contains a strong consistent set for $\tilde{s}$. We will show that no honest party commits a sequence number higher than $\hat{s}_e + (2 \cdot \mathit{WinSize})$ in its optimistic phase. And as already argued above, at least one honest party commits $\hat{s}_e$ in its optimistic phase.

After computing the watermark, all parties "catch up" to the watermark, i.e., commit all sequence numbers up to $\hat{s}_e$, by simply waiting for $t+1$ consistent *2-bindings* for each sequence number up to the watermark. By the logic of the protocol, since one of these *2-bindings* must come from an honest party, the correct request is *a-delivered.* Since one honest party has already committed $s$ in its optimistic phase, at least $t+1$ honest parties have already sent corresponding *2-bindings*, and these will eventually arrive.

*Part 2.* In the second part, we deal with the requests that might or might not have been *a-delivered* by some honest party in the optimistic phase of this epoch. We have to ensure that if some honest party has *a-delivered* a request, then all honest parties do so. The sequence numbers of requests with this property lie in the interval $\hat{s}_e + 1 \ldots \hat{s}_e + 2 \cdot \mathit{WinSize}$. Each party makes a proposal that indicates what action should be taken for all sequence numbers in this critical interval. Again, multivalued Byzantine agreement is used to determine which of possibly several valid proposals should be accepted.

To construct such a proposal for sequence number $s$, each party $P_i$ does the following. Party $P_i$ sends out a signed statement indicating if it sent a *2-binding* for that $s$, and if so, the corresponding request $m$. Then $P_i$ waits for a set of $n-t$ "consistent" signatures for $s$, such that the set does not contain conflicting requests. By the logic of the protocol, an honest party will eventually obtain such a consistent set, which we call a *weak consistent set of signatures for s*. If all signatures in this set are on statements that indicate no *2-binding*, then we say the set *defines no request*; otherwise, we say it *defines request* $m$, where $m$ is the unique request appearing among the signed statements in the set. $P_i$'s proposal consists of a set of weak consistent set of signatures for $s$. Any party that is presented with such a set can conclude the following: if the set defines no request, then no party optimistically commits $s$; if the set defines $m$, then *if* any honest party optimistically commits $s$ to some $m'$, then $m = m'$. Note that if the set defines some request $m$, this does not imply that $s$ was committed optimistically, and indeed, if $s$ was not optimistically committed, then the adversary can construct sets that define different requests.

*Part 3.* In the third part, we use a multivalued Byzantine agreement protocol to agree on a set of additional requests that should be *a-delivered* this epoch. This set will include the (possibly empty) initiation queues of at least $n-2t$ distinct honest parties. This property will be used to ensure fairness. Also, this set is guarantee to be non-empty if no requests were previously *a-delivered* (optimistically or otherwise) in this epoch. This property will be used to ensure efficiency.



🟩 Request Delivered

🟨 Requests "seen"

Figure 8.4: Three kinds of requests, depending on their state at the end of the optimistic phase

**The recovery procedure**

We begin with some terminology.

For any party $P_i$, and any message $\alpha$, we denote by $\{\alpha\}_i$ a signed form of the message, i.e., $\alpha$ concatenated with a valid signature under $P_i$'s public key on $\alpha$.

For any $s \geq -1$, a *strong consistent set* $\Sigma$ *for* $s$ is a set of $t+1$ correctly signed messages from distinct parties, each of the form $\{(ID, \texttt{s-2-bind}, e, s')\}_j$ for some $j$ and $s' \geq s$

A *valid watermark proposal* $\mathcal{M}$ is a set of $n-t$ correctly signed messaged from distinct parties, each of the form $\{(ID, \texttt{watermark}, e, \Sigma_j, s_j)\}_j$ for some $j$, where $\Sigma_j$ is a strong consistent set of signatures for $s_j$. The maximum value $s_j$ appearing in these watermark messages is called the *maximum sequence number* of $\mathcal{M}$.

For any $s \geq 0$, a *weak consistent set* $\Sigma'$ *for* $s$ is a set of $n-t$ correctly signed messages from distinct parties — each of the form $\{(ID, \texttt{w-2-bind}, e, s, m_j)\}_j$ for some $j$ — such that either all $m_j = \perp$, or there exists a request $m$ and all $m_j$ are either $m$ or $\perp$. In the former case, we say say $\Sigma'$ *defines* $\perp$, and in the latter case, we say $\Sigma'$ *defines* $m$.

For a set $\mathcal{Q}$ of requests and an integer $k \geq 0$, a $(\mathcal{Q}, k)$*-valid recover proposal* $\mathcal{P}$ is a set of $n-t$ correctly signed messages from distinct parties each of the form $\{(ID, \texttt{recover-request}, e, \mathcal{Q}_j)\}_j$ for some $j$, where $\mathcal{Q}_j$ is a set of at most *BufSize* requests with $\mathcal{Q}_j \cap \mathcal{Q} = \emptyset$; moreover, if $k = 0$, we require that some $\mathcal{Q}_j$ is non-empty. We define the *request set* for $\mathcal{P}$ as the set of all requests that appear in any of the sets $\mathcal{Q}_j$.

## 8.3.4 Analysis

If honest party $P_i$ enters epoch $e$, let $\mathcal{D}_e^{(i)}$ denote the sequence of requests that honest party $P_i$ *a-delivered* at the point in time where it entered this epoch. We say *consensus holds on entry to epoch e* if for any two honest parties $P_i$ and $P_j$ that enter epoch $e$, $\mathcal{D}_e^{(i)} = \mathcal{D}_e^{(j)}$. If consensus holds on entry to epoch $e$, and any honest party does enter epoch $e$, we denote by $\mathcal{D}_e$ the common value of the $\mathcal{D}_e^{(i)}$, and we denote by $N_e$ the length of $\mathcal{D}_e$.

Recall that we say that an honest party $P_i$ *commits* $s$ to $m$ in epoch $e$, if $m$ is the $s$th request (counting from 0) that it *a-delivered* in this epoch, optimistically or pessimistically. If this occurs in the optimistic phase, we say $P_i$ *optimistically commits* $s$ to $m$.

**Lemma 8.3.** *In any epoch, if two honest parties* 2-bind *a sequence number* $s$*, then they* 2-bind $s$ *to the same request.*

*Moreover, if for some* $s, m, m'$*, one honest party receives a set of* $t+1$ 2-bindings *of* $s$ *to* $m$ *and one honest party (possible the same one) receives a set of* $t+1$ 2-bindings *of* $s$ *to* $m'$*, then* $m = m'$*.*

*Proof.* This is a fairly standard argument. If some honest party *2-binds* $s$ to $m$, then some honest party (not necessarily the same one) has receives $n-t$ *1-bindings* of $s$ to $m$. But since any two sets of $n-t$ parties must contain a common honest party, and no party *1-binds* a sequence number more than once, if one honest party receives $n-t$ *1-bindings* of $s$ to $m$, and another receives $n-t$ *1-bindings* of $s$ to $m'$, then $m = m'$. That proves the first statement.

The second statement follows from the first, and the fact that any set of $t+1$ parties must contain an honest party. □

**Lemma 8.4.** *If all all honest parties have entered epoch* $e$*, and all messages and timeouts have been delivered, and one honest party enters the pessimistic phase of the protocol in this epoch, then all honest parties have gone pessimistic in epoch* $e$*.*

*Proof.* An honest party enters the pessimistic phase of an epoch if it receives $n-t$ complaint messages. This implies that at least $t+1$ honest parties have sent a complaint message, thus every honest party will eventually receive at least $t+1$ complaint messages. This will cause all honest parties to send out complaint messages, thus all honest parties eventually receive at least $n-t$ complaints and thus will go pessimistic. □

```
**Procedure** recover **for party** $P_i$ **and tag** $ID$
  /* Part 1: Recover Potentially delivered Requests */

  Send a the signed message $\{(ID, \texttt{s-2-bind}, e, \max(BIND_2 \cup \{-1\}))\}_i$ to all parties.
  wait for a strong consistent set $\Sigma_i$ for $w-1$.
  Send the signed message $\{(ID, \texttt{watermark}, e, \Sigma_i, w-1)\}_i$ to all parties.
  wait for a valid watermark proposal $\mathcal{M}_i$.
  Propose $\mathcal{M}_i$ for multivalued Byzantine agreement on a valid watermark proposal $\mathcal{M}$.
  Set $\hat{s}_e \leftarrow \tilde{s} - WinSize$, where $\tilde{s}$ is the maximum sequence number of $\mathcal{M}$.
  while $w \leq \hat{s}_e$ do:
        wait for $t+1$ messages of the form $(ID, \texttt{2-bind}, e, m, w)$ from distinct parties that agree on $m$, such that
            $acnt \geq |\mathcal{D}|$.
        Output $(ID, \texttt{out}, \texttt{a-deliver}, m)$.
        Increment $w$.
        Add $m$ to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).
  /* Part 2: Recover potentially delivered Requests */

  For $s \leftarrow \hat{s}_e + 1$ to $\hat{s}_e + (2 \cdot WinSize)$ do:
        If $P_i$ sent the message $(ID, \texttt{2-bind}, e, m)$ for some $m$, set $\tilde{m} \leftarrow m$; otherwise, set $\tilde{m} \leftarrow \perp$.
        Send the signed message $(ID, \texttt{w-2-bind}, e, s, \tilde{m})$ to all parties.
        wait for a weak consistent set $\Sigma'_i$ for $s$.
        Propose $\Sigma'_i$ for multivalued Byzantine agreement on a weak consistent set $\Sigma'$ for $s$.
        Let $\Sigma'$ define $m$.
        If $(s \geq w$ and $m \in \mathcal{D})$ or $m = \perp$, exit the for loop and go to Part 3.
        If $m \notin \mathcal{D}$ then:
              wait for $acnt \geq |\mathcal{D}|$.
              Output $(ID, \texttt{out}, \texttt{a-deliver}, m)$.
              Increment $w$.
              Add $m$ to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).
  /* Part 3: Recover undelivered Requests */

  Send the signed message $\{(ID, \texttt{recover-request}, e, \mathcal{I})\}_i$ to all parties.
  wait for a valid $(\mathcal{D}, w)$-recover proposal $\mathcal{P}_i$.
  Propose $\mathcal{P}_i$ for multivalued Byzantine agreement on a valid $(\mathcal{D}, w)$-recover proposal $\mathcal{P}$.
  Sequence through the request set of $\mathcal{P}$ in some deterministic order, and for each such request $m$, do the following:
        wait for $acnt \geq |\mathcal{D}|$.
        Output $(ID, \texttt{out}, \texttt{a-deliver}, m)$.
        Increment $w$.
        Add $m$ to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).
  /* Start New Epoch */

  Set $e \leftarrow e + 1$.
  Set $l \leftarrow (e \bmod n) + 1$.
  Set $\mathcal{SR} \leftarrow BIND_1 \leftarrow BIND_2 \leftarrow \emptyset$.
  Set $complained \leftarrow false$.
  Set $w \leftarrow scnt \leftarrow 0$.
  For each $m \in \mathcal{I}$:
        Send the message $(ID, \texttt{initiate}, e, m)$ to the leader.
        Set $it(m) \leftarrow 0$.
```

Figure 8.5: Recovery procedure of the protocol Opt-AB for optimistic atomic broadcast.

**Lemma 8.5.** *Suppose that consensus holds on entry to some epoch $e$, that some honest party has entered this epoch, and that no honest party has gone pessimistic in this epoch. The the following conditions hold.*

**local consistency:** *If some honest party commits $s$ to $m$, any honest party that also commits $s$, also commits $s$ to $m$.*

**local completeness:** *If some honest party commits $s$ to $m$, and all messages and timeouts have been delivered, and all honest parties have entered epoch $e$ and received $N_e + s$ acknowledgments, then all honest parties have committed $s$.*

**local deadlock-freeness:** *If all messages, timeouts, and acknowledgments have been delivered, and all honest parties have entered epoch $e$, then at most $t$ honest parties have non-empty initiation queues.*

**local unique delivery:** *Any honest party* a-delivers *each request at most once in this epoch.*

*Proof.* If some honest party commits $s$ to $m$, then it has received $n - t$ *2-bindings* of $s$ to $m$. At least $t + 1$ of these are from honest parties. Moreover, by Lemma 8.3, any set of $t + 1$ consistent *2-bindings* for $s$ that an honest party receives are *2-bindings* to $s$.

*Local consistency* is now immediate.

If *local completeness* does not hold, let us choose $s$ to be the minimal $s$ for which this it does not hold.

Consider any honest party $P_i$. We want to show that in fact, $P_i$ has committed $s$, yielding a contradiction.

By the minimality of $s$, it is easy to verify that the local value of $w$ for any honest party $P_j$ is at least $s$. Since $t + 1$ honest parties have *2-bound* $s$ to $m$, these *2-bindings* will be received at a point in time where $s$ lies in $P_j$'s window. So if $P_j$ will itself *2-bind* $s$ to $m$. Therefore all honest parties have *2-bound* $s$ to $m$, and $P_i$ has received these *2-bindings* while $s$ was in its sliding window. Because consensus holds on entry to epoch $e$, and by the *consistency* part of this lemma, and by the minimality of $s$, it follows that all honest parties' $\mathcal{D}$ sets are equal at the point in time when $w = s$ (locally), and in particular $m \notin \mathcal{D}$ at this point in time, and so is not "filtered out" as a duplicate. Also, $P_i$ has received sufficient acknowledgments, and so commits $s$ to $m$.

Suppose *local deadlock-freeness* does not hold. Then the $t + 1$ honest parties would certainly have sent complaint messages, and it is easy to verify that this would eventually cause all parties to complain, and hence go pessimistic. This contradicts our assumption that no party has gone pessimistic.

*Unique delivery* is clear from inspection, as duplicates are explicitly "filtered" in the optimistic phase. □

**Lemma 8.6.** *If all honest parties have entered the pessimistic phase of epoch $e$, and all messages and timeouts have been delivered, then all honest parties have agreed on a watermark $\hat{s}_e$.*

*Proof.* When an honest party $P_i$ enters Part 1 of the pessimistic phase in some epoch, it will eventually obtain a strong consistent set $\Sigma_i$ for $w - 1$. To see this, observe that when $P_i$ waits for strong consistent set $\Sigma_i$, it has already *a-delivered* sequence number $w - 1$, and hence has received $n - t$ *2-bindings* for $w - 1$. Of these, at least $t + 1$ came from honest parties who, when they eventually enter the pessimistic phase for this epoch, will send an *s-2-bind* message with a sequence number at least $w - 1$. These $t + 1$ *s-2-bind* messages form a strong consistent set for $w - 1$.

Thus, all honest parties eventually obtain strong consistent sets, and send corresponding watermark messages. Thus, all honest parties eventually obtain valid watermark proposals, and enter the multivalued Byzantine agreement with these proposals, and so by the liveness property of Byzantine agreement, all parties eventually agree on a common watermark proposal $\mathcal{M}$ with maximum sequence number $\tilde{s} = \hat{s}_e + \mathit{WinSize}$. □

**Lemma 8.7.** *If some honest party has computed $\hat{s}_e$, then*

*(i) some honest party has optimistically committed $\hat{s}_e$, and*

*(ii) no honest party has optimistically committed a sequence number $\hat{s}_e + 2 \cdot \mathit{WinSize} + 1$.*

*Proof.* Let $\tilde{s} = \hat{s}_e + \mathit{WinSize}$. To prove (i), note that $\mathcal{M}$ contains a strong consistent set for $\tilde{s}$. The existence of a strong consistent set for $\tilde{s}$ implies that at least one honest party *2-bound* $\tilde{s}$, which implies that this party has optimistically committed $\hat{s}_e$, because of the sliding window logic.

To prove (ii), suppose some honest party $P_j$ optimistically commits $\hat{s}_e + 2 \cdot \mathit{WinSize} + 1 = \tilde{s} + \mathit{WinSize} + 1$. Then by the logic of the optimistic protocol, $P_j$ must have received $n - t$ *2-bindings* for $\tilde{s} + \mathit{WinSize} + 1$, and so there must be a set $\mathcal{S}$ of $t + 1$ honest parties who sent these *2-bindings*. By the logic of the sliding window, each party in $\mathcal{S}$ has optimistically committed $\tilde{s} + 1$, and so has sent out a strong consistent set for a sequence number greater than $\tilde{s}$. By a standard counting argument, $\mathcal{M}$ must contain a contribution from some member of $\mathcal{S}$, and therefore the maximum sequence number of $\mathcal{M}$ is greater than $\tilde{s}$, which is a contradiction. □

**Lemma 8.8.** *Suppose $\hat{s}_e$ has been computed by some honest party. Let $s$ be in the range $\hat{s}_e + 1 \ldots \hat{s}_e + 2 \cdot \mathit{WinSize}$.*

*(i) If all honest parties generate* w-2-bind *messages for $s$, these messages form weak consistent set for $s$.*

*(ii) If one honest party optimistically commits $s$ to $m$, then any weak consistent set for $s$ defines $m$.*

*Proof.* Part (i) follows directly from Lemma 8.3.

To prove (ii), if an honest party optimistically committed $s$ to $m$ in epoch $e$, then he received $t + 1$ *2-bindings* of $s$ to $m$ from honest parties. Any set of $n - t$ *w-2-bind* messages must contain a contribution from one of these $t + 1$ parties, and hence defines $m$. □

**Lemma 8.9.** *Suppose that consensus holds on entry to some epoch $e$, and that some honest party has entered the pessimistic phase in this epoch.*

**local consistency:** *If some honest party commits $s$ to $m$, any honest party that also commits $s$, also commits $s$ to $m$.*

**local completeness:** *If some honest party commits $s$ to $m$, and all messages and timeouts have been delivered, and all honest parties have entered epoch $e$ and received $N_e + s$ acknowledgments, then all honest parties have committed $s$.*

**local deadlock-freeness:** *If all messages, timeouts, and acknowledgments have been delivered, and all honest parties have entered epoch $e$, then all parties have entered epoch $e + 1$.*

**boundary consistency:** *If some honest party $P_i$ commits $s$ in epoch $e$, and some honest party $P_j$ has entered epoch $e + 1$, then $P_j$ commits $s$ in epoch $e$.*

$e + 1$ **consensus:** *Consensus holds on entry to epoch $e + 1$.*

**boundary completeness:** *If some honest party enters epoch $e+1$, and all messages and timeouts have been delivered, and all honest parties have entered epoch $e$ and received $N_{e+1}-1$ acknowledgments, then all honest parties have entered epoch $e+1$.*

**at least one delivery:** *If some party enters epoch $e+1$, then $N_{e+1} > N_e$ (i.e., at least one request is delivered in epoch $e$).*

**local unique delivery:** *Any honest party* a-delivers *each request at most once in this epoch.*

*Proof (sketch).* The same proof in the *local consistency* part of Lemma 8.5 implies in this case as well that any two parties that optimistically commit $s$, commit $s$ to the same request.

If one honest party goes pessimistic, then by Lemma 8.4, all honest parties eventually go pessimistic. By Lemma 8.6, all honest parties eventually compute a common watermark $\hat{s}_e$.

By Lemma 8.7, part (i), all parties will eventually move through the loop in Part 1 of the pessimistic phase. To see this, note that since some honest party has optimistically *committed* $s$ for all $s$ up to $\hat{s}_e$, $t+1$ honest parties have *2-bound* $s$ to $m$, and so when these *2-bindings* are delivered to any honest party, that party can *commit* $s$. Note also that these *commitments* are consistent, and no party *a-delivers* a request twice, since we are only delivering requests that have been optimistically *a-delivered*, and these are guaranteed to be consistent and duplicate-free.

By Lemma 8.8, part (i), all parties will eventually move through the loop in Part 2 of the pessimistic phase, since all of the weak consistent sets that they need will eventually be available. Lemma 8.7, part (ii), and Lemma 8.8, part (ii), together imply that any request that is optimistically *a-delivered* by some honest party will be *a-delivered* in Part 2 of the pessimistic phase in the same order by all honest parties.

Note that on entry to Part 3, consensus holds: all honest parties have exactly the same value $\mathcal{D}$ as they reach this point. If no requests were *a-delivered* either optimistically or in Parts 1 or 2, then all honest parties will expect a *recover proposal* in Part 3 containing a non-empty *recover request*. This will ensure that at least one request is *a-delivered* in this epoch, but one has to check that all honest parties will eventually receive a non-empty *recover request* in this case. To see why this is so, note that one honest party, say $P_i$, must have timed out while holding a non-empty initiation queue (otherwise, no party could have gone pessimistic). But since no requests were *a-delivered* prior to Part 3, $P_i$'s *recover request* is non-empty. Thus, all honest parties move through Part 3 of the pessimistic phase consistently and without obstruction.

All of the claims in the lemma can be easily verified, given the above discussion. □

**Lemma 8.10.** *The* fairness *condition of Definition 8.1 holds with $\Delta = \mathit{WinSize} + \mathit{Thresh} + 2\cdot \mathit{PBound}$, where $\mathit{PBound} = 2\cdot \mathit{WinSize} + (n-t)\cdot \mathit{BufSize}$.*

*Proof.* Observe that *PBound* is an upper bound on the number of requests that can be *a-delivered* by any honest party in Parts 2 and 3 of the pessimistic phase of the protocol.

At any time $\tau$, let us define $\mathcal{D}^*(\tau)$ to be the value of $\mathcal{D}^*$ at time $\tau$. Also, define $e_{max}(\tau)$ to be the maximum value of $e$ for any honest party at time $\tau$.

Suppose that at some time $\tau_0$, there is a set $\mathcal{S}$ of $t+1$ honest parties such that for all $P_j \in \mathcal{S}$, the sets $\mathcal{B}^{(j)}\backslash\mathcal{D}^*$ are non-empty at time $\tau_0$. For each $P_j$ in $\mathcal{S}$, let $m_j$ denote the oldest request in $\mathcal{B}^{(j)}\backslash\mathcal{D}^*$ at time $\tau_0$.

Clearly, either $m_j$ lies in $P_j$'s initiation queue at time $\tau_1$, or $P_j$ is currently in the pessimistic phase of some epoch, its initiation queue is empty, and $m_j$ will enter its initiation queue as soon as $P_j$ enters its next epoch.

Consider any point in time $\tau_1 > \tau_0$ such that $|\mathcal{D}^*(\tau_1) - \mathcal{D}^*(\tau_0)| = \mathit{PBound}$. If some $m_j$ is in $\mathcal{D}^*(\tau_1)$, we are done; so we assume from now on that no $m_j$ is in $\mathcal{D}^*(\tau_1)$.

If some honest party is in the pessimistic phase of epoch $e_{max}(\tau_0)$ at time $\tau_0$, then since $|\mathcal{D}^*(\tau_1)-\mathcal{D}^*(\tau_0)| = \mathit{PBound}$, we must have $e_{max}(\tau_1) > e_{max}(\tau_0)$. Therefore, for all parties in $P_j \in \mathcal{S}$ such that $P_j$ is in epoch $e_{max}(\tau_1)$ at time $\tau_1$, it must hold that $m_j$ is in $P_j$'s initiation queue at time $\tau_1$.

At any point in time after $\tau_1$, if $m_j$ lies in $P_j$'s initiation queue, the value of $it(m_j)$ is the minimum among all requests in its initiation queue.

We define the quantity $it_{max}$ as follows: if no party in $\mathcal{S}$ is in epoch $e_{max}(\tau_1)$ at time $\tau_1$, then $it_{max}$ is 0; otherwise, $it_{max}$ is the maximum value of $it(m_j)$ for any party $P_j$ in $\mathcal{S}$ that is in epoch $e_{max}(\tau_1)$ at time $\tau_1$.

An honest party that *a-delivers* "too many" requests, none of which lie in its initiation queue, will refuse to send *1-bindings*. The precise statement of this is as follows.

Consider any point in time $\tau_2 > \tau_1$. For any party $P_j \in \mathcal{S}$, if $P_j$ has not *a-delivered* $m_j$ at time $\tau_2$, then $P_j$ has not generated any *1-bindings* in in epoch $e_{max}(\tau_1)$ for sequence numbers $it_{max}$ + *WinSize* + *Thresh* or above at time $\tau_2$.

Further suppose that at time $\tau_2$, no $m_j$ is in $\mathcal{D}^*(\tau_2)$. Then we claim that no party has entered epoch $e_{max}(\tau) + 1$. To see this, note that in Part 3 of the pessimistic phase, since a valid *recover proposal* must contain contributions from $n - t$ parties, one of these must come from a party $P_j$ in $\mathcal{S}$, who would have contributed a *recover request* containing $m_j$. Also, since no party $P_j$ in $\mathcal{S}$ issued *1-bindings* for sequence numbers $it_{max}$ + *WinSize* + *Thresh* or above, no honest party could have optimistically committed such a sequence number. Therefore, $|\mathcal{D}^*(\tau_2) - \mathcal{D}^*(\tau_1)| \leq \textit{WinSize} + \textit{Thresh} + \textit{PBound}$.

That proves the lemma. □

We now state and prove our main theorem.

**Theorem 8.11.** *Our protocol satisfies the properties in our Definition 8.1 for atomic broadcast.*

*Proof.* We first define some auxiliary notions.

Let us say that an honest party $P_i$ *globally commits a sequence number $s$ to a request $m$*, if $m$ is the $s$th request (counting from zero) *a-delivered* by $P_i$.

We then define *consistency*, *completeness*, and *unique delivery* as follows.

**consistency:** If some honest party globally commits $s$ to $m$, any honest party that also globally commits $s$, also globally commits $s$ to $m$.

**completeness:** If some honest party globally commits $s$ to $m$, and all messages and timeouts have been delivered, and all honest parties have received $s$ acknowledgments, then all honest parties have globally committed $s$.

**unique delivery:** Any honest party *a-delivers* each request at most.

It is clear that *consistency* and *completeness* hold if and only if *agreement* and *total order* (from Definition 8.1) hold.

One can prove by a completely routine induction argument, using Lemmas 8.9 and 8.5, that *consistency*, *completeness*, *deadlock-freeness*, *unique delivery* hold.

*Validity* trivially follows from *unique delivery* and by simple inspection of the protocol.

*Efficiency* is also follows from the *at least one delivery* property in Lemma 8.9, and by simple inspection of the protocol.

*Fairness* follows from Lemma 8.10. □

## 8.4 Secure Causal Atomic Broadcast

Secure causal atomic broadcast (SC-ABC) is a useful protocol for building secure applications that use state machine replication in a Byzantine setting. It provides atomic broadcast, which ensures that all recipients receive the same sequence of messages, and also guarantees that the payload messages arrive in an order that maintains "input causality," a notion introduced by Reiter and Birman [RB94]. Informally, input causality ensures that a Byzantine adversary may not ask the system to deliver any payload message that depends in a meaningful way on a yet undelivered payload sent by an honest client. This is very useful for delivering client requests to a distributed service in applications that require the contents of a request to remain secret until the system processes it. Input causality is related to the standard causal order (going back to Lamport [Lam78]), which is a useful safety property for distributed systems with crash failures, but is actually not well defined in the Byzantine model [HT93].

Input causality can be achieved if the sender encrypts a message to broadcast with the public key of a threshold cryptosystem for which all parties share the decryption key [RB94]. The ciphertext is then broadcast using an atomic broadcast protocol; after delivering it, all parties engage in an additional round to recover the message from the ciphertext.

In our description of secure causal atomic broadcast, one of the parties acts as the sender of a payload message. If SC-ABC is used by a distributed system to broadcast client requests, then encryption and broadcasting is taken care of by the client. In this case, additional considerations are needed to ensure proper delivery of the replies from the service (see [RB94] for those details).

### 8.4.1 Problem Statement

Associated with any instance of a secure causal atomic broadcast protocol with tag $ID$ is an encryption algorithm $E_{ID}$. It should be possible to infer this algorithm from the dealer's public output. $E_{ID}$ is a probabilistic algorithm that maps a message $m$ to a ciphertext $c$. We call $c = E_{ID}(m)$ an encryption of $m$ (with tag $ID$). Since the encryption algorithm is probabilistic, there will in general be many different encryptions of a given message; indeed, this will necessarily be the case if the system is to be secure.

An application that wants to securely broadcast a payload message should first encrypt it using $E_{ID}$ and invoke the broadcast protocol with the resulting ciphertext. Since $E_{ID}$ is publicly known, also clients from outside the group $P_1, \ldots, P_n$ can produce ciphertexts.

A secure causal atomic broadcast protocol is activated when $P_i$ receives an input message of the form

$$(ID, \texttt{in}, \texttt{s-broadcast}, c).$$

We say $P_i$ *s-broadcasts* $c$ with tag $ID$.

Unlike the other broadcasts, delivery consists of two distinct steps: the first is the generation of an output message of the form

$$(ID, \texttt{out}, \texttt{s-schedule}, c),$$

and the second is the generation of an output message of the form

$$(ID, \texttt{out}, \texttt{s-reveal}, m).$$

We shall require that honest parties generate sequences of such pairs of output messages—there must never be two consecutive `s-schedule` or `s-reveal` messages. When the `s-schedule` message is generated, we will say that $P_i$ *s-schedules* the ciphertext $c$ (with tag $ID$). When the `s-reveal` message is generated, we will say that $P_i$ *s-delivers* the ciphertext $c$ (with tag $ID$), where $c$ is the most recently *s-scheduled* ciphertext; we call $m$ the *associated cleartext*.

**Definition 8.2 (Secure Causal Atomic Broadcast).** A secure causal atomic broadcast protocol satisfies the properties of an atomic broadcast protocol, where the *s-broadcast* and *s-delivery* of ciphertexts in the secure causal atomic broadcast protocol play the role of the *a-broadcast* and *a-delivery* of payload messages in an atomic broadcast protocol.

Additionally, the following conditions hold.

**Message Secrecy:** According to the basic system model, the parties run an atomic broadcast protocol (and possibly other broadcast protocols), and the adversary plays the following game:

B1. The adversary interacts with the honest parties in an arbitrary way.

B2. The adversary chooses two messages $m_0$ and $m_1$ and a tag $ID$; it gives them to an "encryption oracle." The oracle chooses a bit $B$ at random and computes an encryption $c$ of $m_B$ with tag $ID$, and gives this ciphertext to the adversary.

B3. The adversary continues to interact with the honest parties subject only to the condition that no honest party *s-schedules* $c$ with tag $ID$.

B4. Finally, the adversary outputs a bit $\hat{B}$.

Then, for any adversary, the probability that $\hat{B} = B$ must exceed $\frac{1}{2}$ only by a negligible amount.

**Message Integrity:** According to the basic system model, the parties run an atomic broadcast protocol (and possibly other broadcast protocols), and the adversary plays the following game:

C1. The adversary interacts with the honest parties in an arbitrary way.

C2. The adversary chooses a message $m$ and a tag $ID$, and gives it to an "encryption oracle." The oracle computes an encryption $c$ of $m$ with tag $ID$, and gives this ciphertext to the adversary.

C3. The adversary continues to interact with the honest parties in an arbitrary way.

We say the adversary wins the game if at some point an honest party *s-delivers* $c$ with tag $ID$, but corresponding cleartext $m'$ is not equal to $m$. Then, for any adversary, the probability that it wins this game is negligible.

**Message Consistency:** If two parties honest parties *s-deliver* the same ciphertext $c$ with tag $ID$, then with all but negligible probability, the associated cleartexts are the same.

It is easy to verify that this definition implies input causality in the sense of Reiter and Birman [RB94], i.e., that a cleartext remains hidden from the adversary until the corresponding ciphertext is *s-scheduled.* But the cleartext may be revealed to the adversary before the first honest party outputs it in a `s-reveal` message, and this is also the reason for introducing our two-step delivery process. Although this is necessary for the proper definition of security, *s-scheduling* a ciphertext might be omitted in a practical implementation.

The *message integrity* condition gives clients access to the broadcast protocol for cleartext payload messages, and implies that payloads contained in correctly encrypted ciphertexts are actually output by the honest parties.

### 8.4.2 A Protocol for Secure Causal Atomic Broadcast

Protocol SC-ABC (Figure 8.6) implements secure causal atomic broadcast. It uses an $(n, t+1)$-threshold cryptosystem $\mathcal{E}_1$ that is secure against adaptive chosen ciphertext attacks (see Chapter 3) for which the parties share the decryption key. It also uses an atomic broadcast protocol according to Section 8.1.

During initialization, the dealer generates a public key for $\mathcal{E}_1$, together with the corresponding private key shares, and distributes them according to the initialization algorithm of $\mathcal{E}_1$.

For a tag $ID$, $E_{ID}(m)$ is computed by applying the encryption algorithm of $\mathcal{E}_1$ to $m$ with label $ID$, using the generated public key of the cryptosystem.

We emphasize that all instances of the secure causal broadcast protocol share the same public key for $\mathcal{E}_1$, and so the use of *labeled ciphertexts* is essential to properly "isolate" different instances of the protocol from one another.

To *s-broadcast* a ciphertext $c$, we simply *a-broadcast* $c$. Upon *a-delivery* of a ciphertext $c$, a party *s-schedules* $c$. Then it computes a decryption share $\delta$ and sends this to all other parties in an `s-decrypt` message containing $c$. It waits for $t+1$ `s-decrypt` messages pertaining to $c$. Once they arrive, it recovers the associated cleartext and *s-delivers* $c$. After receiving the acknowledgment, the party continues processing the next *a-delivery* by generating the corresponding acknowledgment. The details are in Figure 8.6. For ease of notation, the protocol is formulated using a FOREVER loop; it can be decomposed into the respective message handlers in straightforward way.

```
Protocol SC-ABC for party $P_i$ and tag $ID$
Initialization:
        open an atomic broadcast channel with tag $ID|$scabc
upon receiving $(ID, \texttt{in}, \texttt{s-broadcast}, c)$:
        a-broadcast $c$ with tag $ID|$scabc
loop forever
        wait for the next message $c$ that is a-delivered with tag $ID|$scabc
        compute an $\mathcal{E}_1$-decryption share $\delta$ for $c$ with label $ID$
        output $(ID, \texttt{out}, \texttt{s-schedule}, c)$
        send the message $(\texttt{s-decrypt}, c, \delta)$ to all parties
        $\delta_j \leftarrow \perp \quad (1 \leq j \leq n)$
        wait for $t+1$ messages $(\texttt{s-decrypt}, c, \delta_j)$ from distinct parties that contain valid
           decryption shares for $c$ with label $ID$
        combine the decryption shares $\delta_1, \ldots, \delta_n$ to obtain a cleartext $m$
        output $(ID, \texttt{out}, \texttt{s-reveal}, m)$
        wait for an acknowledgment
        acknowledge the last a-delivered message with tag $ID|$scabc
end loop
```

Figure 8.6: SC-ABC for secure causal atomic broadcast.

**Theorem 8.12.** *Given an atomic broadcast protocol and assuming $\mathcal{E}_1$ is a $(n, t+1)$-threshold cryptosystem secure against adaptive chosen-ciphertext attacks, Protocol* SC-ABC *provides secure causal atomic broadcast for $n > 3t$.*

*Proof.* We have to show that the protocol implements atomic broadcast and satisfies message secrecy and message integrity conditions.

We first show *deadlock-freeness*. Suppose an honest party $P_i$ has *s-broadcast* $c$ and all associated messages have been delivered and all acknowledgments have been generated. Thus, $P_i$ has *a-broadcast* $c$. By the deadlock-freeness of the atomic broadcast protocol and because the messages associated to the secure broadcast contain also all those associated to the atomic broadcast, $c$ has been *a-delivered*. According to the agreement condition of atomic broadcast, all honest parties have therefore generated decryption shares for $c$ and sent `s-decrypt` messages to all parties. Thus, $P_i$ has received at least $t+1$ valid shares for $c$. But then $P_i$ has also *s-delivered* $c$.

It is perhaps interesting to note that the above proof of *deadlock-freeness* made essential use of *both* the *deadlock-freeness* and *agreement* properties of the underlying atomic broadcast protocol.

For *agreement*, suppose that an honest $P_i$ has *s-delivered* $c$ and $P_j$ has not, and yet, all associated messages have been delivered and acknowledgments have been generated for those parties who have not *s-delivered* $c$. By the agreement condition of the underlying atomic broadcast, all other honest parties must also have *a-delivered* $c$. Thus, they have generated decryption shares and also $P_j$ has received at least $t+1$ valid shares for $c$. Therefore, $P_j$ must have *s-delivered* $c$, a contradiction.

To show *efficiency*, we must bound the amount of work done (as measured by communication complexity) per *s-delivered* message. But since the *s-delivery* messages is synchronized with the *a-delivery* of ciphertexts in Protocol SC-ABC, the number of *a-delivered* messages exceeds the number of *s-delivered* ones by at most one, and efficiency follows from the efficiency condition of the atomic broadcast protocol.

Note that without this synchronization, we could not achieve efficiency, since the lower level atomic broadcast protocol could "run ahead" of the higher level secure causal atomic broadcast protocol—lots of messages would be generated, but very few messages would be *s-delivered.*

It is easy to see that the remaining broadcast properties (*total order*, *validity*, and *fairness*) hold as well, using the corresponding properties of the underlying atomic broadcast.

*Message secrecy, integrity, and consistency* follow easily from the properties of the underlying threshold encryption scheme. □

# Chapter 9

# Conclusions

"God has put me on Earth to do certain things; now I am so far behind that I will never die."

(**Bill Watterson**, *Calvin and Hobbes*)

This thesis addresses the problem of coordinating mutually distrusting parties connected by an unreliable asynchronous network. By using new cryptographic primitives, we developed a protocol suite on top of which distributed applications can run without needing to worry about the problems induced by malicious participants and the asynchrony of the network.

## Tied Knots

The general problems addressed in this thesis are not new; much work has been done on fault-tolerant middleware, and implementations have already proven their value in the real world [BJ87, vRBG$^+$95, Hay98].

The main difference between above protocols and our work is that we assumed a stronger model: we not only tolerate Byzantine faults, we also eliminate timing assumptions. The impact of the different model is larger than it initially seems. Our protocols have a completely different approach from the protocols used in real implementations so far. Most practical protocols provide a group membership protocol that eliminates bad parties and maintains consensus on the state of the group. By using randomization techniques, we can solve the elemental problem of Byzantine agreement directly, and our higher protocols then build on this primitive.

Thus, our protocol suite cannot be seen as a strengthened version of previous implementations, but rather as implementing a previously theoretical approach in a practical way.

Beside providing a protocol suite that might be of practical value one day, there are three main theses that we prove in this dissertation:

**Randomization is practical.** In the area of fault tolerance, randomized protocols have been neglected for a long time. After an initial boom triggered by the first protocols that could circumvent the FLP [FLP85] impossibility result [BO83, Rab83], randomized protocols were assumed to be a nice theoretical concept, but inherently unpractical.

The protocols presented in this work prove that randomization is a very practical concept, Both in a theoretical analysis and in a prototype implementation, our protocols can compete with non-randomized protocols in the literature, despite providing more security.

My hope is that the existence of efficient randomized protocols can revitalize the randomized approach, and also rid it of the taste of black magic it seems to have in large parts of the community (randomization exists, but one is not supposed to use it).

**Cryptography and Distributed systems can greatly benefit from each other.**

"And now comes the session with all those horribly technical crypto papers nobody understands and where everybody falls asleep within five minutes"

(*Anonymous, unaware of the fact that I was the next speaker*)

The above statement illustrates a cultural gap between the distributed systems community and the cryptography community.

I envision this thesis as one brick in the bridge between the communities; it demonstrates how distributed protocols can greatly benefit from a broader view on cryptography that does not idealise it to some abstract functions that provide encryption and signatures.

We also provide a model for distributed systems that incorporates standard cryptographic assumptions. In this model, cryptography and distributed systems can be comfortably combined. It is close enough to the models used in distributed systems so that it easily transforms definitions from the I/O–automata model into our computationally bounded setting. While it is not nearly so precise as modern formal models for multiparty computation as they are developed in the cryptographic community [PW01, Can01], it is sufficient to do end-to-end security proofs that do not need to idealize away essential assumptions.

**Byzantine, fully asynchronous fault tolerance is plausible.** So far, most practical work done in fault tolerance sacrificed security for performance. All implementations that I am aware of depend on timing, and most of them work in the crash failure model only.

The protocols presented in this thesis form a protocol stack that provides services similar to existing products like Isis [BJ87], Horus [vRBG+95], Rampart [Rei94], Ensemble [Hay98], and other implementations of dependable middleware. These services provide a powerful synchronization mechanism that shields higher-level applications from having to worry about asynchrony and failures. All our protocols are practicable, offering both optimal security (i.e., the highest number of corruptions possible tolerable and no timing assumptions or failure detectors) as well as good performance (especially with optimistic protocols) that can compete with the performance of timed protocols.

Although there is long road ahead in terms of architecture and implementation issues — after all, timed protocols have head start of 20 in this field — it seems plausible that our techniques can be used in "real-world" middleware to improve security without sacrificing performance.

## Loose ends

The thesis has presented all building blocks necessary to create dependable distributed middleware that is powerful enough to mask the worst problems introduced by corrupt participants and asynchrony. Although this work is complete in the sense that it contains all protocols needed to present standard interfaces (to some extent), some compromises had to be taken along the way, and many problems remained open.

**Improved Cryptography.** The cryptographic primitives used in this thesis imply some artificial, nagging restrictions. Our corruption model is static, i.e., the adversary must decide which parties to corrupt independently from the behavior of the system. Allowing for adaptive corruptions would provide stronger security guarantees. While the higher level protocols presented here do not require static adversaries, it is currently not known how to realize all of our threshold-cryptographic primitives efficiently with adaptive security.

The same holds for the random oracle model — it is a very useful and practical crutch used to prove practical cryptographic protocols, and we do not know how to do the proofs without it. So far, its use for the threshold signatures can be avoided by using sets of signatures or the broadcast-based threshold signatures from Section 3.3.4. However, we do not know how to prove the security of the coin-tossing scheme without the random oracle, and we do not have an alternative scheme we could use instead.

The need for random oracles and static adversaries is somewhat unfortunate, but necessary for the cryptographic proofs. Even though it is more an artifact created by the proof technique rather than a serious threat, it would be nicer to offer threshold cryptography that does not depend on this model.

**Key Management and Group Membership.** Most other approaches to create dependable middleware [BJ87, Rei95, Hay98, vRBG$^+$95, MMSA94] build on group membership; a low-level, timing-based protocol is used to (hopefully) exclude faulty parties from the active group, allowing higher level protocols to rely on every member of this group being honest.

Our protocols do not use this mechanism; they try to tolerate rather than detect faulty parties. On the one hand, this is a strength of our protocols: we do not *need* to identify and exclude corrupted parties as the traditional approach does. On the other hand, maintaining group membership in our model is not straightforward: an excluded party may still own all cryptographic keys and give them to the adversary. The problems for group-key management are similar to those of a dealer-free initialization: a primitive is needed that reshares keys in such a way that the new group of parties uses keys, independent of the ones that excluded parties still know.

Using the same techniques as for group membership, we might also be able to replace the trusted dealer currently required to initialize the system. This dealer seems to contradict the entire notion of eliminating a single point of failure, although it can well be justified — after all, the trusted dealer is needed only once and can be destroyed after initialization. Nevertheless, it would be nice if the system could be boot-strapped without a trusted dealer. Key management could also be used to provide proactivity, i.e., that occasionally all cryptographic keys are recomputed to render potentially lost keys invalid and useless [CGHN97].

**Extensions and Implementation for Higher Protocols.** For the Byzantine agreement protocol ABBA, we introduced several variations (e.g. adversary structures and link failures) that help to optimize it for a real setting, as well as an implementation. It is still left open to apply equivalent variations for the higher-level protocols, and to make a full implementation of the protocol stack with which useful experiments can be made.

**Architectural Issues.** In terms of an architecture, our protocols are in a state in which failure-detector-based protocols were about 15 years ago [BJ87]; the functionality is the main issue, and little thought has been spent on a good architecture. Since then, failure-detector-based protocols have evolved greatly; they consist of many "microprotocols" that can be glued together in a very flexible way to achieve the functionality required by an application.

By using randomized protocols that are not based on group membership, fault tolerance is achieved by very different means. This prevents us from simply taking over an architecture developed for an existing system. Exploring the impact of the different approaches on the architecture, and finding ways to provide a similar architecture for our approach seems a worthwhile, but laborious task.

Wir stehen selbst enttäuscht und sehn betroffen

Den Vorhang zu und alle Fragen offen.

Bertold Brecht

# Bibliography

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR95-1534, Cornell University, Computer Science Department, August 25, 1995.

[AT96] M. K. Aguilera and S. Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In Ö. Babaoglu and K. Marzullo, editors, *Distributed Algorithms, 10th International Workshop, WDAG '96*, volume 1151 of *Lecture Notes in Computer Science*, pages 29–39, Bologna, Italy, 9–11 October 1996. Springer.

[AW98] H. Attiya and J. Welch. *Distributed Computing.* McGraw-Hill, 1998.

[Bab74] C. Babbage. *The Origins of Digital Computers: Selected papers*, chapter C. Babbage. On the mathematical powers of the calculating engine (December 1837). Unpublished Manuscript. Buxton MS7, Museum if the History of Science., pages 17–52. Springer, 1974.

[BCG93] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993.

[BG93] P. Berman and J. A. Garay. Randomized distributed agreement revisited. *23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23)*, pages 412–419, 1993.

[Bir01] K. Birman. Horus/Ensemble Mailinglist, Cornell University, October 2001. Available form the author.

[BJ87] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. Technical Report TR87-811, Cornell University, Computer Science Department, February 1987.

[BKR94] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computation with optimal resilience. In *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[BL90] J. Benaloh and J. Leichter. Generalized secret sharing and monotone functions. In S. Goldwasser, editor, *Advances in cryptology — CRYPTO 88*, number 403 in Lecture Notes in Computer Science, pages 27–36, Santa Barbara, CA, USA, August 1990. Springer-Verlag.

[BNDDS92] A. Bar-Noy, D. Dolev, C. Dwork, and H. R. Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, April 1992.

[BO83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, Canada, 17–19 August 1983.

[BOEY91] M. Ben-Or and R. El-Yaniv. Interactive consistency in constant time. Unpublished, 1991. Referenced by [FM97] and [CR93].

[BOKR94] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192. ACM Press, 1994.

[Bon98] D. Boneh. The decision Diffie-Hellman problem. *Lecture Notes in Computer Science*, 1423:48–63, 1998.

[BR94] M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *Proc. CRYPTO 93*, pages 232–249. Springer, 1994. Lecture Notes in Computer Science No. 773.

[BR95] M. Bellare and P. Rogaway. Provably secure session key distribution—the three party case. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 57–66, Las Vegas, Nevada, 29 May–1 June 1995.

[Bra84] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In PODC84 [POD84], pages 154–162.

[Bra01] B. Brassel. *Die Ideale Logik.* PhD thesis, University Aachen, 2001.

[BS93] D. Beaver and N. So. Global, unpredictable bit generation without broadcast. In T. Helleseth, editor, *Advances in Cryptology—EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 424–434. Springer-Verlag, 1994, 23–27 May 1993.

[BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[Can00] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):107–142, 2000.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 01, to appear*, 2001.

[Cas00] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, Massachusetts Institute of Technology, November 2000.

[CD89] B. Chor and C. Dwork. Randomization in Byzantine agreement. In S. Micali, editor, *Randomness and computation*, volume 5 of *Advances in Computing Research*, pages 443–497. JAI Press, 1989.

[CEvdGP87] D. Chaum, J.-H. Evertse, J. van de Graaf, and R. Peralta. Demonstrating possession of a discrete logarithm without revealing it. In A. Odlyzko, editor, *Proc. CRYPTO 86*, pages 200–212. Springer-Verlag, 1987. Lecture Notes in Computer Science No. 263.

[CF95] F. Cristian and C. Fetzer. Timed asynchronous systems: A formal model. Technical Report CSE95-454, UCSD, 1995.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, June 1999. To appear. http://www.cs.ucsd.edu/˜cfetzer/MODEL/.

[CG99] R. Canetti and S. Goldwasser. An efficient threshold public-key cryptosystem secure against adaptive chosen-ciphertext attack. In J. Stern, editor, *Advances in Cryptology: EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 1999.

[CGHN97] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: Long-term protection against break-ins. *CryptoBytes*, 3(1):1–7, 1997.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. Cryptology ePrint Archive, Report 2001/006, March 2001. `http://eprint.iacr.org/`.

[CKS00] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 123–132, July 16–19 2000.

[CL99a] M. Castro and B. Liskov. Authenticated byzantine fault tolerance without public-key cryptography. Tech. Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, June 1999.

[CL99b] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. Third Symp. Operating Systems Design and Implementation*, 1999.

[CP92] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *Advances in Cryptology – CRYPTO ' 92*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, Berlin Germany, 1992.

[CP01] C. Cachin and J. Poritz. Hydra: Secure Replication on the Internet. To appear., 2001.

[CR93] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993.

[CT91] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, Montreal, Quebec, Canada, 19–21 August 1991.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[Des88] Y. Desmedt. Society and group oriented cryptography: A new concept. In C. Pomerance, editor, *Advances in Cryptology: CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 120–127. Springer, 1988.

[DF90] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology: CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1990.

[DF92] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology: CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 457–569. Springer, 1992.

[DGG00] Doudou, Guerraoui, and Garbinato. Abstractions for devising byzantine-resilient state machine replication. In *SRDS: 19th Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, 2000.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DM94] D. Dolev and D. Malki. Consensus made practical. Technical report, Hebrew University of Jerusalem, July 1994.

[DRS90] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[DS98] A. Doudou and A. Schiper. Muteness Detectors for Consensus with Byzantine Processes, 1998.

[FC95] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, pages 86–91, Newport Beach, CA, December 1995. http://www-cse.ucsd.edu/users/cfetzer/CONS/cons.html.

[FHM98] M. Fitzi, M. Hirt, and U. Maurer. Trading correctness for privacy in unconditional multi-party computation. In H. Krawczyk, editor, *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*. Springer, 1998.

[FHM99] M. Fitzi, M. Hirt, and U. Maurer. General adversaries in unconditional multi-party computation. In K. Y. Lam, E. Okamoto, and C. Xing, editors, *Advances in Cryptology - ASIACRYPT '99*, volume 1716 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, 1999.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinsky, editor, *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*. Springer, 1983. Also published as Tech. Report YALEU/DCS/TR-273, Department of Computer Science, Yale University.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FM97] P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, August 1997.

[FM00] M. Fitzi and U. Maurer. From partial consistency to global broadcast. In ACM, editor, *Proceedings of the thirty second annual ACM Symposium on Theory of Computing: Portland, Oregon, May 21–23, [2000]*, pages 494–503, 2000. ACM order number 508000.

[FS87] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology: CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1987.

[Gär98] F. C. Gärtner. Fundamentals of fault tolerant distributed computing in asynchronous environments. Technical Report TUD-BS-1998-02, Darmstadt University of Technology, Darmstadt, Germany, July 1998. To appear in *ACM Computing Surveys*, 31(1), March 1999.

[GJKR99] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In J. Stern, editor, *Advances in Cryptology: EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 295 – 310. Springer, 1999.

[GLR95] L. Gong, P. Lincoln, and J. Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In R. K. Iyer, M. Morganti, W. K. Fuchs, and V. Gligor, editors, *Dependable Computing for Critical Applications—5*, volume 10 of *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, September 1995. IEEE Computer Society.

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, April 1988.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, February 1989.

[Gol01a] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[Gol01b] O. Goldreich. *Secure Multi–Party Computation (Working Draft, Version 1.3)*. 2001. http://www.wisdom.weizmann.ac.il/˜oded/pp.html.

[GP90] O. Goldreich and E. Petrank. The best of both worlds: Guaranteeing termination in fast Byzantine Agreement protocols. *Information Proceeding Letters*, 36:45–49, October 1990.

[GS96] R. Guerraoui and A. Schiper. Consensus Service: a modular approach for building agreement protocols in distributed systems. In *Proc. 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, 1996.

[Hay98] M. G. Hayden. *The Ensemble System.* PhD thesis, Cornell University, January 1998.

[Hec76] H. Hecht. Fault-tolerant software for real-time applications. *Computing Surveys*, 8(4):391–407, 1976.

[Her00] M. Herlihy. Personal communication, 2000.

[HM97] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 25–34, Santa Barbara, California, 21–24 August 1997.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. J. Mullender, editor, *Distributed Systems.* ACM Press & Addison-Wesley, New York, 1993. An expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[ISN87] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *Proceedings IEEE Globecom '87*, pages 99–102. IEEE, 1987.

[KMMS97] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[KMMS98] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *31st Hawaii International Conference on System Sciences*, pages 317–326, Kona, Hawaii, January 1998. IEEE.

[KN01] M. Kwiatkowska and G. Norman. Automated verification of a randomized byzantine agreement protocol. Technical Report CSR-01-11, University of Birningham, 2001.

[KS01] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. Cryptology ePrint Archive, Report 2001/022, March 2001. `http://eprint.iacr.org/`.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT87] N. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, September 1989.

[Lub96] M. Luby. *Pseudorandomness and Cryptographic Applications.* Princeton University Press, 1996.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[MGB01] P. Maniatis, T. Giuli, and M. Baker. Enabeling the long-term archival of signed documents through time stamping. Technical report, Stanford University, 2001.

[MMR00] D. Malkhi, M. Merritt, and O. Rodeh. Secure reliable multicast protocols in a WAN. *Distributed Computing*, 13(1):19–28, 2000.

[MMSA94] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[Mon48] B. d. Montesquieu, Charles de Secondat. *The Spirit of Law.* 1748.

[MPL80] R. S. M. Pease and L. Lamport. Reaching agreement in the presence of faults, 1980.

[MR91] S. Micali and T. Rabin. Collective coin tossing without assumptions nor broadcasting. In A. J. Menezes and S. A. Vanstone, editors, *Proceedings of Advances in Cryptologie (CRYPTO '90)*, volume 537 of *LNCS*, pages 253–267, Berlin, Germany, August 1991. Springer.

[MR97] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computation. In *CSFW97*, 1997.

[MS95] S. Micali and R. Sidney. A simple method for generating and sharing pseudo-random functions, with applications to clipper-like key escrow systems. In D. Coppersmith, editor, *Advances in Cryptology: CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 185–196. Springer, 1995.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press LLC, 1997.

[NPR99] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In J. Stern, editor, *Advances in Cryptology: EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*. Springer, 1999.

[NR97] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *Proc. 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1997.

[PDS01] R. Pletka, P. Droz, and B. Stiller. A buffer management scheme for bandwidth and delay differentiation using a virtual scheduler. In P. Lorenz, editor, *ICN 2001*, volume 2093 of *Lecture Notes in Computer Science*, pages 218 – 234. Springe-Verlag, 2001.

[Ped91] T. P. Pedersen. A threshold cryptosystem without a trusted party. In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer-Verlag, 8–11 April 1991.

[Per85] K. J. Perry. Randomized Byzantine Agreement. *IEEE Transactions on Software Engeneering*, 11(6):539 – 546, June 1985.

[Pie65] W. H. Pierce. *Failure Tolerant Computer Design.* Academic Press, New York, 1965.

[POD84] *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, 27–29 August 1984.

[PS98] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, September 1998.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PW00] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *ACM Conference on Computer and Communication Security*, pages 245–254, 2000.

[PW01] B. Pfitzmann and M. Waidner. Model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security and Privacy*, pages 184–200, 2001.

[Rab83] M. O. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, pages 403–409, Tucson, Arizona, 7–9 November 1983. IEEE.

[Rab98] T. Rabin. A simplified approach to threshold and proactive RSA. In H. Krawczyk, editor, *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*. Springer, 1998.

[Ran75] B. Randell. System structuring for software fault-tolerance, 1975.

[RB94] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[Rei95] M. K. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems*, pages 99 – 110, 1995.

[SAAAA95] H. M. Sayeed, M. Abu-Amara, and H. Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with Byzantine faulty links. *Distributed Computing*, 9(3):147–156, 1995.

[Sch00] A. Schiper. Personal communication, 2000.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Advances in Cryptology: EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*. Springer, 1997.

[Sho00] V. Shoup. Practical threshold signatures. In B. Preneel, editor, *Advances in Cryptology: EUROCRYPT 2000*, Lecture Notes in Computer Science, pages 207–220. Springer, 2000.

[Sim88] G. Simmons. How to (really) share a secret. In S. Goldwasser, editor, *Proc. CRYPTO 88*, pages 390–449. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 403.

[SRA81] A. Shamir, R. L. Rivest, and L. M. Adleman. Mental poker. In D. Klarner, editor, *The Mathematical Gardner*, pages 37–43. Wadsworth, Belmont, California, 1981.

[Sta96] M. Stadler. Publicly verifiable secret sharing. In U. Maurer, editor, *Advances in Cryptology: EUROCRYPT '96*, volume 1233 of *Lecture Notes in Computer Science*, pages 190–199. Springer, 1996.

[TC84] R. Turpin and B. A. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18:73–76, 1984.

[Tou84] S. Toueg. Randomized Byzantine Agreements. In PODC84 [POD84], pages 163–178.

[VA95] P. Verissimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Winter 1995 Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS )*, 7 (4):35 – 39, 1995.

[VC99] P. Verissimo and A. Casimiro. The timely computing base. Technical Report DI/FCUL TR-99-2, University of Lisboa, May 1999.

[Ver01] P. Verissimo. Personal communication, 2001.

[vRBG$^+$95] R. van Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Cornell University, Computer Science Department, March 1995.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[Wen72] J. H. Wensley. Sift software implemented fault tolerance. In *AFIPS 1972 Fall Joint Computer Conference (FJCC)*, pages 243–253, 1972.

[WLG$^+$78] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Gerrn, K. N. Levitt, P. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240 – 1255, October 1978.

[Yao82a] A. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symp. on Foundations of Comp. Science*, pages 160–164, Chicago, 1982. IEEE.

[Yao82b] A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.

[Zam96] A. Zamsky. A randomized Byzantine agreement protocol with constant expected time and guaranteed termination in optimal (deterministic) time. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 201–208, Philadelphia, Pennsylvania, USA, 23–26 May 1996.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | | A | | $\alpha$ | | $\mathcal{A}$ | symbol of the Adversary |
| b | a bit | B | | $\beta$ | | $\mathcal{B}$ | |
| | number of byzantine parties | | | | | | Set of byzantine parties |
| c | number of crashing parties | C | | $\gamma$ | | $\mathcal{C}$ | Set of crashing parties |
| c | a ciphertext | C | | $\Gamma$ | an access structure | | |
| d | digest (output of a hash function) | D | | $\delta$ | a negligible function | $\mathcal{D}$ | |
| | | | | $\Delta$ | | | |
| e | ca. 2.72 | E | | $\epsilon$ | something very small | $\mathcal{E}$ | encryption scheme |
| | | E | | $\epsilon$ | a negilgible function | | |
| f | number of occuring faults | F | a distributed Function | $\zeta$ | | $\mathcal{F}$ | |
| | a function | | | | | | |
| g | a group element | G | a group | $\eta$ | | $\mathcal{G}$ | |
| h | a hash function | H | hash function | $\theta$ | | $\mathcal{H}$ | set of honest parties |
| i | counter / index | I | | $\iota$ | | $\mathcal{I}$ | |
| j | counter / index | J | | | | $\mathcal{J}$ | |
| k | general threshold | K | | $\kappa$ | security parameter | $\mathcal{K}$ | |
| l | counter | L | | $\lambda$ | Lagragne interpolator | $\mathcal{L}$ | a liveness property |
| m | a message | M | | $\mu$ | combined threshold signature | $\mathcal{M}$ | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| n | number of parties | N | | $\nu$ | signature share | $\mathcal{N}$ | |
| o | | O | | $\xi$ | | $\mathcal{O}$ | |
| p | a polynomial | P | a party | $\pi$ | ca. 3.14 | $\mathcal{P}$ | set of all parties |
| | a prime | | | $\Pi$ | | | |
| q | another polynomial | Q | | | | $\mathcal{Q}$ | |
| | another prime | | | | | | |
| r | round number | R | | $\rho$ | | $\mathcal{R}$ | |
| s | sequence number | S | | $\sigma$ | a signature | $\mathcal{S}$ | digital signature scheme |
| | a secret | | | $\Sigma$ | | | |
| t | number of corruptios | T | a polynomial | $\tau$ | | $\mathcal{T}$ | function to define termination |
| u | | U | a polynomial | $\upsilon$ | | $\mathcal{U}$ | |
| v | input value (ABBA) | V | | $\phi$ | | $\mathcal{V}$ | |
| w | input value (BACS) | W | | | | $\mathcal{W}$ | set of input values |
| x | general purpose variable | X | random variable | $\chi$ | | $\mathcal{X}$ | general purpose set |
| | | X | an Event | | | | |
| y | | Y | | $\psi$ | | $\mathcal{Y}$ | |
| z | | Z | | $\omega$ | | $\mathcal{Z}$ | An adversary structure |
| | | | | $\Omega$ | Complexity measure | | |