ELSEVIER
Decision Support Systems 892 (2000) xxx
Decision Support Systems
www.elsevier.com/locate/dsw

Avoiding loss of fairness owing to failures in fair data exchange systems

Peng Liu [a,*], Peng Ning [b,1], Sushil Jajodia [b,2]

[a] Department of Information Systems, University of Maryland, Baltimore County, Baltimore, MD 21250, USA
[b] Center for Secure Information Systems, George Mason University Fairfax, VA 22030, USA

Abstract

Fair exchange between mutually distrusted parties has been recognized as an important issue in electronic commerce. However, the correctness (fairness) of the existing fair exchange protocols that use a Trusted Third Party (TTP) is based on the assumption that during an exchange there are no failures at any of the local systems involved in the exchange, which is too strong in many situations. This paper points out that (1) system failures may cause loss of fairness, and (2) most of the existing fair exchange protocols that use a TTP cannot ensure fairness in presence of system failures. This paper presents two categories of techniques, transaction-based approaches and message-logging-based approaches, to help develop data exchange systems that can recover from system failures without losing fairness. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Fair exchange; Fairness; Recoverability; Electronic commerce

UNCORRECTED PROOF

1. Introduction

Experience with electronic commerce has shown that an exchange of one data item for another between mutually distrusted parties is usually the crux of an electronic transaction [6,10,11,18]. A desirable requirement for exchange is fairness. An exchange is fair if at the end of the exchange, either each player receives the item it expects or neither player receives any additional information about the other's item. Fair data exchange has been used in many applications such as non-repudiation of message transmission [18], certified mail [11], contract signing [6], and electronic payment systems [10].

Fair data exchange protocols in the literature can be broken into two categories: third party protocols, which use a trusted third party (TTP) and gradual exchange protocols [6] where the probability of correctness is gradually increased over several rounds of communications. A third party is trusted if it will neither misbehave on its own, nor conspire with either of the players. Gradual exchange protocols do not need a TTP, but they can cause substantial communication and computation overhead. Although as network bandwidth and computation power increase, gradual exchange protocols can expect more real world applications. This paper will focus on third party protocols, which have already been widely used in many applications, especially e-commerce.

* Corresponding author. Tel.: +1-410-455-3268; fax: +1-410-455-1073.

E-mail addresses: pliu@umbc.edu (P. Liu), pning@gmu.edu (P. Ning), jajodia@gmu.edu (S. Jajodia).

[1] Tel.: +1-703-993-1629; fax: +1-703-993-1638.

[2] Tel.: +1-703-993-1653; fax: +1-703-993-1638.

0167-9236/00/$ - see front matter © 2000 Elsevier Science B.V. All rights reserved.
PII: S0167-9236(00)00141-X

Although many practical third party protocols have been proposed [2–4,10,12,18], most of these protocols depend on certain strong assumptions about the communication channels and the local systems. Specifically, most of them assume that during an exchange, no failures will happen at the local systems of either the players or the TTP. Although all of these protocols can achieve certain degree of fairness under these assumptions, most of them cannot ensure fairness in presence of system failures, which will happen from time to time in real computer systems. This implies that most of the existing third party protocols cannot be directly implemented to really achieve fair data exchange. The goal of this paper is to present techniques that can enable a data exchange system to survive system failures without losing fairness.

In the following, we use an example to clarify our motivation. The example will also be used throughout the paper to help illustrate our solutions. Consider the following fair non-repudiation protocol, which is adapted from Ref. [18], an efficient non-repudiation protocol that can ensure fairness when there are no system failures. Relevant notations of the protocol are summarized in Fig. 1. The protocol consists of five messages. Note that messages 4 and 5 are interchangeable. The protocol is to guarantee that when a message is transmitted from the sender A to the receiver B, if B has successfully received the message, then neither can B repudiate having received the message, nor can A repudiate having sent the message. In particular, the purpose of this protocol is for A to exchange a message together with a non-repudiation of origin (NRO) token, denoted $\{m, S_A(B, m.id, E_k(m))\}$, for a non-repudiation of receipt (NRR) token from B, denoted $\{S_B(A, m.id, E_k(m)), S_{TTP}(A, B, m.id, k)\}$. The rational is that if the exchange is fair then non-repudiation can be achieved. Note that the protocol is slightly different from that presented in Ref. [18] where messages 4 and 5 are retrieved by A and B instead of being delivered by the TTP. The impact of this difference on fairness is addressed in Section 4.

m	message sent from A to B
k	message key used by A
$E_k(m)$	m encrypted with k
$m.id$	unique identifier of m
$S_X(y)$	text y signed with the private key of party X

Fig. 1. Notation.

1. A → B: B, $m.id$, $E_k(m)$, $S_A(B, m.id, E_k(m))$
2. B → A: A, $m.id$, $S_B(A, m.id, E_k(m))$
3. A → TTP: B, $m.id$, k, $S_A(B, m.id, k)$
4. TTP → B: A, B, $m.id$, k, $S_{TTP}(A, B, m.id, k)$
5. TTP → A: A, B, $m.id$, $S_{TTP}(A, B, m.id, k)$

Based on the assumption that there are no local system failures, and the assumption that each communication channel is reliable, that is, the messages inserted into the channel by the sender can always be received by the recipient within a known, constant time interval, fairness of the protocol can be proved in a way similar to Ref. [18]. However, system failures, e.g., process crashes, can cause loss of fairness. To illustrate this, consider an exchange instance during which no player (process) stores any data item into the stable storage, if the process on behalf of player B crashes after message 2 is sent out, then message 1, which has been delivered to the process, is lost. As a result, when the protocol terminates, A will get the expected item, but B cannot, even if the process on behalf of B restarts instantly after the failure and gets message 4. Similar problems can also be caused by failures at the local systems of player A and the TTP.

This example raises several questions on how to survive system failures without losing fairness in data exchange, which to the best of our knowledge have not yet been clearly answered. We believe that answering these questions is important because it gives developers a much clearer understanding of the relationship between security (fairness) and fault tolerance (recoverability) in data exchange, which will not only help to develop secure and fault tolerant fair exchange systems, but also help to develop other kinds of secure and fault tolerant electronic commerce applications.

• Besides the problem identified in the example above, how many kinds of security problems caused by local system failures are with current fair data exchange protocols?

UNCORRECTED PROOF

• Can traditional fault tolerance mechanisms for distributed computing, such as transaction processing and message logging, be directly applied to fair exchange systems to avoid the fairness loss caused by system failures? How can they be applied to the fair exchange problem in a better way?

• What are the fault tolerance requirements for fair exchange systems?

In this paper, we present a systematic way to develop such data exchange systems that can recover from system failures without losing fairness. We identify a new design goal for fair exchange systems, i.e., surviving local system failures without losing fairness. We formalize a data exchange system as a specific distributed system and identify a set of security risks caused by system failures. We present two categories of techniques, namely, transaction-based approaches and message-logging-based approaches, to immunize a data exchange system from the security risks caused by system failures. Our techniques are application independent; thus, they can be enforced in all kinds of data exchange applications that use trusted third parties.

The remainder of this paper is organized as follows. Section 2 identifies the set of security risks caused by system failures. Section 3 investigates transaction-based approaches. In Section 4, a message-logging-based approach is presented. Section 5 concludes the paper.

2. Loss of fairness due to failures

2.1. Fair data exchange systems

Fair data exchange systems are implementations of a fair data exchange protocol. Existing third party protocols can be modeled by Fig. 2, where two players, A and B, and two communication channels between A and B, the normal channel and the trusted channel, are involved in an exchange. The normal channel models the direct communication between A and B. Since A and B mutually distrust each other, exchange solely dependent on this channel cannot be assured to be fair. The trusted channel between A and B is therefore established with the help of a TTP (A semi-trusted TTP is allowed in Ref. [12], where the third party may misbehave on its own but will not conspire with either of the players). Items sent via the trusted channel are first sent to the TTP and then forwarded by the TTP to the recipient. Exchange performed in the trusted channel can be considered fair because of the mediation of the TTP. Note that A and B are interchangeable in this model.

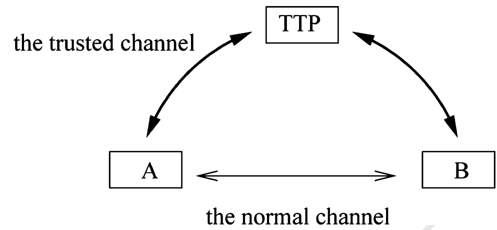


Fig. 2. Fair data exchange using a TTP.

Although exchanging items using the trusted channel is fair, it is usually undesirable since the TTP will then become the performance bottleneck and the main target of attacks. Due to this consideration, one of the design goals for fair exchange protocols is to minimize use of the TTP. There are also some other design goals identified: (1) Tolerating temporary communication failures without losing fairness [3]; (2) Relaxing the extent to which the TTP can be trusted without losing fairness [12]; (3) Relaxing the fairness requirement to further reduce the overhead of the TTP [3]. Note that these goals may conflict with each other; thus, trade-offs are needed in some situations.

Third party protocols can be classified into two categories in terms of how the TTP is exploited.

• *Exchange with on-line TTPs.* An exchange cannot be completed without using the trusted channel, even if both players play honestly (A player plays honestly if he or she will send his or her own item even after he or she receives the other party's item). Semi-trusted TTPs can be supported in some cases [12]. Protocols proposed in Refs. [10,12,18] fall into this class.

• *Exchange with off-line TTPs.* An exchange can be completed without interference of the TTP if the two players play honestly. If one player realizes that they cannot fairly exchange the items as expected, another exchange will be carried out in the trusted channel. Protocols proposed in Refs. [2–4] fall into this class. In some cases, only weak fairness can be

UNCORRECTED PROOF

achieved. That is, in some cases the players will have to use some specific affidavits in an external dispute resolution system, such as a court, to achieve fairness.

2.1.1. Exchange with on-line TTPs

Fair exchange protocols with on-line TTPs can be modeled by Fig. 3a where player A exchanges an item X for an item Y from player B. Each player splits its item into two parts: one is to be sent via the trusted channel, and the other via the normal channel. The item should be split in such a way that (1) from only the part sent via the normal channel the other player cannot figure out any information about the other part, and (2) the part sent via the normal channel is valueless to the other player without being combined with the other part. In the model, $X = f(X_1, X_2)$ and $Y = g(Y_1, Y_2)$, and no information about X_2 (Y_2) can be derived from X_1 (Y_1). Here, f and g are two functions that are determined by specific exchange protocols. For better performance, the part sent via the trusted channel should be as small as possible. As a result, the exchange of X for Y is split into two exchanges: the exchange of X_1 for Y_1 via the normal channel followed by the exchange of X_2 for Y_2 via the trusted channel. The fairness of the exchange is ensured by the fairness of the TTP in forwarding X_2 and Y_2. Sometimes, Y_2 can be generated by the TTP [18]. This situation is shown in Fig. 3b. Note that the example specified in Section 1 falls into this class. Sometimes, forwarding Y_2 from players B to A can be handled by the TTP without any message passing. For example, in NetBill [10], each player has an account maintained at the TTP, namely, the NetBill server, and forwarding money from players B to A is done by transferring money from Bs account to As. This situation is shown in Fig. 3c. To be more concrete, a protocol modeled by Fig. 3a is specified as follows. A protocol modeled by Fig. 3b or c can be specified in similar ways.

1. A → B: X_1
2. B → A: Y_1
3. A → TTP: X_2; B → TTP: Y_2
4. TTP → A: Y_2; TTP → B: X_2

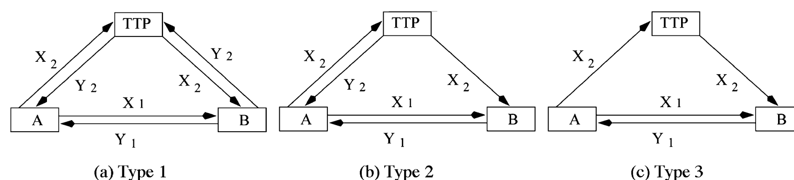
In practice, messages can be passed in different ways from what is shown in Fig. 3 to reduce overhead of the TTP. For example, in Ref. [18], X_2 and Y_2 are retrieved by A and B instead of being delivered by the TTP. In Ref. [10], after being signed by the TTP, X_2 is forwarded by A to B.

It is beneficial to distinguish the two possible ways in which the context of an exchange is handled by TTPs. TTPs in some exchange protocols have a property that is usually called stateless. Stateless TTPs forget an exchange as soon as it is done, that is, they keep no context information about the exchange. For example, the TTP in the basic protocol presented in Ref. [12] is stateless, whereas the TTPs in Refs. [18,10] are not stateless. Making TTPs stateless reduces the overhead of TTPs; however, as we show below, it may make an exchange protocol more vulnerable to failures.

In a fair data exchange system, each party usually has an agent, i.e., a process, running on his or her behalf. We use P_A, P_B, and P_{TTP}, to denote the agents that are running on behalf of player A, player B, and the TTP, respectively. During each exchange, P_A and P_B should first do the exchange according to a specific exchange protocol, then they can output the items they get to players A and B in a proper way, such as showing the item on the screen and sending the item to a printer.

2.1.2. Exchange with off-line TTPs

Fair exchange protocols with off-line TTPs can be modeled by Fig. 4a [2,3]. For simplicity, only the



(a) Type 1 (b) Type 2 (c) Type 3

Fig. 3. Fair data exchange with on-line TTPs.

UNCORRECTED PROOF

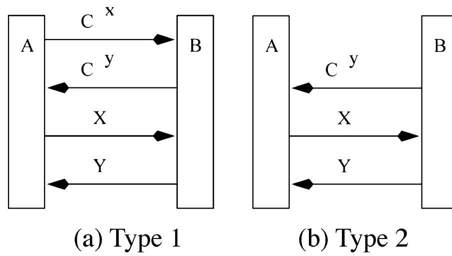


Fig. 4. Fair data exchange with off-line TTPs.

normal exchange procedure is specified. If players A and B play honestly, then they can exchange fairly without the interference of the TTP. Otherwise, if player A cannot get Y after X is sent to player B, then messages 3 (X) and 4 (Y) will be re-exchanged under the supervision of the TTP. The (weak) fairness is ensured by the following property:

$\exists f$ such that $f(C^x, C^y)=1$ indicates that P_A plays honestly; and
$\exists g$ such that $g(C^x, X)=1$ indicates that X is valid; $g(C^y, Y)=1$ indicates that Y is valid.

Sometimes, message 1 (C^x) need not be sent and the re-exchange of X and Y can be avoided even if P_B does not play honestly, instead, P_{TTP} can extract Y from C^y for P_A but P_A cannot do this by itself [4]. This situation is shown in Fig. 4b. However, in order to ensure fairness, the following property must be satisfied:

$\exists g$ such that $g(X, C^y)=1$ indicates that P_A plays honestly; and
$\exists f$ such that $f(C^y, K_{TTP})=Y$. Here, K_{TTP} is the private key of TTP.

2.2. Failures

To study the impact of failures on fairness, we must first specify what kinds of failures we are trying to overcome. We consider two categories of failures in a fair data exchange system: communication failures and system failures. We consider two types of communication failures: transmission failures that happen when a message is corrupted or lost, and channel breaks that happen when a communication link is broken for a while.

The simplest system failure model is the crash model. In the model, the only kind of failure is that a processor may suddenly halt and kill all the processes that are executing there. We say these processes crash. Operational processes never perform incorrect actions, nor do they fail to perform correct actions. Moreover, all operational processes can detect the failure of a processor. For the rest of the paper, we assume process crashes are the only kind of system failures that can occur. There are a couple of reasons for restricting our attention to crash failures. First, the abstraction of crash failures can be implemented on top of a system subject to more complex failures by running an appropriate software protocol [8]. Second, techniques are available to automatically translate a protocol that tolerates crash failures into protocols that tolerate larger classes of failures [15].

2.3. Fairness loss owing to failures

Process crashes can cause the following types of fairness loss in a fair exchange protocol modeled by Fig. 3. Here, we assume that no data item is stored to the stable storage during an exchange. Although an exchange protocol with an off-line TTP is quite different from a protocol with an on-line TTP, they suffer from similar types of fairness loss risks. However, it should be noticed that protocols with off-line TTPs will not suffer from fairness loss caused by TTP failures because the re-exchange process under the supervision of TTPs can be reenforced after P_{TTP} crashes. Identifying the types of fairness loss for protocols modeled by Fig. 4 is omitted here for space reason.

• *Type 0.* In a fair data exchange system, the fact that an agent successfully gets an item does not guarantee that the player that the agent works for will get the item. Failures during the output process can cause loss of fairness for players, although not for agents. In particular, if P_A (P_B) crashes after it gets Y_2 (X_2), but before it outputs Y (X), then player A (B) cannot get Y (X) but player B (A) may have already got X (Y), thus fairness is lost. This type of fairness loss is common to all exchange protocols.

UNCORRECTED PROOF

• The following two types are specific to the protocols modeled by Fig. 3a.

Type 1.1. If P_A (P_B) crashes after X_2 (Y_2) is sent out, then P_B (P_A) could get X (Y), but P_A (P_B) cannot get Y (X) since Y_1 (X_1) is lost.
Type 1.2. If P_{TTP} crashes after it delivers X_2 (Y_2), but before it delivers Y_2 (X_2), then P_B (P_A) could get $X(Y)$, but P_A (P_B) cannot get Y (X) since Y_2 (X_2) is lost. P_{TTP} is unable to force P_A (P_B) to provide X_2 (Y_2) again in many cases. For example, P_A (P_B) may lie that it has not got Y_2 (X_2) and it wants to abort the exchange.

• The following three types are specific to the protocols modeled by Fig. 3b.

Type 2.1. If P_B crashes after Y_1 is sent out, then P_A could get Y, but P_B cannot get X. Note that the fairness loss identified in the example specified in Section 1 is of this type.
Type 2.2. If P_A crashes after X_2 is sent out, then P_B can get X, but P_A cannot get Y. Type 2.2 is a special case of Type 1.1.
Type 2.3. If P_{TTP} crashes after Y_2 is delivered, but before X_2 is delivered, then P_A could get Y, but P_B cannot get X. Note that no loss of fairness will be caused when P_{TTP} crashes after X_2 is delivered but before Y_2 is delivered because P_A can get Y_2 later on by resending X_2 to restarted P_{TTP}.

• The following types are specific to the protocols modeled by Fig. 3c.

Type 2.1 can also apply here.
Type 3.1. If P_{TTP} crashes during transferring Y_2 from B's item-store (i.e., an account) to A's, then it is possible for P_{TTP} to repeatedly transfer Y_2 if the atomicity (i.e., all-or-nothing) of the transfer operation does not hold. As a result, fairness could be lost.

Channel breaks can also cause fairness loss. It is shown in Ref. [3] that channel breaks can destroy the fairness for the originator in some exchange protocols with off-line TTPs such as Ref. [2]. We found that in some exchange protocols with on-line TTPs, such as Ref. [18], channel breaks can cause fairness loss as well. In both cases, the main reason is that in practice the TTP usually would not allow an exchange to last for an arbitrary period of time before reaching a termination for a couple of reasons, i.e., saving resources, or enabling each player to know the exact final state of the exchange within a constant time interval. As a result, the TTP in some protocols will set up an overall time limit parameter and reject to serve an exchange after the specified time limit. For example, in a protocol modeled by Fig. 3, if the channel between player A and the TTP breaks when P_{TTP} forwards Y_2 to P_A, then P_{TTP} could stop trying to connect to P_A and forget the exchange after the specified time limit. As a result, the fairness for player A will be lost if player B gets X. In Ref. [18], X_2 and Y_2 are retrieved by A and B instead of being delivered by the TTP (thus, the TTP needs not to keep on trying to connect to a player); however, since a time limit is set up for the TTP to maintain X_2 and Y_2, channel breaks can also cause fairness loss in a similar way.

It should be noticed that transmission failures usually do not cause more fairness loss. Since transmission failures typically happen occasionally and can be overcome by a couple of retransmissions, transmission failures usually will not cause the types of fairness loss caused by channel breaks. Moreover, since requests in fair exchange protocols are typically idempotent, replayed messages that may be caused by transmission failures will not gain any more profits to either of the two players.

It should also be noticed that it is easy to show that concurrent failures will not cause more fairness loss. On the contrary, in some situations, concurrent failures can even avoid loss of fairness. For example, in a protocol modeled by Fig. 3a, when P_A crashes after X_2 is sent out, if there is a communication failure between player A and the TTP, or if P_{TTP} crashes when receiving X_2, then P_{TTP}, thus P_B, will not get X_2. Therefore, Type 1.1 fairness loss can be avoided in this situation.

In the above discussion, we assume that stateless TTPs are used. In practice, many TTPs are not stateless [18,10], and it is beneficial to notice that these TTPs can help to avoid fairness loss caused by failures with the context information they maintained for an exchange. For example, in Ref. [18] where X_2

UNCORRECTED PROOF

and Y_2 are first saved to stable storage by the TTP before being retrieved by A and B, Type 1.2 and Type 2.3 fairness loss can be avoided, because even if P_{TTP} crashes after X_2 is retrieved by P_B but before Y_2 is retrieved, P_A is still able to retrieve Y_2 after P_{TTP} restarts.

2.4. Possible solutions

The fact that failures can cause fairness loss in most of the existing fair data exchange protocols indicates that specific techniques are required to remove the negative impact of failures on fairness when implementing these fair data exchange protocols. In the following, we will present two categories of techniques, namely, transaction-based approaches and message-logging-based approaches, to help develop such data exchange systems that can recover from failures without losing fairness. Their advantages and disadvantages will also be discussed. To focus on these specific implementing techniques, we assume in the rest of the article that there are no channel breaks. Note that whether a fair data exchange system will suffer from fairness loss caused by channel breaks is primarily dependent on the underlying exchange protocol itself instead of how we implement the protocol. For example, Ref. [3] shows that changing an exchange protocol from synchronous to asynchronous can immunize the protocol from fairness loss caused by channel breaks.

3. Transaction-based approaches

Transactions have been widely used in database systems and distributed computing to provide reliability, availability, and performance [7,13]. A transaction can be considered a sequence of system-state-changing operations with the ACID properties, namely, atomicity, consistency, isolation, and durability. Atomicity allows us to run a transaction as a single unit, that is, when a transaction is executed, atomicity can ensure that either all the operations involved in the transaction are executed or none of them are. Therefore, atomicity can mask all the failures that may happen during the execution of a transaction. In this section, we study if transactions can be used as a mechanism to immunize a data exchange system from fairness loss risks caused by failures.

3.1. Limitations of transactions

As a distributed system with specific goals, i.e., achieving fairness in exchanges, fair data exchange systems can surely use transactions to help improve reliability. However, we found that although transactions can effectively mask the failures that may happen during an exchange, they cannot guarantee fairness, since fault tolerance does not always ensure fairness.

To illustrate, consider a protocol modeled by Fig. 3a, it is easy to see that if delivering X_2 and Y_2 is atomic, then Type 1.2 fairness loss can be avoided. Grouping these two message delivering operations into a single distributed transaction can technically provide such atomicity. In particular, according to the 2PC (two phase commit) protocol, a standard mechanism to ensure the atomicity of distributed transactions, the transaction commits only if both P_A and P_B get the item they want and vote YES; if P_{TTP} crashes during message delivery, the transaction will abort, and P_A and P_B will discard (or return) whatever they have received and stop.

However, in an environment where the agents involved in a transaction do not mutually trust each other, this type of atomicity is not enough to ensure fairness. In the above example, when the transaction aborts, that both P_A and P_B will discard (or return) whatever they have received does not mean that the items they have received will be kept confidential to the players they are working for before the abort. In fact, before a received item is discarded (or returned), players A or B may have already had a look of the item or even have made a copy of the item if they do not play honestly, because they have the ability to check the memory spaces of their agents whenever they want. As a result, although the atomicity in terms of delivering messages to agents is achieved, the atomicity in terms of delivering messages to players is lost and thus the fairness is lost. Based on similar reasoning, we can see that Type 2.1 fairness loss can not be avoided either by grouping multiple exchange protocol operations into a single transaction.

UNCORRECTED PROOF

The above discussion indicates that technical atomicity in terms of agents cannot guarantee fairness, and high level atomicities in terms of players, such as the atomicity of delivering messages to players, are required for data exchange systems. For example, in Ref. [17], Tygar identifies three levels of atomicity requirements for payment systems, namely, money atomicity, goods atomicity, and certified delivery. However, it should be noticed that these high level atomicity requirements usually cannot be satisfied by normal transaction processing.

Sometimes fairness can be lost even if a transaction commits. For example, in an exchange system modeled by Fig. 3a, sending X_1 to P_B and receiving Y_1 from P_B can be grouped into a single transaction to ensure the atomicity of the exchange performed on the normal channel. But if P_A crashes after the transaction commits and X_2 is sent out and before Y_2 is received, the reply Y_1 may still be lost, because that a distributed transaction commits means that the changes to every local system state involved in the transaction will be recoverable, but does not mean that every received message will be recoverable.

3.2. Usefulness of transactions

Although atomicity in terms of agents cannot guarantee fairness, transactions can still be very useful in many aspects of fair exchange systems. First, transactions can easily ensure the atomicity of a sequence of operations within the TTP. As a result, Type 3.1 fairness loss can be avoided if a transaction is used to transfer Y_2 from B's item-store to A's.

Second, transactions can make each player not ambiguous about the current state of an exchange. In a protocol modeled by Fig. 3a, after X_2 is successfully sent to TTP, player A is committed to the exchange and must be prepared to accept Y. We informally denote this special state as the point-of-no-return for A. However, if a failure happens during transmitting Y, then player A may be unable to determine whether he or she has really passed his or her point-of-no-return, because if P_{TTP} does not receive X_2 correctly then A has not passed his or her point-of-no-return, but if P_{TTP} does receive X_2 correctly but the acknowledge to P_A is lost, then A has. This ambiguity can be avoided by using a transaction to transmit X_2. Since TTP is trusted, atomicity of the transaction can enable A to unambiguously determine whether or not he or she has entered his or her point-of-no-return.

Third, in distributed transaction processing, a TP monitor is set up on each site involved in a distributed transaction to manage the context of the transaction. We found that the context information maintained by TP monitors can help avoid fairness loss. For example, a TP monitor typically maintains a queue to buffer incoming and outgoing messages. Thus, in a protocol modeled by Fig. 3a, after X_1 (Y_1) arrives at B (A), the corresponding TP monitor will first enqueue X_1, then P_B can dequeue X_1 and process it. Although, if P_B crashes after sending out Y_2, the fairness will still be lost because the dequeued X_1 is lost, if we make the incoming queue transactional then the fairness loss can be avoided. Transactional queues [13], also called *persistent queues*, make messages recoverable even after being dequeued by logging each enqueue operation to stable storage.

The major drawback of using distributed transactions in fair exchange systems is that significant extra costs and performance penalty can be caused. Running a distributed transaction needs a TP monitor, a transaction manager, and a log manager for each participant involved in the transaction. Moreover, the 2PC protocol needs a coordinator and two rounds of message interactions between the coordinator and each participant. These costs can be well rewarded for the distributed applications where the ACID properties are critical. However, for a fair exchange system many of these costs could be wasted because the success of an exchange is not dependent on the ACID properties. In some cases, using transactions can even cause extra performance penalty. Reconsider the situation where a transaction is used to make delivering X_2 and Y_2 atomic. Since atomicity can not guarantee fairness, the corresponding transaction management costs are wasted, and the transaction rollback action caused by transmission failures during the delivery can cause more delay than simple retransmissions. Although we can avoid Type 1.2 fairness loss by maintaining a persistent incoming queue on the TTP, the corresponding cost is still more than simply logging X_2 and Y_2 as soon as they are received by P_{TTP}.

UNCORRECTED PROOF

4. Message-logging-based approaches

Message logging is the standard method in the literature of distributed systems to achieve recoverability. In this section, we study the application of message logging to fair exchange protocols, trying to use the technique in an appropriate way. Compared with transaction-based approaches, message logging can be cheaper.

We view an exchange as a set of interactions among three processes: P_A, P_B and P_{TTP}. Processes interact by sending and receiving messages. Each process has a local state and performs computation based on the current state, which is kept in the volatile storage. For example, during an exchange modeled by Fig. 3a, the actions which P_A will take after receiving a message from P_B or P_{TTP} are dependent on the state variable indicating which phase P_A is currently in the exchange.

Processes execute events, including *send* events, *receive* events, and *local* events [1]. Each event transforms one process state to another. Therefore, a process can be viewed as an interleaved sequence of events and states. A local event is *deterministic*, if based on the same current process state (and the same message used in the event), the event does the same state transformation and outputs the same message to the following send event (if there is any). An exchange system is deterministic if the local events of P_A, P_B, or P_{TTP} are all deterministic. Note that fair exchange protocols modeled by Fig. 3 can be implemented by a deterministic exchange system. One may doubt that the event of splitting X (Y) into X_1 (Y_1) and X_2 (Y_2) is nondeterministic, since an item is usually split using a randomly chosen cryptographic key to make the split unpredictable. However, the split event is deterministic if we consider the random key as a message.

4.1. Traditional message-logging approaches and their limitations

A message is *logged* if both its content and sequence number have been saved on stable storage. Assuming that each local event executed by a process is deterministic, it is then always possible to reconstruct the current process state from the set of messages delivered. Occasionally, a process takes a *checkpoint*, a snapshot of it local state, and writes it to the stable storage to reduce recovery time. A process state is *recoverable* if every message delivered after the latest checkpoint is logged.

In our model, a global state S of an exchange system is a collection of three process states: s_A, s_B and s_{TTP}, one for each process. Viewing processes as sequences of events and states, a global exchange system state S can also be viewed as the collection of three subsequences: (1) the subsequence of P_A ended with s_A; (2) the subsequence of P_B ended with s_B; and (3) the subsequence of P_{TTP} ended with s_{TTP}. A global exchange system state S is consistent if and only if every message received via a receive event of S has been sent out via a sent event of S. A consistent global exchange system state S is recoverable if s_A, s_B and s_{TTP} are all recoverable.

A naive approach to achieve recoverability of a fair exchange protocol is to use pessimistic message logging, which has been applied in many systems to support transparent, application-independent recovery [5,9]. The mechanism is *pessimistic* because it never rolls back process computations. The application of pessimistic message logging to fair exchange is straightforward. In a data exchange system modeled by Fig. 3, a message m is always first logged before being delivered to another agent.

It is easy to show that for a deterministic data exchange system pessimistic message logging not only ensures that the current global state is always recoverable, but also ensures that all types of fairness loss risks (excluding Type 3.1 risks and the risks caused by channel breaks) we have previously identified can be avoided because no messages will be lost any more when a process crashes. To illustrate, in the example specified in Section 1, since message 1 will be logged before being delivered to P_B, so even if P_B crashes after message 2 is sent out, P_B can still get message 1 (from the log) after being restarted.

The advantage of pessimistic message logging is the absence of cascading rollback, since there always exists a consistent system state constructible from the most recent checkpoint and the logged messages. However, the writing time for a message to be logged before processing can be significant, compared with the computation. This is unacceptable in

UNCORRECTED PROOF

an environment where a large number of messages are exchanged. This results in the optimistic message-logging approach [14,16]. Optimistic message logging allows messages to be processed independent of when they are logged, in an asynchronous manner. In the absence of failure, the only overhead is the asynchronous logging of messages.

Optimistic message logging ensures that a maximal recoverable system state is reconstructible after a failure. (Readers can refer to Ref. [14] for the definition of maximal recoverable system states and the algorithm to compute such states. Details are omitted here for space reason.) However, this kind of recoverability does not ensure fairness. Consider the example shown in Fig. 5 where P_0 denotes an external process that generates inputs for P_A, P_B, and P_{TTP}. We view these inputs as the first messages delivered to P_A, P_B, and P_{TTP}, respectively. And we assume that only the input delivered to each process is logged but all the other messages are not. Fig. 5 depicts the message interactions of an exchange modeled by Fig. 3a. When P_A crashes after P_{TTP} sends X_2 to P_B, according to optimistic message logging, all volatile messages delivered to P_B and P_{TTP} will be logged. At this time, although state S_4 is the current system state, it is not recoverable because the second message delivered to P_A is lost. In fact, the maximal recoverable state is S_1. However, the ability to find the maximal recoverable state does not imply the ability to ensure fairness. In particular, before we roll back the system state to S_1 the fairness may already be lost because after getting X, P_B may reject to continue the exchange process from state S_1.

4.2. Semantics-based message logging

Pessimistic message logging can ensure fairness in fair data exchange but may result in unacceptable performance, while optimistic message logging cannot even ensure fairness. Moreover, in both pessimistic and optimistic message logging, messages are logged regardless whether they contribute to the fairness of an exchange or not. This will inevitably result in extra performance penalty, especially when most of the logged messages are unnecessary to ensure the fairness.

The limitations of traditional message-logging approaches are due to the ignorance of the semantics of fair exchange protocols. In this section, we identify the semantics of fair exchange protocols that can be exploited to ensure both fairness and good performance.

Definition 1. A state of player process P_A (P_B) is called the point-of-no-return of P_A (P_B) if P_B (P_A) can get X (Y) without further information provided

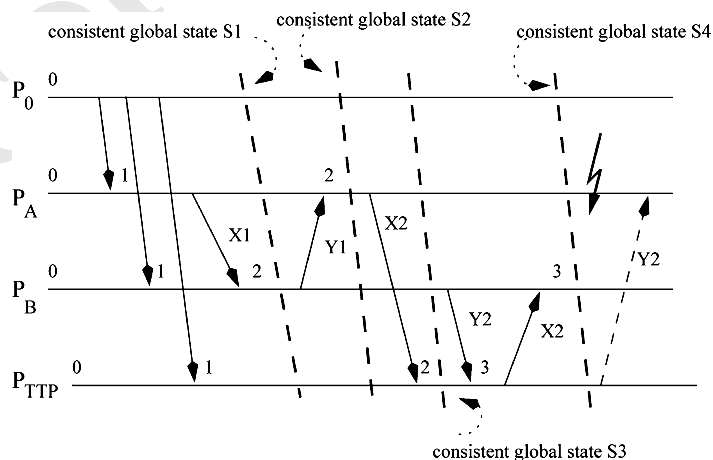


Fig. 5. Recoverable state with loss of fairness.

UNCORRECTED PROOF

by P_A (P_B) after P_A (P_B) enters the state, and there are no other such states, which P_A (P_B) can enter before it enters the state.

Intuitively, the point-of-no-return of a player in a data exchange represents the stage after which the player cannot take back his data item even if the exchange is stopped. Thus, in order to avoid losing the fairness due to process crashes, the player process must remember what has happened so that it can continue the exchange even after a crash. Semantics of points-of-no-return has been mentioned in some specific exchange protocols such as Ref. [10].

It is easy to see that fair exchange protocols modeled by Fig. 3 have the following points-of-no-return: (1) for Type 1 protocols, the point-of-no-return of P_A is the state that P_A enters by sending out X_2; the point-of-no-return of P_B is the state that P_B enters by sending out Y_2; (2) for Type 2 protocols, the point-of-no-return of P_A is the state that P_A enters by sending out X_2; the point-of-no-return of P_B is the state that P_B enters by sending out Y_1; (3) for Type 3 protocols, the points-of-no-return are the same as for Type 2 protocols. Similarly, in protocols modeled by Fig. 4, sending out X let P_A enter its point-of-no-return, and sending out Y let P_B enter its point-of-no-return. Finally, it is easy to see that in the example specified in Section 1 sending out message 2 let P_B enter its point-of-no-return, and sending out message 3 let P_A enter its point-of-no-return.

Similar to the two player systems, there is also a critical state for P_{TTP}, which is defined as follows. It is easy to see that in a protocol modeled by Fig. 3, if all the messages that have already been delivered to P_{TTP} are logged when P_{TTP} enters its point-of-fair-delivery, then Type 1.2 and Type 2.3 fairness loss can be immunized. Note that the agents for protocols with off-line TTPs have no points-of-fair-delivery.

Definition 2. A state of process P_{TTP} is called the point-of-fair-delivery of P_{TTP} if after P_{TTP} enters the state only X_2 or Y_2 is delivered but not both, and there are no other such states that P_{TTP} can enter before it enters the state.

Based on the notions of points-of-no-return and points-of-fair-delivery, we can define a correctness criteria for fault-tolerant fair data exchange systems as follows. See that the correctness of pessimistic message logging can also be justified by the fact that it achieves fairness–lossless recoverability.

Definition 3. In a fair exchange system where P_A, P_B or P_{TTP} may crash concurrently, a global exchange system state S is fairness–lossless recoverable if whenever a crash happens,

- If P_A (P_B) has entered its point-of-no-return, then all the messages delivered to P_A (P_B) before P_A (P_B) entered its point-of-no-return are logged, and
- If P_{TTP} has entered its point-of-fair-delivery, then all the messages delivered to P_{TTP} before P_{TTP} entered its point-of-fair-delivery are logged.

Based on our discussion on traditional message-logging approaches, it should be easy to show that fairness–lossless recoverability can generally ensure the fairness of deterministic fair exchange systems in presence of failures. In particular, if a data exchange system modeled by Fig. 3 is deterministic and fairness–lossless recoverable, then the system is immunized from all types of fairness loss caused by process crashes except for Type 0 and Type 3.1.

There can be several solutions for immunizing the system from Type 0 fairness loss risks. For example, (1) letting P_A (P_B) output Y_2 (X_2) before acknowledging receiving Y_2 (X_2), or (2) letting P_A (P_B) log Y_2 (X_2) before acknowledging receiving Y_2 (X_2). In both methods, P_{TTP} will continue to be active unless player A (B) is assured to get Y (X).

Type 3.1 fairness loss risks can not be avoided by message-logging approaches because they are not caused by unrecoverable messages. Instead, as we have shown, using transactions inside TTP is a sound and desirable solution to avoid Type 3.1 risks. This also shows that (1) transaction-based approaches and message-logging-based approaches are usually complementary to each other, and (2) the approaches needed to immunize a data exchange system from fairness loss caused by failures are usually hybrid.

It should also be easy to show that fairness–lossless recoverability can also generally ensure fairness in exchanges with off-line TTPs. The difference is that since agents for off-line TTPs have no points-

UNCORRECTED PROOF

of-fair-delivery, there is no recoverability requirements for P_{TTP}.

Although reducing the overhead of TTPs is critical to the overall performance of a data exchange system, logging messages cannot be avoided in exchange systems with on-line TTPs in order to ensure fairness. However, the overall performance of a data exchange system can be improved by optimizing the ways we do message logging at the TTP. For example, in Ref. [18], after messages are logged, they are retrieved by players instead of being delivered by the TTP. In this way, P_{TTP} is relieved from sending messages directly to players, which can be very resource consuming when many communication failures happen. In Ref. [10] (see Fig. 3c), after X_2 is logged by the TTP, it will be signed and returned to player A, which will then forward it to player B. In this way, P_{TTP} does not need to open a new connection to P_B. Nevertheless, it should be noticed that generally data exchange systems with off-line TTPs have better performance than those with on-line TTPs, not only because TTPs are not involved in many exchanges, but also because even if a TTP is involved in a exchange it is not required to log messages.

In the following, we propose an algorithm to achieve fairness–lossless recoverability for fair exchange systems using on-line TTPs. Note that the algorithm can be directly extended to exchange systems with off-line TTPs except that P_{TTP} needs no recovery services. We assume that (1) there is a recovery manager, a process responsible for logging, at each player's local system, and (2) whenever a message is delivered to a player it is also delivered to the recovery manager. We augment traditional optimistic message-logging systems with a specific facility called message filter, a process responsible for detecting points-of-no-return or points-of-fair-delivery, at each local system. We assume that every message sent out by an agent is forwarded by a message filter to communication channels.

Algorithm 1 (Semantics-based message logging). For a data exchange system modeled by Fig. 3.
(1) Before the data exchange system starts to function, each player registers its point-of-no-return with its Message Filter. The TTP registers its point-of-fair-delivery with its Message Filter.
(2) When an agent (P_A, P_B, or P_{TTP}) is sending out a message, its Message Filter checks if sending out the message will make the agent enter its point-of-no-return or point-of-fair-delivery. If the answer is YES, the Message Filter will inform the Recovery Manager to log all the messages that have already been delivered to the agent. The message will not be sent out to the channel until the Recovery Manager succeeds. If the answer is NO, the message will be forwarded to the channel.
(3) The recovery actions taken by the system when a set of messages are logged, the method to compute the maximal recoverable state after a crash, and the rolling back process are the same as those performed in a normal optimistic message logging system such as Ref. [14].
(4) After a crash, each agent rolls its state back to the corresponding one specified in the maximal recoverable system state and continues its computation.

Similar to pessimistic message logging, the semantics-based approach is *application independent*. The only information that a data exchange protocol needs to inform the recovery system is the point-of-no-return of each player and the point-of-fair-delivery of the TTP. Since this interaction is finished by an off-line registration process, the codes of the protocol need not be tailored. As a result, recovery facility can be provided by the system and transparent to the fair exchange protocols. With almost no modifications, existing fair exchange protocols, which assume no system failures, can be directly performed on a platform where semantics-based message logging is enforced to avoid the fairness loss risks caused by process crashes.

Algorithm 1 can cause two kinds of overheads: failure-free overhead and recovery overhead. Failure-free overhead is the fixed overhead imposed on the system even when there is no failure. Recovery overhead is the cost paid to restart the system from a certain state. Each kind of overheads can be broken down into three parts: disk I/O overhead, message overhead, and computation overhead. Assuming computation is much faster than communication and disk I/O, we can concentrate on the disk I/O overhead and the message overhead. In particular, the overhead of Algorithm 1 when being applied to a data exchange system modeled by Fig. 3a can be

UNCORRECTED PROOF

summarized as follows. We call the messages of the underlying data exchange protocol *application messages* and the messages of Algorithm 1 *recovery messages*.

• The maximum failure-free overhead of Algorithm 1 is six disk I/O operations and six recovery messages. The minimum failure-free overhead is three disk I/O operations. (Note that each disk I/O operation can handle more than one messages) and three recovery messages.

• The maximum recovery overhead of Algorithm 1 is four disk I/O operations and five recovery messages. The minimum recovery overhead is zero disk I/O operations and zero recovery messages.

4.3. Application specific approaches

Although having the advantage of being transparent, application-independent methods usually cannot achieve optimal performance in terms of the amount of computing resources used. Better performance is usually achievable by incorporating recovery semantics directly into fair exchange system code. Consider the example specified in Section 1, before P_B enters its point-of-no-return in step 2, it may log only the NRO token, namely, $S_A(B, m.id, E_k(m))$, instead of the whole message 1 after verifying the token. Similarly, P_A can only log the data item $S_B(A, m.id, E_k(m))$ instead of the whole message 2. In this way, the I/O cost can be reduced. The drawback of application specific approaches is that the developing cost and the possibility of incurring faults can be substantially increased. It should be noticed that no general application specific approaches can be presented here because they are dependent on application semantics and thus may be different from one application to another.

5. Conclusion

In this paper, we systematically identified and studied the negative impact of failures on the fairness of fair exchange systems. We investigated the application of two categories of techniques, namely, transaction-based approaches and message logging, to tackle the impact. Although transactions can be very useful in such aspects of fair exchange systems as enabling players to unambiguously determine whether or not they have passed their points-of-no-return, atomicity of transactions can not guarantee fairness. Message logging is a cheaper approach; however, traditional message-logging approaches are limited in achieving both fairness and good performance. We proposed a semantics-based approach that can exploit exchange protocol semantics to achieve these goals.

References

[1] L. Alvisi, K. Marzullo, Message logging: pessimistic, optimistic, causal and optimal, IEEE Transactions on Software Engineering 24 (2) (1998) 149–159.
[2] N. Asokan, M. Schunter, M. Waidner, Optimistic protocols for fair exchange, Proceedings of 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1997.
[3] N. Asokan, V. Shoup, Asynchronous protocols for optimistic fair exchange, Proceedings of IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 1998, pp. 86–99.
[4] F. Bao, R.H. Deng, W. Mao, Efficient and practical fair exchange protocols with off-line ttp, Proceedings of IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 1998, pp. 77–85.
[5] J.F. Bartlett, A ‘nonstop’ operating system, Proceedings 11th Hawaii International Conference on System Sciences, Hawaii, 1978.
[6] M. Ben-Or, O. Goldreich, S. Micali, R.L. Rivest, A fair protocol for signing contracts, IEEE Transactions on Information Theory 36 (1) (1990) 40–46.
[7] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.
[8] K.P. Birman, T.A. Joseph, Reliable communication in the presence of failures, ACM Transactions on Computer Systems 5 (1) (1987) 47–76.
[9] A. Borg, J. Baumbach, S. Glazer, A message system supporting fault tolerance, Operating System Review 17 (5) (1983) 90–99.
[10] B. Cox, J.D. Tygar, M. Sirbu, NetBill security and transaction protocol, Proceedings of the First USENIX Workshop on Electronic Commerce, New York, July 1995, pp. 77–88.
[11] R.H. Deng, L. Gong, A.A. Lazer, W. Wang, Practical protocols for certified electronic mail, Journal of Network and System Management 4 (3) (1996).
[12] M.K. Franklin, M.K. Reiter, Fair exchange with a semi-trusted third party, Proceedings of 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1997, pp. 1–7.

UNCORRECTED PROOF

[13] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993.
[14] D.B. Johnson, W. Zwaenepoel, Recovery in distributed systems, Journal of Algorithms 11 (3) (1990) 462–491.
[15] G. Neiger, S. Toueg, Automatically increasing the fault-tolerance of distributed systems, Proceedings Seventh ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, August 1998.
[16] R.E. Strom, S. Yemini, Optimistic recovery in distributed systems, ACM Transaction on Computer System 3 (3) (1985) 204–226.
[17] J.D. Tygar, Atomicity versus anonymity: distributed transactions for electronic commerce, Proceedings 24th VLDB Conference, New York, 1998, pp. 1–12.
[18] J. Zhou, D. Gollmann, A fair non-repudiation protocol, Proceedings of IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 1996, pp. 55–61.

Peng Liu is an assistant professor of Information Systems at the University of Maryland Baltimore County. He has published in *Distributed and Parallel Databases*, and *Journal of Computer Security*. His current research interests are in information security, survivability and information warfare, semantics-based transaction processing, and e-commerce. He is an IEEE member and an ACM member.

Peng Ning received a BS degree in Information Science in 1994 and an MS degree in Communication and Electronic Systems in 1997, both from the University of Science and Technology of China. Since 1997, he has been a PhD student in the School of Information Technology and Engineering and the Center for Secure Information Systems at George Mason University, Fairfax, VA. His main research interests include computer and network security, secure electronic commerce and applied cryptography. Peng Ning is a member of the IEEE and the IEEE Computer Society.

Sushil Jajodia is BDM Chair Professor and Chairman of Department of Information and Software Engineering and Director of Center for Secure Information Systems at the George Mason University, Fairfax, VA. He joined GMU after serving as the director of the Database and Expert Systems Program within the Division of Information, Robotics, and Intelligent Systems at the National Science Foundation. Before that, he was the head of the Database and Distributed Systems Section in the Computer Science and Systems Branch at the Naval Research Laboratory, Washington. He has also been a visiting professor at the University of Milan, Italy and at the Isaac Newton Institute for Mathematical Sciences, Cambridge University, England.

Dr. Jajodia received his PhD from the University of Oregon, Eugene. His research interests include information security, temporal databases, and replicated databases. He has authored three books including Time Granularities in Databases, Data Mining, and Temporal Reasoning (Springer-Verlag, 2000) and Information Hiding: Steganography and Watermarking—Attacks and Countermeasures (Kluwer, 2000), edited 16 books, and published more than 200 technical papers in the refereed journals and conference proceedings. He received the 1996 Kristian Beckman award from IFIP TC 11 for his contributions to the discipline of Information Security, and the 2000 Outstanding Research Faculty Award from GMU's School of Information Technology and Engineering.

Dr. Jajodia has served in different capacities for various journals and conferences. He is the founding editor-in-chief of the Journal of Computer Security. He is on the editorial boards of IEEE Concurrency, ACM Transactions on Information and Systems Security, and International Journal of Cooperative Information Systems. He is the consulting editor of the Kluwer International Series on Advances in Information Security. He serves as the program chair of the 2000 ACM Conference on Computer & Communications Security (CCS'00) and 2001 International Conference on Conceptual Modeling (ER2001). He also serves on the Board of Directors of the National Colloquium for Information Security Education (NCISSE). He has been named a Golden Core member for his service to the IEEE Computer Society. He is a senior member of the IEEE and a member of IEEE Computer Society and Association for Computing Machinery. The URL for his web page is isse.gmu.edu/ ~ csis/faculty/jajodia.html.

UNCORRECTED PROOF