

Fair exchange is incomparable to consensus

Simona Orzan¹ and Mohammad Torabi Dashti²

¹ Technical University of Eindhoven, The Netherlands

² ETH Zürich, Switzerland

Abstract. In asynchronous systems where processes are prone to crash failures, we show that fair exchange is incomparable to distributed consensus. By incomparability we mean there exist failure detector classes that solve fair exchange and not distributed consensus, and vice versa. Remarkably, this is in contrast to the folklore belief that solving fair exchange is generally harder than solving distributed consensus.

1 Introduction

Distributed consensus (DC) is an essential building block for fault-tolerant distributed computing (e.g. see [27]). Fair exchange (FE) is a fundamental problem in computer security, upon which various contract signing, certified email, and non-repudiation protocols are built [5, 14, 1, 28]. There are remarkable similarities between these two problems, as observed by Tygar [38] and Pagnia and Gärtner [30]. The goal of this paper is to give a formal comparison between FE and DC in asynchronous multiparty settings. A clear picture of this relation has two immediate benefits: (1) from a practical point of view, this would help in translating efficient solutions of one to the other, as much research has been done independently on these two problems so far, and (2) from a theoretical point of view, this can give us a better understanding of the limits of *solvability* of FE, as solvability for DC is to a great extent well understood.

(Un)solvability criteria for DC, such as [18, 9, 8], virtually define the boundaries of *what is possible* in fault-tolerant distributed computing, and have therefore been subject to intense research. Below, we give a chronological overview on work dealing with solvability of FE (in deterministic systems):

In synchronous systems, Even and Yacobi [15] (1980), and independently Rabin [34] (1981), informally argue that two processes cannot fairly exchange secrets, when one of them is prone to Byzantine failure. This negative result has been later on formalized by DeMilo, Lynch and Merritt [12]. However, in fully connected network topologies, the completeness theorems of Ben-Or, Goldwasser and Wigderson [4] (independently derived in [10]) show that, in the presence of t Byzantine processes, any secure n party computation, including FE and DC, can be solved when $t < \frac{n}{3}$.

In asynchronous systems, Pagnia and Gärtner show that when one process is prone to crash failure, DC between two processes is reducible to FE, which entails that FE is *harder* than DC [30]. This, along with the impossibility of DC in such settings [18], establish the unsolvability of FE. It is worth noticing that

the unsolvability result of [15] (and [34, 12]) is based on the malicious act of withholding (parts of) information, whereas in [30], inability to decide termination in asynchronous systems when processes may crash is used to imply unsolvability of FE. These, thus, establish their results based on orthogonal difficulties in solving FE, and neither of them immediately follows from the other.

A natural question at this point is: Can we conclude that FE is solvable only in the settings in which DC is solvable? In other words, is solving FE harder than solving DC? The answer turns out to be negative, as we contend in this paper.

We show that the reduction of DC to FE in [30] is bound to two processes, and does not hold in general. We prove that in asynchronous systems where participating processes are prone to crash failures, while a majority of the processes are correct, FE is *incomparable* to DC. That is, there exist failure detector classes that solve FE and not DC, and vice versa. This is in contrast to the folklore belief that solving FE is harder than solving DC (e.g. see [30, 16]). To prove this result, we build upon Guerraoui’s work on incomparability of non-blocking atomic commit (NBAC) to DC [25]. As a side note, we describe why in the special case of only two participating processes, FE is indeed harder than DC, hence coming to terms with the previous result of Pagnia and Gärtner [30]. We use an LTL [33] formalization to define the DC, NBAC and FE problems. This removes doubts about what textual requirements for FE, often found in the literature, might mean. The LTL formulas are interpreted in the well-accepted finite state machine model, widely used in the distributed computing literature (e.g. [9]). This neatly places the FE security problem into a framework often used to reason about distributed computing. Other approaches to formal definition of the FE requirements, such as game theory, have been investigated in [32, 30, 31, 6, 22].

Our incomparability result is stated in asynchronous models, where processes may *crash*. The choice of this model has certain implications: (1) It follows that in more sophisticated models where faulty processes are allowed to misbehave beyond benign crashes (such as crash-recovery, or Byzantine failure models), FE and DC remain incomparable. Therefore, our result draws an incomparability line in stronger failure models as well. (2) In practical terms, crash failures are usually considered to be simplistic for design and analysis of security protocols. Although this weakens the relevance of our result to some practical scenarios, we point out that recent advances in trusted computing devices, and their use in security protocols, allow for more restricted hostile environment models. In particular, in section 4, we discuss how our model is related to the existing literature on practical realizations of FE using *guardians* and *TrustedPals*.

Road map We start with introducing the notions and notations used in the paper in section 2. In section 3 we present our incomparability result. Section 4 discusses related work, while section 5 concludes the paper.

2 Preliminaries

We consider the asynchronous message passing model of [18], plus the failure detector abstraction of [9]. A brief description of this model follows.

The system model We consider n processes, $\Omega = \{p_1, \dots, p_n\}$, sitting on the nodes of a fully connected communication graph. No bound is assumed on their clock drifts, or the time needed to complete a local instruction. The bidirectional channels that connect these processes are *asynchronous*, i.e. they guarantee to eventually deliver sent messages, but no time bounds or specific ordering is enforced. The processes are prone to *crash failures*, i.e. a failed process ceases to act from the point of crash onwards. A process that does not crash is *correct*.

We assume a discrete global clock, which is not accessible to the participating processes. The range of the clock, denoted Φ , is the set of natural numbers.

A *failure pattern* is a non-decreasing function $F : \Phi \rightarrow 2^\Omega$, where $F(t)$ contains the set of crashed processes at time t . An *environment* E is a set of failure patterns. Although processes do not have direct access to F , each process has a local *failure detector module* which gives hints about the processes that are *suspected* of crash. Failure detector modules can in general make mistakes. A *failure detector* is intuitively such a distributed oracle. A *failure detector history* is a function $H : \Omega \times \Phi \rightarrow 2^\Omega$. Intuitively, $H(p, t)$ characterizes the output of the failure detector module hosted in process p , at time t . A failure detector class \mathbf{D} is a function that maps any failure pattern to a set of failure detector histories.

An *algorithm* assigns a finite state machine to each $p \in \Omega$. At time $t \in \Phi$, three tasks are performed atomically: (1) p picks a message non-deterministically from the set of buffered incoming messages, or receives the null message λ , (2) p gets the value of $H(p, t)$ from its local failure detector module, (3) p performs a computation based on the values from (1) and (2) and its current state, selects a next state, and sends out a message (possibly none) to another process $q \in \Omega$.

A *configuration* is a pair (P, M) , where P is the Cartesian product of the states of elements of Ω , and M is a multiset of messages, containing the messages buffered for delivery. The *initial* configuration is (P_0, \emptyset) , where P_0 is the Cartesian product of the initial states of the elements of Ω . A *transition* is a triple $\theta = (p_i, m, d)$, where p_i is the process that takes the transition, m is the message that p_i receives, and d is the value that p_i reads off its local failure detector module. Clearly, θ is applicable to (P, M) only if $m \in M \cup \{\lambda\}$. The unique configuration resulted is denoted $\theta((P, M))$.

The *Kripke structure* resulting from the asynchronous interleaving of the transitions of processes in Ω , is a triple (C, \hat{c}_0, R) , where C is a countable set of configurations, $\hat{c}_0 = (P_0, \emptyset)$, and $R \subseteq C \times C$ is defined as: $(c, c') \in R$ iff $c' = \theta(c)$, for an applicable θ . $c \in C$ is *reachable* iff $(\hat{c}_0, c) \in R^*$, the reflexive transitive closure of R . We only consider reachable configurations. A *trace* in $K = (C, \hat{c}_0, R)$ is an infinite sequence of configurations $\gamma = c_0, c_1, \dots$, such that $\forall i \geq 0. (c_i, c_{i+1}) \in R$. We say γ is *rooted* iff $c_0 = \hat{c}_0$. When $T = t_0, t_1, \dots$ is an increasing sequence of natural numbers and γ is rooted, $\sigma = (\gamma, T)$ is an *execution* in K . For $\sigma = (\gamma, T)$, we abuse the notation and write $F(\sigma) = \cup_{t \in T} F(t)$.

The LTL logic We use a subset of LTL [33] for describing properties of Kripke structures. For a finite set of atomic propositions AP ,

- If $a \in AP$, then a is an LTL formula.

- If ϕ and ϕ' are LTL formulas, then so are $\neg\phi$, $\phi \vee \phi'$ and $\diamond\phi$.

As shorthands, we write $\phi \wedge \phi'$ for $\neg(\neg\phi \vee \neg\phi')$, and $\square\phi$ for $\neg\diamond(\neg\phi)$.

Given a Kripke structure $K = (C, \hat{c}_0, R)$, a *labeling* function $[\cdot] : C \rightarrow 2^{AP}$ assigns propositions to the configurations. Intuitively, for a configuration c , the set $[c]$ is the set of atomic propositions (elements of AP) true at c . For example, $(y_p \neq y_q) \in [c]$ iff the value of y_p is different from y_q at configuration c . Other propositions are likewise interpreted in the natural way.

Kripke structures are models of LTL formulas. We say K satisfies formula ϕ iff for all rooted traces γ in K , $\gamma \models \phi$. For γ being a trace in K , and ϕ an LTL formula, the relation $\gamma \models \phi$ is defined below. We let $\gamma = c_0, \dots, c_n, c_{n+1}, \dots$, and write $\gamma^n = c_n, c_{n+1}, \dots$

- $\gamma \models a$, with $a \in AP$, iff $a \in [c_0]$.
- $\gamma \models \neg\phi$, iff $\neg(\gamma \models \phi)$.
- $\gamma \models \phi \vee \phi'$, iff $\gamma \models \phi$ or $\gamma \models \phi'$.
- $\gamma \models \diamond(\phi)$, iff $\exists n \geq 0. \gamma^n \models \phi$.

Intuitively, \diamond expresses eventual reachability, while \square is used for invariants.

Problem definitions We now specify the requirements of the DC, NBAC and FE problems using LTL formulas. As a syntactic shorthand, we use quantifiers inside the formulas. Since Ω is finite, existential (\exists) and universal (\forall) quantifiers over elements of Ω can be rewritten into a finite number of disjunctions and conjunctions, respectively. Therefore, these formulas remain inside LTL.

Definition 1 (DC). Consider a set of items $V = \{0, 1\}$. Each process $p \in \Omega$ starts with a pair (x_p, y_p) , where $x_p \in V$ is its input item, and $y_p \in V \cup \{b\}$ is a write-once buffer containing the process's output item. We assume $\forall p \in \Omega. y_p = b$ at the initial state. An algorithm is said to solve DC using a failure detector class \mathbf{D} for environment E , iff, for any crash failure pattern $F \in E$ and any $H \in \mathbf{D}(F)$, every rooted execution $\sigma = (\gamma, T)$ in the Kripke structure resulting from the algorithm satisfies the following properties:

- *Termination:* $\forall p \notin F(\sigma). \diamond(y_p \neq b)$.³
- *Agreement:* $\forall p, q \notin F(\sigma). \square((y_p \neq b \wedge y_q \neq b) \Rightarrow y_p = y_q)$.
- *Validity:* $\forall p \notin F(\sigma). \square(y_p = b \vee (\exists q \in \Omega. y_p = x_q))$.⁴

Definition 2 (NBAC). Consider a set of items $V = \{0, 1\}$. Each process $p \in \Omega$ starts with a pair (x_p, y_p) , where $x_p \in V$ is its input item (also called vote), and $y_p \in V \cup \{b\}$ is a write-once buffer containing the process's output item. We assume $\forall p \in \Omega. y_p = b$ at the initial state. An algorithm is said to solve NBAC using a failure detector class \mathbf{D} for environment E , iff, for any crash failure pattern $F \in E$ and any $H \in \mathbf{D}(F)$, every rooted execution $\sigma = (\gamma, T)$ in the Kripke structure resulting from the algorithm satisfies the following properties:

³ At configuration c , $y_p \neq b$ holds (i.e. $(y_p \neq b) \in [c]$) iff process p has assigned a value to its local y_p that is different from b , and similarly for other propositions.

⁴ This variant of the validity condition is sometimes called *uniform validity* [9].

- *Termination*: $\forall p \notin F(\sigma). \diamond(y_p \neq b)$.
- *Agreement*: $\forall p, q \notin F(\sigma). \square((y_p \neq b \wedge y_q \neq b) \Rightarrow y_p = y_q)$.
- *A-validity*: $(\exists q \in \Omega. x_q = 0) \Rightarrow \forall p \notin F(\sigma). \square(y_p \neq b \Rightarrow y_p = 0)$.
- *C-validity*: $(F(\sigma) = \emptyset \wedge \forall q \in \Omega. x_q = 1) \Rightarrow \forall p \in \Omega. \square(y_p \neq b \Rightarrow y_p = 1)$.

Intuitively, in NBAC processes aim for output 1. Each process has however a right to veto the outcome, by voting 0. We note that when every process votes 1, and then one crashes, A-validity and C-validity enforce no particular outcome for the NBAC problem. Thus, in such cases, both 0 and 1 are legitimate outcomes.

Now, we turn to the requirements of FE. Fair exchange aims at exchanging items in a *fair* manner. Informally, fair means that either all the participants receive a desired item in exchange for their own, or none of them does so. In the literature, there are various definitions for multiparty FE, depending on which topology is chosen, whether one unit or more are exchanged, etc., see [21, 29].

Below, we focus on *ring exchange patterns* [21], where processes are sitting in a ring and each process p receives its desired item from its predecessor and sends its item to its successor. The underlying communication network is nonetheless a fully connected graph. Since any permutation can be decomposed into disjoint cycles [21], this pattern can capture any exchange situation in which each process has one unit of item to offer and expects one unit of item in exchange.

The ‘+’ and ‘-’ operators on indexes of Ω are calculated modulo n , the size of Ω . We confine the items exchanged to single bits. Our results can naturally be extended to the case in which arbitrary strings of bits are subject to exchange.

Definition 3 (FE). *Consider a set of items $V = \{0, 1\}$. Each process $p \in \Omega$ starts with a pair (x_p, y_p) , where $x_p \in V$ is its input item, and $y_p \in V \cup \{b, \perp\}$ is a write-once buffer containing the process’s output item. We assume $\forall p \in \Omega. y_p = b$ at the initial state. An algorithm is said to solve FE using a failure detector class \mathbf{D} for environment E , iff, for any crash failure pattern $F \in E$ and any $H \in \mathbf{D}(F)$, every rooted execution $\sigma = (\gamma, T)$ in the Kripke structure resulting from the algorithm satisfies the following properties:*

- *Soundness*: $\forall p \notin F(\sigma). \square(y_p \neq b \Rightarrow (y_p = \perp \vee y_p = x_{p-1}))$.
- *Timeliness*: $\forall p \notin F(\sigma). \diamond(y_p \neq b)$.
- *Effectiveness*: $F(\sigma) = \emptyset \Rightarrow \forall p \in \Omega. \square(y_p \neq b \Rightarrow y_p = x_{p-1})$.
- *Fairness*: $\forall p \notin F(\sigma). \square((y_{p+1} = x_p \wedge y_p \neq b) \Rightarrow y_p = x_{p-1})$.
- *Consistency*: $\forall p, q \notin F(\sigma). \square((y_p = \perp \wedge y_q \neq b) \Rightarrow y_q = \perp)$.

Intuitively, \perp is the mark of *unsuccessful* exchanges. The soundness requirement enforces that all unsuccessful correct processes assign \perp to their output buffer. Since we are in the crash failure model, when p receives an item from $p - 1$, it can be sure that the received item is indeed x_{p-1} . In the presence of Byzantine failures, in order to recognize the right item, p is usually assumed to have a *description* of x_{p-1} , which characterizes x_{p-1} “with enough precision” [1].

Timeliness forces termination of non-faulty processes. Effectiveness is a sanity check, ensuring that if every process is correct, the exchanges are indeed successful. A weaker variant of effectiveness is used in [7], where if $F(\sigma) = \emptyset$,

then only one (as opposed to all) execution of the algorithm is required to satisfy $\forall p \in \Omega. \Box(y_p \neq b \Rightarrow y_p = x_{p-1})$. Fairness states that if a non-faulty process reveals its item to its successor, it will certainly receive the item of its predecessor. The consistency condition guarantees that either all the non-faulty processes terminate successfully, or none of them do so. In some applications, the consistency requirement is deemed unnecessary for FE, e.g. fair certified email protocols typically allow some of the correct processes being excluded from the exchange, while the rest exchange their items, cf. [23]. However, some protocols, such as fair contract signing protocols, explicitly rely on consistency, cf. [24].

Note that if a crash occurs (i.e. $F(\sigma) \neq \emptyset$), the FE requirements do not enforce the exchange to be unsuccessful (or to be successful). This is because crashes may occur after faulty processes have sent out their items to the corresponding processes, thus potentially allowing correct process to terminate successfully.

Remark 1. If a process p crashes after it has completed its role in an execution σ , i.e. it has assigned a value to y_p , formally we have $p \in F(\sigma)$. It is however unreasonable to allow the behavior of p after it has left the protocol to affect the algorithm (and the requirements of the problem). Therefore for any execution σ , if p assigns a value to y_p at time t and $p \notin F(t)$, we assume $p \notin F(\sigma)$.

Solvability and comparability We say that a failure detector class \mathbf{D} solves problem B in environment E , iff there exists an algorithm using \mathbf{D} and its execution in E results in a Kripke structure that satisfies B 's specification. A problem B_1 is *harder* than problem B_2 for E , denoted by $E \vdash B_2 \rightarrow B_1$, iff any failure detector class \mathbf{D} that solves B_1 in E , also solves B_2 in E .⁵ When $\neg(E \vdash B_1 \rightarrow B_2)$ and $\neg(E \vdash B_2 \rightarrow B_1)$, we say B_1 and B_2 are *E-incomparable*. Problems B_1 and B_2 are *incomparable* if they are *E-incomparable* for some environment E . When $E \vdash B_1 \rightarrow B_2$ and $E \vdash B_2 \rightarrow B_1$, B_1 and B_2 are *E-equivalent*.

A failure detector class \mathbf{D}_1 is said to be *weaker* than failure detector class \mathbf{D}_2 in environment E , denoted $E \vdash \mathbf{D}_1 \preceq \mathbf{D}_2$, iff there exists a distributed algorithm that, given the information provided by \mathbf{D}_2 , can emulate \mathbf{D}_1 in E .

Failure detector classes $\diamond\mathcal{S}$, \mathcal{P} and \mathcal{B} Of particular interest in this paper are the *eventually strong* failure detector class $\diamond\mathcal{S}$, the *perfect* failure detector class \mathcal{P} and the *stillborn* failure detector class \mathcal{B} . The brief description below is mainly borrowed from [37]. For extensive discussions we refer to [9, 37, 25].

A failure detector class \mathbf{D} is *strongly complete* if every crashed process is eventually suspected by (failure detector modules at) every correct process, that is for each execution $\sigma = (\gamma, T)$, we have $\exists t. \forall t' \geq t. \forall p \in F(\sigma), q \notin F(\sigma). p \in H(q, t')$. The class \mathbf{D} is *strongly accurate* if no process is ever suspected if it has not crashed. More precisely, if for each execution $\sigma = (\gamma, T)$, we have $\forall t. \forall p, q \notin F(t). p \notin H(q, t)$. The class \mathbf{D} is *eventually weakly accurate* if there exist a time and a correct process that is not suspected after that time. More precisely, if for each execution $\sigma = (\gamma, T)$, we have $\exists t. \exists p \notin F(\sigma). \forall t' \geq t. \forall q \notin F(\sigma). p \notin H(q, t')$.

⁵ To be precise, *harder* here stands for *at least as hard as*.

Eventually strong detectors ($\diamond\mathcal{S}$) are the class of strongly complete and eventually weakly accurate failure detectors. Perfect detectors (\mathcal{P}) are the class of strongly complete and strongly accurate failure detectors. The stillborn detectors (\mathcal{B}), originally introduced in [25], behave as perfect detectors \mathcal{P} if no process initially crashes, i.e. $F(0) = \emptyset$. However, if $F(0) \neq \emptyset$, then every failure detector module at every correct process permanently “suspects” its own host process. More precisely, we have $\forall t. \forall p \notin F(t). H(p, t) = \{p\}$. This is a way to inform the correct processes that *some other* process has initially crashed.

From a practical point of view, as noted in [25], \mathcal{B} and \mathcal{P} effectively require the same underlying synchronization mechanism. However, \mathcal{B} is formally weaker than \mathcal{P} , and can serve as a technical means to differentiate between synchrony requirements of FE and DC, as shown below.

3 FE and DC are incomparable

We show that DC and FE are incomparable in asynchronous environments, where a majority of processes are correct (that is, $|F(\sigma)| < \frac{n}{2}$ for each execution σ), while at least two processes can crash. Below, the symbol \mathbf{E} is fixed to refer to such an environment.⁶

Theorem 1. *FE is incomparable to DC.*

Proof. To prove this theorem, it is shown that DC and FE are \mathbf{E} -incomparable. We follow the proof technique of [25]. The proof consists of two parts:

1. To establish $\neg(\mathbf{E} \vdash \text{FE} \rightarrow \text{DC})$, it is enough to prove that the failure detector class $\diamond\mathcal{S}$ does not solve FE in \mathbf{E} , as lemma 1 below shows. It is a well-known result that $\diamond\mathcal{S}$ does solve DC in \mathbf{E} , see [9].
2. To establish $\neg(\mathbf{E} \vdash \text{DC} \rightarrow \text{FE})$, we make use of the stillborn failure detector \mathcal{B} . Lemma 2 below proves that \mathcal{B} does solve FE. As shown in [25] (lemma 3.4), $\neg(\mathbf{E} \vdash \diamond\mathcal{S} \preceq \mathcal{B})$. Since $\diamond\mathcal{S}$ is the weakest failure detector that solves DC [8], i.e. $\diamond\mathcal{S}$ can be emulated by any failure detector that solves DC, hence \mathcal{B} does not solve DC in \mathbf{E} .

This completes the proof. □

Lemma 1. *FE is not solvable using $\diamond\mathcal{S}$ in \mathbf{E} .*

Proof. We prove this lemma by showing that $\mathbf{E} \vdash \text{NBAC} \rightarrow \text{FE}$. As NBAC cannot be solved in \mathbf{E} using $\diamond\mathcal{S}$ (see [25], lemma 3.1), it follows that FE can also not be solved using $\diamond\mathcal{S}$ in \mathbf{E} .

Algorithm 1 (specified for process p , which as input receives (x_p, y_p) , where x_p is the initial vote of the process) solves NBAC, given that a black-box procedure to solve FE is available to the processes. Translating this pseudo-code to a finite state machine (cf. section 2) is straightforward. The proposed construct asserts $\mathbf{E} \vdash \text{NBAC} \rightarrow \text{FE}$.

⁶ The condition that at least two processes can crash in \mathbf{E} is required to ensure the validity of $\neg(\mathbf{E} \vdash \diamond\mathcal{S} \preceq \mathcal{B})$, needed in the proof of theorem 1.

Algorithm 1 $E \vdash \text{NBAC} \rightarrow \text{FE}$

```
1: let  $i_p := x_p$ ;  
2: for  $cntr_p := 1$  to  $n - 1$  do  
3:   let  $o_p := b$ ;  
4:    $\text{FE}(i_p, o_p)$ ;  
5:   if  $o_p = \perp$  then  
6:     let  $y_p := 0$ ;  
7:     quit;  
8:   else if  $o_p \neq \perp$  then  
9:     let  $i_p := i_p \times o_p$ ;  
10:  end if  
11: end for  
12: let  $y_p := i_p$ ;  
13: quit;
```

Intuitive description of algorithm 1: We think of the processes who want to perform NBAC as being placed on a ring (conforming to the ring exchange pattern of the FE procedure available to the processes). The algorithm consists of $n - 1$ rounds, where $n = |\Omega|$. In each round, each process receives the vote of its predecessor, and sends its vote to its successor, both using the FE procedure available to it. Each process updates its vote to the product of its vote and the value it receives from its predecessor (here, we could in effect use the *min* function instead of product).

If no failure occurs, after $n - 1$ rounds, the initial vote of each process is propagated through the entire ring. Finally, for each p , y_p is assigned with $\prod_{q \in \Omega} x_q = x_{p_1} \times \dots \times x_{p_n}$. If $\forall q \in \Omega. x_q = 1$, then each y_p is assigned with 1. However, if there exists a process whose initial vote is 0, then $\prod_{q \in \Omega} x_q = 0$, resulting in $y_p = 0$ for all $p \in \Omega$.

Correctness of algorithm 1:

- (termination) The termination of this algorithm relies on the timeliness property of FE. Note that the only possible blocking point in the code is the call to FE; the rest of the code is executed purely locally. However, from timeliness of FE (see definition 3) we know that $\forall p \notin F(\sigma). \diamond(o_p \neq b)$, i.e. in each FE call, any correct process p eventually assigns a value to o_p .
- (agreement) Let $\sigma = (\gamma, T)$ be an execution of the algorithm. We distinguish two possibilities for σ : (1) $F(\sigma) = \emptyset$, (2) $F(\sigma) \neq \emptyset$. In case (1), due to effectiveness of FE, all the processes assign $y_p := \prod_{q \in \Omega} x_q$, thus $y_p = y_q$ for any two processes p and q for which $y_p \neq b$ and $y_q \neq b$. Agreement is therefore satisfied in this case.

In case (2), let us assume the first crash happens at time t . We distinguish three cases for t : (a) t is before the last (i.e. the $n - 1^{\text{th}}$) call to FE occurs, (b) t is placed in the time interval in which the last call to FE has started, but has not finishes yet, (c) the last call to FE has completed before t .

In case (a), observe that all correct processes will assign $y_p := 0$. This is because of fairness and consistency of FE that any $p \notin F(\sigma)$ will receive

$o_p = \perp$ in its next FE call, which is definitely forthcoming. In case (b), both outcomes $o_p = \perp$ and $o_p \neq \perp$ are possible. However, the outcome would in any event be consistent, due to consistency of FE. Therefore, in both these situations, $y_p = y_q$ for any two processes $p, q \notin F(\sigma)$ for which $y_p \neq b$ and $y_q \neq b$. In case (c), all the correct processes will assign $y_p := \prod_{q \in \Omega} x_q$, hence meeting agreement. This is due to effectiveness of FE that any process receives $o_p \neq \perp$. We remark that a crash after the last call to FE is not observed by the FE procedure (cf. remark 1). Agreement is thus satisfied in case (2) as well.

- (A-validity) Consider an execution in which no process crashes and a correct process votes 0. Then, due to effectiveness of FE, clearly $\prod_{q \in \Omega} x_q = 0$, hence follows $\Box(y_p \neq b \Rightarrow y_p = 0)$ for any p . This proves A-validity. For the case (at least) a process crashes, we reuse the proof of the agreement property above. We note that for correct processes two outcomes are possible: (1) all the correct processes assign $y_p := 0$, thus meeting A-validity, or (2) all the correct processes assign $y_p := \prod_{q \in \Omega} x_q$. In the latter case, if there is a process q with $x_q = 0$, then clearly y_p is assigned with 0 for all correct p , thus satisfying A-validity. If there is no process who has voted 0 initially, then the antecedent of the A-validity condition (i.e. $\exists q \in \Omega. x_q = 0$) is false. A-validity holds for such an execution automatically.
- (C-validity) To check C-validity we only need to consider executions in which $F(\sigma) = \emptyset \wedge \forall q \in \Omega. x_q = 1$. Observe that if no process crashes and $\forall q \in \Omega. x_q = 1$, then $\prod_{q \in \Omega} x_q = 1$ due to effectiveness of FE, thus $\Box(y_p \neq b \Rightarrow y_p = 1)$ for any p . The C-validity condition follows from this.

This completes the proof of the correctness of algorithm 1. □

Below, we use $\text{send}_{\rightarrow p}(m)$ and $\text{recv}_{\leftarrow p}(v)$ actions to indicate sending message m to process p and receiving a message from process p and assigning the local variable v with the received content, respectively. Since there is a designated communication channel between every two processes (recall that the communication graphs are fully connected, section 2), no confusion may arise regarding the source or destination of the messages exchanged using these actions.

Lemma 2. *FE is solvable using \mathcal{B} in \mathbf{E} .*

Proof. We prove this lemma by showing that $\mathbf{E} \vdash \text{FE} \rightarrow \text{NBAC}$.⁷ As NBAC can be solved in \mathbf{E} using \mathcal{B} (see [35], and lemma 3.3 in [25]), it follows that FE can also be solved in \mathbf{E} using \mathcal{B} .

To solve FE, algorithm 2 is executed by each process $p \in \Omega$. Note that this algorithm assumes access to a black-box procedure for solving NBAC. Translating this pseudo-code to a finite state machine (cf. section 2) is straightforward.

⁷ A reduction of FE to the *biased consensus* problem is given in [2], for synchronous systems. We note that this reduction cannot be used here, because of asynchrony in our model. In particular, recv actions may see only empty messages for an arbitrary (though not infinite) number of times.

Algorithm 2 $E \vdash FE \rightarrow NBAC$

```
1: send $\rightarrow_{p+1}$ ( $x_p$ );
2: let  $w_p := 1$ ; let  $v_p := \lambda$ ;
3: repeat
4:   recv $\leftarrow_{p-1}$ ( $v_p$ );
5:   if  $v_p = \lambda$  then
6:     let  $i_p := 0$ ;
7:   else
8:     let  $i_p := 1$ ;
9:   end if
10:  if local time out is reached then
11:    let  $w_p := 0$ ;
12:  end if
13: until  $w_p = 0 \vee i_p = 1$ 
14: let  $z_p := b$ ;
15: NBAC( $i_p, z_p$ );
16: if  $z_p = 1$  then
17:   let  $y_p := v_p$ ;
18: else
19:   let  $y_p := \perp$ ; % in case  $z_p = 0$ 
20: end if
21: quit;
```

Intuitive description of algorithm 2: Any correct process sends its item to its successor, and waits to receive the item of its predecessor. If a process p does not receive its desired item within a certain time interval, locally specified by p itself, it will time out and stop waiting, reflected in the code by letting $w_p := 0$ (over asynchronous channels, messages may be delivered with an arbitrary finite delay). Any correct process, therefore, will eventually exit its **repeat** loop.

The call to the NBAC procedure, available to the processes, is meant to ensure that if a correct process p has not received x_{p-1} , then every correct process q will respect fairness and consistency, and set $y_q := \perp$.

Notice that it cannot occur that a correct process p assigns λ to y_p . This is because if “let $y_p := v_p$;” (line 17 of the code) is executed by p , then p must have set $i_p := 1$, due to A-validity of NBAC. This, in turn, implies $v_p \neq \lambda$.

If no failure or time-out occurs, eventually all the items reach their destinations, and only then does the call to NBAC return $z_p := 1$ for all $p \in \Omega$. This lets all the participants assign the received items to their output buffers, and quit.

Correctness of algorithm 2: Below, we argue for the correctness criteria from the view point of a correct process, called p . The arguments can naturally be used for other correct processes as well.

- (soundness) Since y_p is assigned only with either of \perp or v_p , which contains x_{p-1} , the soundness requirement of FE is met. We emphasize that in the crash failure model, the value sent by $p - 1$ to p , if it ever arrives, is x_{p-1} . This is simply because a faulty process in this model, by definition, does not tamper with data.

- (timeliness) Timeliness for p hinges on the termination condition for NBAC. Note that except for the NBAC call, all the actions (including setting a definite time out) performed by p are local. From definition 2, we know that $\diamond(z_p \neq b)$, i.e. p will eventually receive a value for z_p . Process p would then assign the proper value to y_p , and quit the exchange. Therefore, $\gamma \models \diamond(y_p \neq b)$ in all $\sigma = (\gamma, T)$, where $p \notin F(\sigma)$ (recall that $F(\sigma) \in \mathbf{E}$).
- (effectiveness) We only need to consider executions σ for which $F(\sigma) = \emptyset$. When no process crashes and messages are delivered in a timely manner ⁸, there is a point in time at which all the correct processes have received their desired items. Then, any such correct process p calls the NBAC procedure by letting $i_p := 1$. Therefore, we have $\forall q \in \Omega. i_q = 1$. This, and $F(\sigma) = \emptyset$, according to C-validity of NBAC, guarantee $z_p \neq b \Rightarrow z_p = 1$, for all $p \in \Omega$. Since y_p will be assigned with v_p in this case, we have $\forall p \in \Omega. \Box(y_p \neq b \Rightarrow y_p = x_{p-1})$.
- (fairness) To check fairness for process p , we need to consider only executions $\sigma = (\gamma, T)$ in which $p \notin F(\sigma)$. Now consider any configuration c on γ such that $(y_{p+1} = x_p) \in [c]$ (i.e. process $p + 1$ has assigned x_p to y_{p+1} in configuration c). According to remark 1 we get $p + 1 \notin F(\sigma)$. Two cases are then possible:
 1. $(y_p \neq b) \in [c]$: This is the case in which p has assigned some value to y_p . According to soundness (see above), we have $(y_p = \perp) \in [c]$ or $(y_p = x_{p-1}) \in [c]$. The latter situation would immediately imply fairness. However, the situation $(y_p = \perp) \in [c]$ can only happen if $(z_p = 0) \in [c]$. According to agreement of NBAC, this implies $(z_{p+1} = 0) \in [c]$. Consulting algorithm 2, this shows that process $p+1$ should have assigned $y_{p+1} := \perp$, contradicting the assumption that $y_{p+1} = x_p$. Hence, the situation $(y_p = \perp) \in [c]$ cannot happen.
 2. $(y_p = b) \in [c]$: This is the case p has not yet assigned any value to y_p . We split this into two cases: (a) p has received x_{p-1} , (b) p has not received x_{p-1} . Note that since $p + 1$ has $y_{p+1} = x_p$, NBAC(i_p, z_p) should have returned $z_p = 1$ (a result of the agreement property of NBAC). Now, in case (a), p can (and according to timeliness will eventually do) assign $y_p := v_p$, where $v_p = x_{p-1}$. This shows that $\gamma \models \Box(y_p \neq b \Rightarrow y_p = x_{p-1})$, hence attaining fairness. In case (b), since p has not received x_{p-1} , clearly p has set $i_p := 0$ in its call to NBAC. Since $p + 1 \notin F(\sigma)$, according to A-validity of NBAC, $\gamma \models \Box(z_{p+1} \neq b \Rightarrow z_{p+1} = 0)$. This contradicts $(y_{p+1} = x_p) \in [c]$, as $z_{p+1} = 0$ enforces $y_{p+1} = \perp$ in algorithm 2.

This shows that the algorithm achieves fairness.

⁸ Although needed in this step of our proof, we note that effectiveness in definition 3 places no conditions on processes not timing out, or messages being delivered timely. In general, if processes want to abandon the exchange, e.g. by early time-outs, no protocol can achieve its goals. This is indeed reflected in the definition of effectiveness given by Asokan (see page 9 in [1]). We feel putting the timeout condition in the formal definition of FE would, however, unnecessarily clutter the presentation, and make the definition rather low-level.

- (consistency) To check consistency, we confine to executions $\sigma = (\gamma, T)$ in which $p, q \notin F(\sigma)$, and $y_p \neq b, y_q \neq b$. Towards a contradiction, assume there is a configuration c on γ , in which $(y_p = \perp) \in [c]$ and $(y_q \neq \perp) \in [c]$. According to soundness, $(y_q = x_{q-1}) \in [c]$. Therefore, q should have received $z_q = 1$ in its NBAC call. The agreement property of NBAC, nevertheless, states that $z_p = 1$ as well. Now, we distinguish two cases: (1) If $(i_p = 0) \in [c]$, then the A-validity condition of NBAC has been violated by returning $z_p = 1$, thus reaching a contradiction. (2) If $(i_p = 1) \in [c]$, then p must have received x_{p-1} . From the argument above, we know $(z_p = 1) \in [c]$. Therefore, p would, according to algorithm 2, set $y_p := x_{p-1}$ in this situation, contradicting the assumption that $(y_p = \perp) \in [c]$. This shows no such configuration c can belong to γ .

This completes the proof of the correctness of algorithm 2. □

Remark 2. Suppose some process q crashes in algorithm 2, while all other processes are correct. If q crashes *after* sending its item to $q+1$, the item may arrive at $q+1$ in time. We note that even if everything goes well with all the other processes in this scenario (i.e. they all vote 1 by letting $i_p := 1$), the outcome of the NBAC call is not guaranteed to be 1 nor 0, because q has crashed, no correct process has voted 0, while q could have voted 0 or 1. This may (wrongly) seem to violate effectiveness of FE, since despite all the correct processes receiving their desired items, the exchange may still terminate unsuccessfully (in case NBAC returns $z_{q+1} = 0$). We remark that because $F(\sigma) \neq \emptyset$ in this scenario, the effectiveness condition does not enforce a successful exchange.

The following corollary is straightforward, hence we omit the proof.

Corollary 1. *FE and NBAC are E-equivalent.*

The connection between FE and NBAC has been noticed in previous studies, such as [38] and [26]. These however rely only upon informal arguments.

4 Related work

In this section we discuss how our incomparability result relates to the existing literature on FE and DC.

The incomparability of FE to DC (theorem 1) may wrongly seem to contradict the result of [30], where it is shown that in any asynchronous system of two processes, while one process is prone to crash failure, FE is harder than DC (that is $DC \rightarrow FE$). The fact that theorem 1 assumes a majority of correct processes should clarify this discrepancy from a technical point of view.

To give an intuitive reason for this distinction, let us proceed with considering a scenario in which one process crashes, and then all the correct processes, using failure detectors, learn that this process has crashed. In FE, these correct processes can all safely return \perp and quit the exchange, while in DC, they still need to reach a consensus on what value they would return (they cannot all return 0,

because it would violate the validity condition if they all had proposed 1, and so forth). An interesting special case is when only two processes are engaged in the protocol, while one process is prone to crash: The aforementioned difficulty in DC does not arise, as *the* correct process can simply output its own input value. This is exactly why the reduction given in [30] works for two processes (and not more), when one process is prone to the crash failure.

The fair exchange problem in the security literature is usually studied in the Dolev-Yao hostile environment model [13]. In this model, the attacker is a (cryptographically bounded) Byzantine process, sitting in the center of a star-like network. All the correct processes therefore communicate *via* the attacker. The network connectivity⁹ in such graphs is therefore 1. This implies DC and FE are not solvable in such environment, as the attacker can simply drop all the transmitted messages, preventing termination, cf. [36].¹⁰

A conventional way to circumvent this difficulty in deterministic systems¹¹ is to assume trusted parties which are connected to other processes via *resilient channels*. A resilient channel guarantees to eventually deliver all the messages submitted through it. Note that adding resilient channels to the system weakens the Dolev-Yao model, in the sense that the attacker node cannot indefinitely delay (or completely suppress) certain messages. For more on this approach see, e.g., [1].

Another way to weaken the Dolev-Yao intruder is via introducing trusted computing devices at each user node, in contrast to a central trustee. These devices can then be used to perform distributed tasks, such as DC or FE. Although these trusted devices are operated by non-trusted malicious entities, it is assumed that they can establish secure channels between themselves, e.g. using encryption. In case the trusted devices are stateful, once the attacker drops a message destined to one of them, the device will behave as if it has crashed (i.e. does not receive or send any message or do any internal computation, unless that particular message arrives). The model used in this paper potentially represents such environments. Practical implementations of such distributed systems have recently attracted much interest; for instance see the literature on using *guardians* [3, 2] and *TrustedPals* [19, 11] to realize FE.

5 Concluding remarks

In this paper, we establish that solving the fair exchange problem is in general not harder than reaching consensus in a distributed system. The model used for proving this result consists of processes connected via asynchronous reliable

⁹ A network has connectivity c iff at least c nodes need to be removed to disconnect the network.

¹⁰ To solve DC in synchronous systems in presence of f Byzantine processes, the network connectivity needs to be at least $2f + 1$, e.g. see [17].

¹¹ Probabilistic protocols were indeed among the first solutions proposed for the FE problem, e.g. see [5, 14]. These are however beyond the scope of this paper; the reader is instead referred to [20] for a comprehensive survey.

channels, while processes are assumed to be prone to crash failures. As a side result, it is also shown that in this model, fair exchange is equivalent to the non-blocking atomic commit problem, i.e. any failure detector class that solves fair exchange, also solves non-blocking atomic commit in this model, and vice versa.

As future research, it must be interesting to explore the practical implications of our incomparability result for solving FE. A related uninvestigated question pertains to the existence of models in which fair exchange can be differentiated from non-blocking atomic commit.

Acknowledgment We are grateful to Wan Fokkink and Felix Freiling for many helpful discussions. M. Torabi Dashti was partially supported by the FP7-ICT-2007-1 Project no. 216471, “AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures”.

References

1. N. Asokan. *Fairness in electronic commerce*. PhD thesis, University of Waterloo, Canada, 1998.
2. G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolić. Gracefully degrading fair exchange with security modules. In *EDCC '05*, volume 3463 of *LNCS*, 2005.
3. G. Avoine and S. Vaudenay. Optimal fair exchange with guardian angels. In *WISA '03*, volume 2908 of *LNCS*, 2003.
4. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88*, pages 1–10. ACM Press, 1988.
5. M. Blum. Three applications of the oblivious transfer: Part I: Coin flipping by the telephone; part II: How to exchange secrets; part III: How to send certified electronic mail. Technical report, Dept. EECS, UC Berkeley, 1981.
6. L. Buttyán, J. Hubaux, and S. Capkun. A formal model of rational exchange and its application to the analysis of Syverson’s protocol. *Journal of Computer Security*, 12(3-4):551–587, 2004.
7. R. Chadha, M. Kanovich, and A. Scedrov. Inductive methods and contract-signing protocols. In *CCS '01*, pages 176–185. ACM Press, 2001.
8. T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
9. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
10. D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *STOC '88*, pages 11–19. ACM Press, 1988.
11. R. Cortiñas, F. Freiling, M. Ghajar-Azadanlou, A. Lafuente, M. Larrea, L. Draque Penso, and I. Soraluze Arriola. Secure failure detection in TrustedPals. In *SSS '07*, volume 4838 of *LNCS*, 2007.
12. R. DeMillo, N. Lynch, and M. Merritt. Cryptographic protocols. In *STOC '82*, pages 383–400. ACM Press, 1982.
13. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, IT-29(2):198–208, 1983.

14. S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.
15. S. Even and Y. Yacobi. Relations among public key signature systems. Technical Report 175, Computer Science Dept., Technion, Haifa, Isreal, March 1980.
16. P. Ezhilchelvan and S. Shrivastava. A family of trusted third party based fair-exchange protocols. *IEEE Trans. Dependable and Secure Computing*, 2(4), 2005.
17. M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distrib. Comput.*, 1(1):26–39, 1986.
18. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
19. M. Fort, F. Freiling, L. Draque Penso, Z. Benenson, and D. Kesdogan. TrustedPals: Secure multiparty computation implemented with smart cards. In *ESORICS '06*, volume 4189 of *LNCS*, 2006.
20. M. Franklin, Z. Galil, and M. Yung. An overview of secure distributed computing. Technical Report TR CUCS-008-92, Columbia University, March 1992.
21. M. Franklin and G. Tsudik. Secure group barter: Multi-party fair exchange with semi-trusted neutral parties. In *FC '98*, volume 1465 of *LNCS*, 1998.
22. B. Garbinato and I. Riekebusch. A topological condition for solving fair exchange in byzantine environments. In *ICICS '06*, volume 4307 of *LNCS*, 2006.
23. J. Gomila, M. Capellà, and L. Rotger. A realistic protocol for multi-party certified electronic mail. In *ISC '02*, volume 2433 of *LNCS*, 2002.
24. N. González-Deleito and O. Markowitch. Exclusion-freeness in multi-party exchange protocols. In *ISC '02*, volume 2433 of *LNCS*, 2002.
25. R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.
26. P. Liu, P. Ning, and S. Jajodia. Avoiding loss of fairness owing to failures in fair data exchange systems. *Decision Support Systems*, 31(3):337–350, 2001.
27. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
28. S. Micali. Simple and fast optimistic protocols for fair electronic exchange. In *PODC '03*, pages 12–19, New York, NY, USA, 2003. ACM Press.
29. A. Mukhamedov, S. Kremer, and E. Ritter. Analysis of a multi-party fair exchange protocol and formal proof of correctness in the strand space model. In *FC '05*, volume 3570 of *LNCS*, 2005.
30. H. Pagnia and F. Gärtner. On the impossibility of fair exchange without a trusted third party. Technical Report TUD-BS-1999-02, Department of Computer Science, Darmstadt University of Technology, Germany, March 1999.
31. H. Pagnia, H. Vogt, and F. Gärtner. Fair exchange. *Comput. J.*, 46(1):55–75, 2003.
32. B. Pfitzmann, M. Schuner, and M. Waidner. Optimal efficiency of optimistic contract signing. In *PODC '98*, pages 113–122. ACM Press, 1998.
33. A. Pnueli. The temporal logic of programs. In *FOCS '77*. IEEE, 1977.
34. M. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Harvard University, May 1981.
35. D. Skeen. Nonblocking commit protocols. In *SIGMOD Conference on Management of Data*, pages 133–142. ACM Press, 1981.
36. P. Syverson. A different look at secure distributed computation. In *CSFW '97*, pages 109–115. IEEE CS, 1997.
37. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2nd edition, 2000.
38. J. Tygar. Atomicity in electronic commerce. In *PODC '96*, pages 8–26. ACM press, 1996.