

Systematic Development of a Family of Fair Exchange Protocols

Paul D Ezhilchelvan and Santosh K Shrivastava
University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK

Abstract: Fair exchange protocols play an important role in application areas such as e-commerce where protocol participants require mutual guarantees that a transaction involving exchange of items has taken place in a specific manner. A protocol is fair if no protocol participant can gain any advantage over an honest participant by misbehaving. In addition, such a protocol is fault tolerant if the protocol can ensure that an honest participant does not suffer any loss of fairness despite any failures of the participant's node. This paper presents a family of fair exchange protocols for two participants under a variety of assumptions concerning participant misbehaviour, node reliability and message delays. While the paper considers, in line with other known protocols, that a dishonest user can control the protocol execution subject to known cryptographic assumptions, it also considers a class of dishonest users whose abilities to abuse are restricted. The restricted-abuse assumption results in a round-optimal (2-round) protocol that eliminates any need for an "after-the-fact" dispute resolution, using traditional symmetric keys and the RSA signatures. The paper also indicates how this assumption can be realised through the use of smart cards. Concerning the node reliability, the development presented here shows how a non-fault tolerant version of a protocol can be made crash-tolerant, thereby highlighting issues that need to be addressed in such transformation. The third dimension concerns the case of the bound on message delays being known or unknown.

Keywords and Phrases: Fair Exchange, Security, Smartcards, Fault tolerance, Distributed Systems.

1. Introduction

Fair exchange protocols play an important role in application areas where protocol participants require mutual guarantees that an exchange of data items has taken place in a specific manner. An exchange is fair if a dishonest participant cannot gain any advantage over honest participants by misbehaving. Practical schemes for fair exchange require a trusted party that essentially plays the role of a notary in the paper based schemes. (Gradual Exchange protocols [BGMR90] which do not need a trusted party have high communication overhead.) Two-participant fair-exchange protocols that make use of a trusted third party have been studied extensively in the literature (e.g., [ZG97, PSW98, ASW98]); these protocols maintain fairness even if the dishonest participant can tamper with the protocol execution in an unrestricted (malicious) manner. They however require that an honest participant's node execute the protocol correctly – suffering no failures. In other words, fault tolerant fair exchange protocols have not been studied adequately. A fair exchange protocol is fault-tolerant if it ensures no loss of fairness to an honest participant even if the participant's node experiences failures of the assumed type.

In this paper we develop a number of fair exchange protocols under a variety of assumptions concerning user misbehaviour, node failures and communication delays. Our development is systematic: we begin by classifying dishonest participants into *restricted abusers* (they cannot tamper with the protocol execution in an arbitrary manner) and unrestricted or *malicious abusers*, and the communication model into synchronous, where a known bound on message delays exists, and asynchronous; we develop the very first protocol under the most constrained set of assumptions: restricted abuser, synchronous communication and no fault tolerance. We then relax the restricted abuser assumption to malicious abuser

and then the synchrony assumption into asynchrony. The resulting family of non-fault-tolerant fair exchange protocols is then transformed into a family of crash-tolerant protocols.

A major advantage of our approach is that it enables a reader to gain a deeper understanding of the impact of a given set of assumptions on the problem of fair exchange; it also highlights the relationships that exist between fairness, fault tolerance and communication delay assumptions. In particular, we show that the use of (optimistic) message logging for crash tolerance is more subtle than that suggested in [LNJ00] - the first paper to consider fault-tolerance and fair-exchange. We first describe the problem and the underlying system models (section 2), and then develop a family of non-fault-tolerant protocols (section 3), followed by their crash-tolerant counterparts (section 4). In section 5, we describe the use of smart cards in restricting dishonest behaviour. Section 6 identifies issues that need to be considered when message logging is used for crash tolerance and concludes the paper.

2. System Models and the Problem Description

We consider two mutually untrusting users, U_A and U_B , who have data items I_A and I_B respectively that are unknown to the other user prior to the exchange. User U_x , $X \in \{A, B\}$, intends to send I_x in return for receiving I_y , where $Y \in \{A, B\}$ and $Y \neq X$. P_x denotes the process that executes an exchange protocol on behalf of user U_x on his node N_x . Our distributed exchange system (Figure 1) has a third node hosting the trusted third party (TTP) process. The exchange must preserve fairness as well as *non-repudiation*: a user gets irrefutable evidence of the actions performed by his process (to guard against the other user's false denial of the occurrence of an action or event).

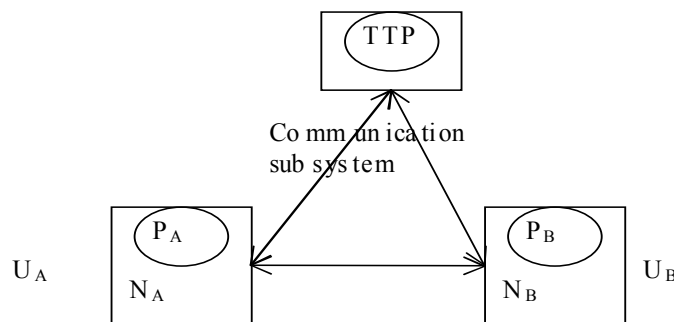


Figure 1. The Fair-Exchange System.

2.1. Classifying User Misbehaviour

Generally, the problem of fair exchange is addressed in a context where a dishonest user, say U_B , *totally controls* the behaviour of P_B and tries to undermine every attempt to ensure fairness and non-repudiation. We term those dishonest users as *malicious abusers* and distinguish such users from a class of *restricted abusers*: a dishonest user is said to be a restricted abuser if he is unable to obtain the values of variables or procedures contained in a piece of (binary) code which his local process receives from the TTP and uses during protocol execution. Say, user U_B is a restricted abuser and the TTP gives P_B a piece of code Π_B that contains, say, some secret encryption keys. By our definition, U_B cannot obtain the keys in Π_B by visual inspection or through a trace analysis even after P_B has executed U_B . The definition does not however restrict the ability of U_B to delay, block, inspect, or tamper with any message which an execution of Π_B generates or is destined to receive.

There are many ways to realise our notion of a restricted abuser. Code obfuscation, used in software watermarking and tamper-proofing (see [CT00] for a survey), is one approach. However, a totally secure, code-obfuscation is impossible to achieve [BG01]: there exists a non-zero probability that an obfuscated program ceases to be a virtual black box to an abuser. Thus, relying on code-obfuscation involves making

assumptions on the abilities of dishonest users. An alternative approach is to have each P_X receive the TTP code Π_X via a secure channel and execute it in a tamper-proof part of the user node. We observe here that there is much interest in developing tamper-proof computing subsystems by the industry led Trusted Computing Platform Alliance, TCPA [TCPA01]. In section 5, we describe how smartcards can be used to provide such tamper-proof computing subsystems. So, we believe that developing fair-exchange protocols under the restricted abuser model is of practical interest; in addition, it exposes the benefits that the model has to offer.

2.2. Classifying Node Behaviour

A fair-exchange protocol is fault-tolerant if it ensures that fairness is preserved to an honest participant despite the participant's node suffers failures of the assumed type. In other words, an honest user does not suffer loss of fairness when his node is unreliable. We consider two types of node behaviour:

Reliable: An honest user's node never fails.

Crash-recovery: An honest user's node fails by stopping to function (crash); it recovers within some finite (but unknown) time after a crash and may crash again after its recovery. An honest user's node has access to a stable store whose contents survive the node crash.

When a user node is regarded to be reliable, the cause of any node crash is attributed to the user who has misbehaved and is therefore not entitled to any fairness guarantee. This is the traditional view taken in the TTP-based protocols (e.g., [ZG97, PSW98, ASW98]) which are therefore not fault-tolerant (see [LNJ00]). In line with the existing, non-fault-tolerant protocols, the TTP is assumed to be *reliable* and *secure* against intrusions and Trojan horses.

2.3. Classifying Communication Delays

Communication between correctly functioning processes is assumed to be resilient to network failures: a message loss is tolerated by retransmission and a message corruption is detected using encryption and reduced to a message loss. A network intruder is therefore assumed to be a restricted abuser: a message in transit is a black box to him; he cannot modify it and have it undetected at the destination. He can only delay a message from being accepted at a destination. We consider two types of network intrusion: intruder gives up delaying a given message transmission after a known period of time or after some unknown finite time, leading to two models: in the *synchronous* model, correct processes exchange messages with delays bounded by a known D ; in the *asynchronous* model, D is finite but unknown.

2.4. Properties of a fair exchange Protocol

A user is *honest* if he makes no attempt to modify the behaviour of a protocol process except through the permitted operations.

Termination: Any execution of the protocol terminates for an honest user U_X . Termination for an honest U_X can be either a *normal* termination in which P_X delivers I_Y to U_X or an *exceptional* termination where P_X informs U_X that the exchange is unsuccessful.

No_Loss_Of_Fairness: If P_X of honest U_X terminates exceptionally, U_Y does not receive I_X .

Non-repudiation: when P_X delivers I_Y to honest U_X , it also provides irrefutable evidence against any false denial on I_Y having been sent by U_Y .

The above properties are met trivially if P_A and P_B terminate always exceptionally.

Non-Triviality: If U_A and U_B are honest, both are guaranteed to have normal termination.

Non-triviality is easy to guarantee for the synchronous, non-fault-tolerant case. When the fair-exchange protocol is deterministic (not probabilistic) in nature, honesty of both the users alone is not sufficient to guarantee non-triviality for the case of crash-recovery nodes and synchronous communication, as well for the case of asynchronous communication. The reasons for this inability are as follows. In crash-recovery model, the bound on the time taken by an honest user node to recover from a crash is unknown and a dishonest user may never allow his node to recover. So, a correctly functioning process (of an honest user

or the TTP) that is waiting too long for a message from a user process cannot resolve whether the latter is honest and its message is unduly delayed due to a crash or is dishonest and is not going to transmit the expected message; similarly, in the asynchronous model, it cannot resolve whether the user process from whom a message is expected is honest and its message is still in transit or is dishonest and the message will not be transmitted at all¹. Therefore, meeting the termination property would mean that an execution of the protocol may have to terminate exceptionally even if both users are honest. Therefore, the necessary condition for ensuring non-triviality is that the honest users be willing to re-execute the protocol after every exceptional termination; the sufficient condition differs with the combination of assumptions and is stated below: there must be an execution in which

- (i) user nodes do not crash, for the case of crash-recovery nodes and synchronous communication;
- (ii) message delays do not increase, for the case of reliable nodes and asynchronous communication; and,
- (i) and (ii), for the case of crash-recovery nodes and asynchronous communication.

2.5. Assumptions, Notations and the Exchange Preliminaries

In all our protocols, the TTP, assumed to be secure and reliable, is used to set-up the context for, and to initiate, the actual exchange. We make the following additional assumptions.

A1: Processing within correctly functioning nodes is synchronous: delays for task scheduling and processing are bounded by a known constant.

A2: The clocks of the TTP and the functioning nodes of honest users are synchronised to real-time within a known bound which, for simplicity, is assumed to be zero. A recovering node receives the current time from its user.

2.5.1. Notations

M: a message;

$V_A (V_B)$: procedure to verify if $I_A (I_B)$ satisfies its description $\Sigma_A (\Sigma_B)$ advertised by $U_A (U_B)$.

$eK(M)$: encryption of M using key K;

$Sig_X(M)$: signature of P_X , $X \in \{A, B\}$, on M using the private key of U_X ;

$Sig_{TTP}(M)$: TTP's signature on M. Signatures are both signer- and content-dependent, and are verifiable using the signer's public key.

L: a label that identifies a given exchange;

N: a large random number (*nonce*) generated securely by TTP to uniquely identify messages of a given execution of the protocol for a given exchange; and,

H: one-way and collision-resistant hash function: it is not feasible to compute from $H(N)$, N nor another N' such that $H(N) = H(N')$.

Π_x : contents of a message sent by the TTP to P_x to initiate the exchange phase.

When a party $Z \in \{A, B, TTP\}$ sends a message M it also includes $Sig_Z(M)$ as the evidence of origin for M. A recipient accepts a received M only if the accompanying $Sig_Z(M)$ is found authentic. For brevity, we will not make the details of this verification explicit and a received message will only refer to a message received with an authentic evidence of origin; further the pair $\{M, Sig_Z(M)\}$ will simply be written as M which will be indicated by its significant fields.

2.5.2. Exchange Preliminaries

All our protocols are structured into two phases: the *start-up* phase followed by the *exchange phase* when the actual exchange of promised data items takes place. During the start-up phase, the following three steps are carried out:

Step 0.1: Each U_X approaches a trusted authority TA_X to generate a verification procedure using which any party can verify if a data item I satisfies specification Σ_X . For example, if I_B is a piece of software S, U_B must approach a software licensing authority (trusted by U_A) who evaluates S against the specification Σ_B that U_B advertised. If satisfied, the authority computes an evidence of evaluation $E_B = H(S)$; V_B is then

¹ Readers familiar with the FLP impossibility result [FLP85] may find these arguments remarkably similar.

generated as a program which contains E_B and evaluates the predicate $E_B = H(I)$ when invoked to verify a data item I . Similarly, if I_A is an electronic cheque, V_A should similarly be generated by a Bank for the amount specified in Σ_A . Using the conventional notations, the step can be expressed as:

$U_A \rightarrow TA_A: (I_A, \Sigma_A);$	$U_B \rightarrow TA_B: (I_B, \Sigma_B);$
$TA_A \rightarrow U_A: \{V_A, \Sigma_A, \text{Sig}_{TA_A}(V_A, \Sigma_A)\};$	$TA_B \rightarrow U_A: \{V_B, \Sigma_B, \text{Sig}_{TA_B}(V_B, \Sigma_B)\};$

Note that our protocols do not require both TA_A and TA_B to be the TTP itself.

Step 0.2: Users exchange their $\{V, \Sigma, \text{Sig}_{TA}(V, \Sigma)\}$, decide on the TTP and on a (future) time T_{EX} when the TTP should initiate the exchange phase.

Step 0.3: $U_A \rightarrow TTP: (A, B, T_{EX}, \{V_A, \Sigma_A\}, \{V_B, \Sigma_B\}, \text{rtt}_{AB})$; similarly, $U_B \rightarrow TTP: (A, B, T_{EX}, \{V_A, \Sigma_A\}, \{V_B, \Sigma_B\}, \text{rtt}_{BA})$. The last field rtt_{XY} is U_X 's estimation of the round trip time between its node and N_Y .

Upon receiving messages sent in step 0.3, the TTP verifies whether all fields except the last one are identical. If so, the TTP decides on a unique L for the exchange if the exchange is new; generates a nonce N for the exchange phase and records N in a variable $active_run\#(L) = N$. If the exchange is not new, it only generates a new nonce and records it in $active_run\#(L)$. It then sends Π_B to N_A and Π_B to N_B which include the parameters L and $H(N)$ which are used to identify messages of a given execution. The nature and the complete contents of Π 's sent by TTP depend on the exchange phase of the protocol for a chosen combination of various assumptions discussed earlier.

2.5.3. TTP Involvement

A protocol is said to keep TTP *offline* if it is possible for honest user processes to achieve normal termination without interacting with TTP after the exchange phase is initiated. Such a protocol is also called *optimistic* in the literature. If TTP is not off-line, it is said to be *on-line*.

A protocol is said to use a *state-keeping* TTP if it requires the TTP to respond to messages from user processes for an unspecified amount of time; the protocol is said to use a *state-relinquishing* TTP if it requires the TTP only for a specified duration after the exchange phase has been initiated. From cost point of view, TTP is preferred to be offline and state-relinquishing.

3. Non-Fault-Tolerant Fair Exchange Protocols

3.1. Restricted Abuser, Synchronous Communication (P1)

Protocol Principles and Outline

The TTP sends a code Π_A to P_A and Π_B to P_B . P_A and P_B perform the exchange phase of the protocol P1 by executing the code given to them. Embedded in Π_X are $L, H(N), V_X, \Delta$, and keys K_A and K_B , each with TTP's evidence of origin. Δ is set to D (the known bound on message delays), K_A and K_B are symmetric and random session keys. When a dishonest U_X is only a restricted abuser, he cannot obtain K_A and K_B from Π_X nor undetectably modify V_X embedded within Π_X . However, U_X can delay, block, inspect, or tamper with any message P_X generates or is destined to receive while it executes Π_X .

On executing Π_A , P_A first verifies whether I_A (input by U_A) passes V_A embedded within Π_A . Only if I_A passes the verification, P_A continues the execution: it encrypts the verified I_A using K_A , forms a message M_A and encrypts M_A with K_B . The encrypted M_A , like all messages sent by P_A , contains a signature generated using the private key of U_A and is sent to P_B . (Any message without valid signature is ignored and hence assumed not 'received' by a destination.) P_B decrypts the received M_A , and if the contents of correct, it sends an acknowledgement $\text{Ack}_B(A)$ to P_A . P_B similarly sends M_B to, and expects to receive $\text{Ack}_A(B)$ from, P_A . The message exchange between P_A and P_B are depicted in Figure 2, where a process sending a message shown along an out-going arrow is conditional upon that process having received the messages shown along all incoming arrows. P_X , after receiving Π_X , starts executing both the rounds

concurrently: send M_X and wait for $Ack_Y(X)$ while waiting for M_Y before $Ack_X(Y)$ can be sent; a timeout of 2Δ is used to avoid indefinite waiting.

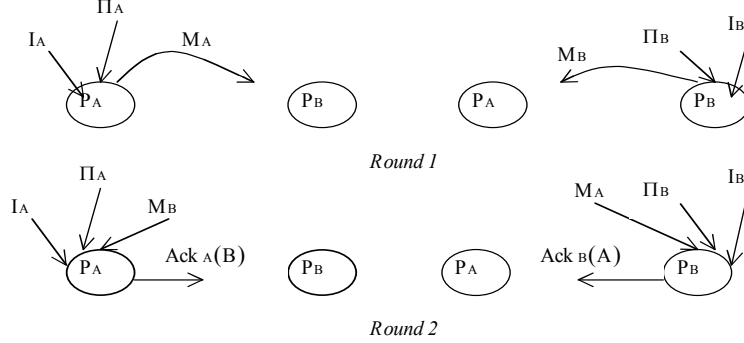


Figure 2. Protocol P1: Message exchange rounds between user processes.

Table 1 summarises the messages used and the intended direction of their flow. A message is indicated by its significant fields, the first three fields will be: the label L, the sender, and H(N). Note that the first and the third message fields uniquely refer to a given execution.

$\Phi_A = eK_A(I_A)$	$\Phi_B = eK_B(I_B)$
$M_A = eK_B(L, A, H(N), \Phi_A); A \rightarrow B$	$M_B = eK_A(L, B, H(N), \Phi_B); B \rightarrow A$
$Ack_A(B) = (L, A, H(N), H(M_B), My_ack); A \rightarrow B$	$Ack_B(A) = (L, B, H(N), H(M_A), My_ack); B \rightarrow A$
$Res_A = (L, A, H(N), M_A, M_B, Resolve_request); A \rightarrow TTP$	$Res_B = (L, B, H(N), M_B, M_A, Resolve_request); B \rightarrow TTP$
$Req_A = (L, A, H(N), M_A, Abort_request); A \rightarrow TTP$	$Req_B = (L, B, H(N), M_B, Abort_request); B \rightarrow TTP$
$M_{TTP}(A) = (L, TTP, H(N), M_B, My_ack); TTP \rightarrow A$	$M_{TTP}(B) = (L, TTP, H(N), M_A, My_ack); TTP \rightarrow B$
$Abort_{TTP}(A) = (L, TTP, H(N), Abort_granted, A); TTP \rightarrow A$	$Abort_{TTP}(B) = (L, TTP, H(N), Abort_granted, B); TTP \rightarrow B$

Table 1. Description of messages used in Protocol P1.

The condition on P_A (P_B) for computing I_B (I_A) from M_B (M_A) is: both M_B (M_A) and $Ack_B(A)$ ($Ack_A(B)$) must be received within 2Δ time after receiving Π_A (Π_B) from TTP. If P_A receives only M_B (within 2Δ time), it asks TTP to *resolve* the exchange by sending message Res_A that contains both M_A and the received M_B . If it receives neither M_B nor $Ack_B(A)$, it *requests* TTP to abort the exchange by sending message Req_A that contains only M_A . TTP, having initiated the exchange at its clock time T_{EX} , executes the following steps at $T_{EX} + 4\Delta$:

Step T1: if it has received Req from both processes or Res from at least one process, it sets variable *outcome* = *resolved* and resolves the exchange by sending $M_{TTP}(X)$ to both processes;

Step T2: if it has received Req from only one process, it sets *outcome* = *aborted* and aborts the exchange by sending $Abort_{TTP}(X)$ to both processes;

Step T3: if it has received no message, it sets variable *outcome* = *unknown*;

Step T4: it terminates (as a state-relinquishing TTP).

An honest P_A terminates normally by receiving either M_B and $Ack_B(A)$ or $M_{TTP}(A)$; exceptionally by receiving $Abort_{TTP}(A)$. The protocol keeps TTP off-line (step T3 above). Appendix A1 argues that the

four properties of fair-exchange (Section 2.4) are met based on the following: (i) dishonest user, say U_B , is a restricted abuser, so he cannot force P_B to deliver I_A against the protocol conditions nor can he obtain K_A and K_B from Π_B ; (ii) an honest P_A sends $\text{Ack}_A(B)$ to P_B only if it receives M_B in a timely manner; (iii) any message that P_A sends to TTP reaches before $T_{\text{EX}}+4\Delta$; and, (iv) TTP's response, if any, is identical to both P_A and P_B .

3.2. Malicious Abuser, Synchronous Communication: Protocol P1.1

P1.1 is the same as P1 except some messages of the exchange phase have different contents. This is because a dishonest user, say U_B , can obtain the keys K_A and K_B in the program Π_B supplied by TTP at the start of the exchange phase. The following changes are needed:

1. Π_A is a message containing all parameters as before except K_B : $\Pi_A = (L, H(N), V_A, \Delta, K_A)$; similarly, $\Pi_B = (L, H(N), V_B, \Delta, K_B)$. Further, TTP should not have used K_A and K_B before.
2. Π_A does not contain K_B means that P_A cannot encrypt M_A with K_B (see Table 1). Therefore M_A is $(L, A, H(N), \Phi_A)$ and M_B is $(L, A, H(N), \Phi_B)$.
3. P_A includes K_A in its $\text{Ack}_A(B)$ so that if P_B has both M_B and $\text{Ack}_A(B)$ it can terminate normally without having to contact TTP. So, $\text{Ack}_A(B)$ of Table 1 becomes $(L, A, H(N), H(M_B), K_A)$ and $\text{Ack}_B(A)$ becomes $(L, B, H(N), H(M_A), K_B)$.
4. Finally, $M_{\text{TTP}}(A)$ by which TTP instructs P_A to resolve the exchange, must contain K_B : $M_{\text{TTP}}(A) = (L, \text{TTP}, H(N), M_B, K_B)$. Also, $M_{\text{TTP}}(B) = (L, \text{TTP}, H(N), M_A, K_A)$.

Correctness: The arguments for *termination* and *non-triviality* remain the same as for P1 since they concern only honest user(s). We claim that the arguments for *no-loss-of-fairness* also hold, due to the following additional reason: honest P_A releases key K_A to P_B only if it receives M_B in a timely manner. The arguments for *non-Repudiation* are also the same for P1, except that the evidence of origin for K_B may come from P_B (in $\text{Ack}_B(A)$) as well.

Remark: *Dispute Resolution:* Non-repudiation guarantee ensures that when honest P_A terminates normally, it has evidence that P_B sent the item which it delivered to U_A . However, if U_B is dishonest, U_A might find I_B not passing the verification test V_B which it agreed with U_B in the start-up phase. Consider this scenario: malicious U_B obtains K_B from Π_B , generates M'_B with authentic evidence of origin but with $I'_B \neq I_B$, and transmits M'_B in place of M_B . Since Π_A does not contain K_B (see modification 1 above), P_B cannot check whether the I'_B in the received M'_B meets V_B at the end of the first round itself (see figure 2). Since M'_B is found to have authentic evidence of origin, it will send $\text{Ack}_A(B)$ that contains K_A , thus letting P_B terminate normally without ever contacting TTP. Only after being delivered of I'_B , U_A can find out that I'_B does not pass V_B and that U_B has been dishonest.

If U_B is a restricted abuser, the above scenario cannot arise: Π_B is tamper-proof relative to the abilities of U_B and therefore it can contain K_A using which P_B encrypts M_B in P1. Since U_B cannot obtain K_A , he cannot modify M_B to M'_B and make P_A accept M'_B . Thus, in P1, what U_A receives is guaranteed to be the expected one.

P1.1 only guarantees that P_A delivers to an honest U_A what a malicious U_B *actually* sent in exchange for I_A , not necessarily what U_B has pledged to send. The TTP can guarantee fairness to U_A in such a scenario, by contacting the trusted agent TA_B employed to generate V_B which U_A accepted. This corresponds to a weaker form of fairness enforcement in the hierarchy of [VPG99]: fairness can only be guaranteed with the help of a trusted agent outside the fair-exchange system (figure 1) and without any cooperation from U_B . Note that the problem of fair-exchange of data items can be reduced into one of fair-exchange of digital signatures over an agreed contract, and the contract signing can be achieved through known abuse-free protocols for fair-exchange of signatures [GJM99, ASW99]. Such an approach also cannot guarantee that post-exchange disputes do not arise: if U_B , after having signed the contract and taken the digital items of U_A , refuses to honour his commitment, then fairness can only be enforced after the contract exchange. The issue of post-exchange dispute resolution is discussed and addressed in [RR00] and [RRN00] for malicious abusers. The protocols of [RR00] and [RRN00] make use of *inverse* and *compatible* keys to

verify an encrypted item without decrypting it. Using these special types of encryption keys, P1.1 can also be modified to ensure that an honest user receives in a normal termination only the item he bargained for. The modification is simple and discussed in Appendix 3. It preserves the 2-round structure which is provably optimal [PSW98]. The protocol of [RR00], like our P1.1, keeps the TTP off-line and requires 4 rounds when both users are honest.

3.3. Restricted and Malicious Abusers, Asynchronous Communication (P1A and P1.1A)

Protocols P1 and P1.1 are not appropriate for the asynchronous model. Impossibility case: Say, both U_A and U_B are honest. Within 2Δ time after receiving Π from TTP, let P_A receive M_B and $Ack_B(A)$, and P_B receive only M_A because $Ack_A(B)$ was unduly delayed. Let Res_B be delayed as well, so it does not reach TTP before $T_{EX} + 4\Delta$ by which time TTP stops responding to any message related to the execution. Honest P_B has no fairness and also cannot terminate.

This impossibility is not surprising as [PSW98] shows that, in the non-fault-tolerant case, there cannot be an asynchronous protocol that works with a TTP that is off-line *and* state-relinquishing, and goes on to present a protocol with an off-line and state-keeping TTP. We here present protocols P1A and P1.1A with an on-line and state-relinquishing TTP, by deriving them from P1 and P1.1 respectively. The derivation is simple: in round 2 (see figure 2), P_X sends $Ack_X(Y)$ to TTP, not to P_Y ; it then waits for either $M_{TTP}(X)$ or $Abort_{TTP}(X)$.

In the start-up phase, TTP should obtain round trip time (rtt) measurements with each P_X (as rtt_A and rtt_B) and estimate $2\Delta = \text{maximum}\{2\Delta, rtt_A, rtt_B, rtt_{AB}, rtt_{BA}\}$.

TTP:

```

when (clock =  $T_{EX}$ ) do {
  send  $\Pi_A$  to  $P_A$ ; send  $\Pi_B$  to  $P_B$ ; Set_of_M  $M\_Bag_L = \{ \}$ ;
  repeat {receive(M); deposit M in  $M\_Bag_L$ ;} until clock <  $T_{EX} + 4\Delta$ ;
  if ( $Ack_A(B) \in M\_Bag_L$  and  $Ack_B(A) \in M\_Bag_L$ )
    then {send  $Ack_B(A)$  to  $P_A$ ; send  $Ack_A(B)$  to  $P_B$ ;} // exchange resolved
    else {send  $Abort_{TTP}(A)$  to  $P_A$ ; send  $Abort_{TTP}(B)$  to  $P_B$ ;} } /* end do

```

Figure 3. Pseudo-code for TTP in protocols P1A and P1.1A.

If honest users re-execute the protocol for a given exchange after every exceptional termination, *non-triviality* is guaranteed if there exists an execution in which the message transfer delays between P_A , P_B and TTP do not exceed the Δ determined by TTP at the start of the exchange phase; i.e., if message transfer delays do not increase during an execution.

4. Crash-Tolerant Fair Exchange Protocols

This section develops crash-tolerant versions of the protocols of the previous section.

4.1. Restricted Abuser, Synchronous Communication (P2)

Two extensions are necessary for protocol P1 to be crash tolerant. *Pessimistic (synchronous) checkpointing*: P_X of an honest user logs every received message and check-points its state before the received message is processed. Thus, in fig. 2, an honest P_X must receive *and* log the messages of the incoming channels before it can send out a message. *State-keeping TTP*: in step T4, the TTP does not terminate the execution, but continues to operate depending on the value of *outcome*: if *outcome* is *unknown*, the TTP executes steps T1 – T3 after a period of 4Δ time; otherwise, it responds to a message from P_X with $M_{TTP}(X)$ or $Abort_{TTP}(X)$, if *outcome* is *resolved* or *aborted* respectively. This modification permits a recovered P_X to terminate. Note that when TTP sets *outcome* to *resolved* or *aborted*, it is irreversible.

4.1.1. Impossibility of making P1 Crash-tolerant

Even with the use of pessimistic logging and state-keeping TTP, P1 cannot become crash-tolerant even if the dishonest user is a restricted abuser. Consider the following scenario. Let N_A crash during the second round shown in figure 2 and before P_A receives $Ack_B(A)$; that is, P_A has logged I_A , Π_A , sending of M_A , and M_B . Let dishonest U_B block all messages P_A sent to P_B but retains a copy of these incoming messages. P_B , having received no message from P_A within 2Δ time, will send Req_B for which TTP will respond by setting *outcome* = *aborted* and sending $Abort_{TTP}(B)$. Say U_B blocks $Abort_{TTP}(B)$ as well and crashes his node. The recovered P_A will find itself in having received only M_B and will send Res_A which, say, reaches TTP after *outcome* was set to *aborted*; TTP's response will be to send $Abort_{TTP}(A)$ to P_A . Let U_B re-boot N_B , delete Π_B from stable-store, and adjust the clock to make it appear as if the exchange phase is being executed for the first time. He replays the arrival of Π_B and of the blocked messages of P_A exactly at those instances when they arrived during the first execution. Since P_B 'receives' both M_A and $Ack_A(B)$, it delivers I_A to U_B .

4.1.1. Outline of Protocol P2

At the core of P1's inability to be crash tolerant is the fact that having both M_A and $Ack_A(B)$ is a sufficient condition for P_B to deliver I_A without consulting TTP at all. To remedy this, we need to add one more round (shown in Fig. 4) to protocol P1 which P_X executes after having executed the first two rounds (concurrently) and check-pointed its state. (The code and correctness reasoning are given in Appendix A2.) In the third round, P_A sends a second acknowledgement $Ack^2_A(B)$ to P_B indicating the reception of $Ack_B(A)$ and M_B .

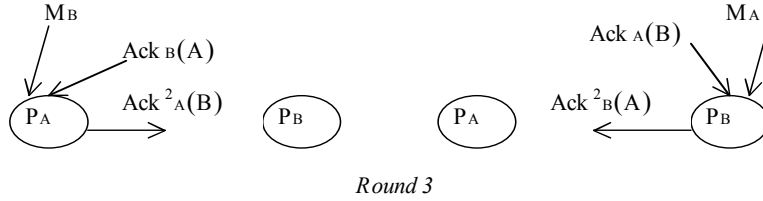


Figure 4. Additional message round for P2.

If P_A does not receive $Ack^2_B(A)$ within 2Δ time after it has sent $Ack^2_A(B)$, P_A appeals to TTP by sending $First_Acks_A$ that contains both $Ack_A(B)$ and $Ack_B(A)$. TTP, after receiving $First_Acks_A$, sends $Ack^2_{TTP}(A)$ to P_A only if it has not earlier received Req_A (which would have resulted in sending $Abort_{TTP}(B)$ to P_B). Thus, the condition for normal termination of P_A is: *received*($Ack^2_{Z_A}(A)$), where $Z_A \in \{B, TTP\}$. By symmetry, the condition for normal termination of P_B is: *received*($Ack^2_{Z_B}(B)$), where $Z_B \in \{A, TTP\}$. Three messages used in addition to those used in P1 are shown in Table 2.

$Ack^2_A(B) = (L, A, H(N), H(Ack_B(A)), My_ack^2)$; $A \rightarrow B$	$Ack^2_B(A) = (L, B, H(N), H(Ack_A(B)), My_ack^2)$; $B \rightarrow A$
$First_Acks_A = (L, A, H(N), Ack_A(B), Ack_B(A))$; $A \rightarrow TTP$	$First_Acks_B = (L, B, H(N), Ack_B(A), Ack_A(B))$; $B \rightarrow TTP$
$Ack^2_{TTP}(A) = (L, TTP, H(N), H(Ack_A(B)), My_ack^2)$; $TTP \rightarrow A$	$Ack^2_{TTP}(B) = (L, TTP, H(N), H(Ack_B(A)), My_ack^2)$; $TTP \rightarrow B$

Table 2. Description of additional messages used in P2.

4.2. Malicious Abuser, Synchronous Communication (P2.1)

Protocol P2.1 is the same as P2, except that some of the messages used need to be changed:

1. Π_A , M_A , $M_{TTP}(A)$, Π_B , M_B , and $M_{TTP}(B)$ of Table 1 change as mentioned in Section 3.2. $Ack_A(B)$ and $Ack_B(A)$ remain exactly as shown in Table 1.

2. In Table 2, $My_ack = K_B$ in $Ack_B^2(A)$ and in $Ack_{TTP}^2(A)$; similarly, $My_ack = K_A$ in $Ack_A^2(B)$ and $Ack_{TTP}^2(B)$.

Remark: *Extension to Asynchronous Communication model.* P2 and P2.1 work correctly in the asynchronous model if TTP computes Δ during the start-up phase as described for P1A and P1.1A.

4.3. Malicious and Restricted Abusers, Asynchronous Communication, on-line TTP (P2A and P2.1A)

These protocols are obtained by making P1A and P1.1A crash-tolerant by having process P_X pessimistically checkpoint its state (as in P2 and P2.1) before processing a received message and by making TTP state-keeping. The latter is done as follows: for any message received from P_X after $T_{EX} + 4\Delta$, TTP sends $Ack_Y(X)$ to P_X if both $Ack_A(B)$ and $Ack_B(A)$ have been received before $T_{EX} + 4\Delta$; otherwise it sends $AbortTTP(X)$.

4.4. Summary

Table 3 summarises the protocols' characteristics using notations: a dishonest user, (DU) who can be either a restricted abuser (RA) or a malicious abuser (MA), the communication delay model (C) can be synchronous (S) or asynchronous (As), faults tolerated (FT) can be none (No) or crash (Cr), and the TTP which can be either off-line (Off) or on-line (On), and either state-relinquishing (Sr) or state-keeping (Sk).

Protocol	DU	C	FT	TTP
P1	RA	S	No	Off/Sr
P1.1	MA	S	No	Off/Sr
P1A(P1.1A)	RA(MA)	As	No	On/Sr
P2	RA	S(As)	Cr	Off/Sk
P2.1	MA	S(As)	Cr	Off/Sk
P2A(P2.1A)	RA(MA)	As	Cr	On/Sk

Table 3. Characteristics of Protocols Developed.

5. Restricting Abuse

Our restricted abuser model requires that a dishonest user be prevented from extracting encryption keys (K_A and K_B) and from modifying verification procedure (V_A or V_B) that are embedded in the code supplied by the TTP. Here we show that how this requirement can be met using the smartcard technology and a protocol of Shoup and Rubin [SR96] designed for secure generation of session keys for distributed processes. This protocol assumes the availability of a TTP and allows P_A and P_B to obtain (at the end of a protocol execution) a shared symmetric key from their respective user's smartcard. Restricting a dishonest user would then mean that the key generated by the smartcards are given to P_A and P_B encrypted with the card's hard-wired long-term key. This means that the cards have to perform some protocol-specific cryptographic operations; more precisely, P_A and P_B execute those operations of protocols P1 and P2 which involve the use of K_A and K_B , by invoking operations on their local smartcard with an appropriate set of parameters which include the encrypted session key. The modification on Shoup and Rubin's protocol – which has been proven correct and implemented [J_98] – is thus very minimal; specifically, the smartcard is preserved to be a stateless probabilistic device.

Figure 5 depicts the Smartcard-based Fair-Exchange system. Each node N_X is attached to a smartcard device to interact with the smartcard C_X of user U_X . We assume that the communication between P_X and the local smartcard device is tamper-proof and synchronous, and that the devices are reliable. We also assume, similar to [SR96], that a smartcard C_X has a hardwired, long-term key K_{1X} which is shared only

with the TTP. A restricted abuser is assumed not to be able to crack any of these long-term keys stored in his card nor to correctly guess the session key from the encrypted form that is supplied to him.

Recall that in protocols P1 and P2, TTP sends Π_A to P_A which contains: $L, N, V_A, \Delta, K_A,$ and K_B . In the smartcard-based system, $L, N,$ and Δ remain the same; instead of $V_A, V_{1A} = eK_{1A}(L, H(N), U_A, V_A)$ is sent (V_{1A} is $(L, H(N), U_A, V_A)$ encrypted with K_{1A}); and, the information necessary for P_A to execute Shoup-Rubin protocol is sent in place of K_A and K_B . P_A (and P_B) execute the protocol and obtain K_{2A} from C_A (and K_{2B} from C_B). The received key K_{2A} (respectively K_{2B}) is the session key K which C_A (also C_B) generates as per Shoup-Rubin protocol, encrypted with K_{1A} (respectively with K_{1B}). (We note here that none of the steps in Shoup-Rubin protocol requires P_X to know or use K itself). Note also that since the TTP knows K_{1A} and K_{1B} , it can obtain K from either K_{2A} or K_{2B} . Thus, the logical TTP, comprising the actual TTP and the smartcard devices (see figure 5), provides all the information which the TTP sends to P_A and P_B in protocols P1 and P2.

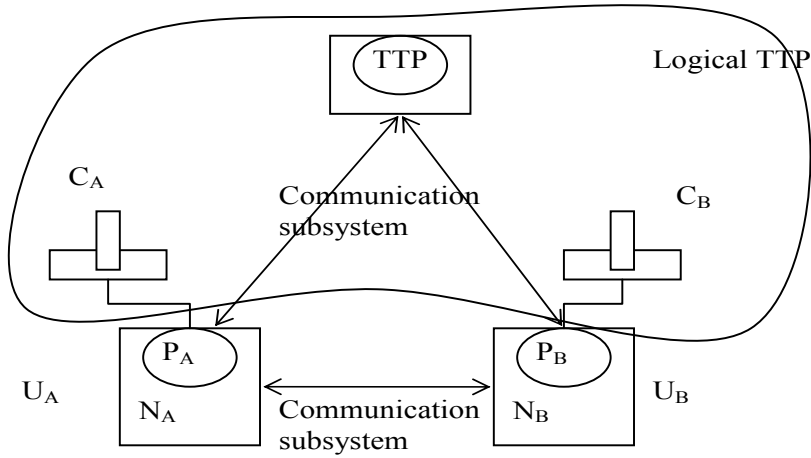


Figure 5. Smartcard Based Fair-Exchange System.

The smartcard C_A supports three operations namely *verify_local()*, *verify_remote()*, and *decrypt()* for P_A to have I_A verified and M_A constructed, to verify the received M_B and to decrypt M_B , respectively:

verify_local(): $P_A \rightarrow C_A: (K_{2A}, V_{1A}, I_A)$; /* note: $V_{1A} = eK_{1A}(L, H(N), U_A, V_A)$.
 $C_A \rightarrow P_A: M_A = eK(L, H(N), U_A, V_A, I_A)$, if $V_A(I_A)$;

verify_remote(): $P_A \rightarrow C_A: (K_{2A}, M_B)$; $C_A \rightarrow P_A: ack_A = eK(L, H(N), U_B, OK)$ if $V_B(I_B)$;

decrypt(): $P_A \rightarrow C_A: (K_{2A}, V_{1A}, M_B, ack_B)$ or $(K_{2A}, V_{1A}, M_B, ack_{TTP})$;
 $C_A \rightarrow P_A: I_B$;

The TTP computes ack_{TTP} for P_A as $eK_{1A}(L, H(N), U_A, OK)$ and ack_{TTP} for P_B as $eK_{1B}(L, H(N), U_B, OK)$, and uses it in place of My_ack and My_ack^2 in protocols P1 and P2 respectively. P_A uses ack_A in place of My_ack and My_ack^2 in protocols P1 and P2 respectively, and includes K_{2A} in all its messages to the TTP (e.g., $Res_A, Req_A, First_Acks_A$).

In the *decrypt()* operation, C_A checks if: the first two fields $\{L, H(N)\}$ of M_B and of ack_B or (ack_{TTP}) , and the first three fields $\{L, H(N), U_A\}$ of V_{1A} , and of ack_B or (ack_{TTP}) are identical.

6. Conclusions and Related Work

We have used smartcards to realise our restricted abuser model. Our smartcards are stateless probabilistic devices. Vogt et. al., developed a fair-exchange protocol [VPG01] using smartcards as stateful, trusted

computing platforms. For example, a smartcard should be able to send signed messages describing its state and be able to authenticate the signed messages it receives.

Message Delay Model and on-line TTP vs. off-line TTP. Liu *et. al.* [LNJ00] proposes a criterion for incorporating crash-tolerance into a non-fault-tolerant protocol by semantics-based message logging scheme which optimises the number of messages that need to be pessimistically (synchronously) logged. The proposed approach is claimed to work for both on-line and off-line TTP based protocols. To demonstrate the effectiveness of the approach, they have considered a protocol with an on-line and state-keeping TTP; this protocol is identical to our P1.1A except that the communication model is taken to be synchronous. In Section 4.1.1, we have shown that the synchronous, offline-TTP protocol P1 cannot be made crash-tolerant merely by pessimistic message logging even if TTP is state-keeping and a dishonest user is a restricted abuser. Another round of message exchange (see figure 4) between user processes was required. This illustrates that message logging alone cannot necessarily make a synchronous protocol crash-tolerant; if a non-fault-tolerant protocol keeps the TTP off-line (as P1 does), then whether the message delay model is synchronous or asynchronous is an important factor that must be considered. The claim that their approach is orthogonal to the underlying delay model (see section 6.2 of [LNJ00]) is true only for online-TTP protocols, and does not hold when TTP is off-line.

On-line TTP: state-relinquishing vs. state-keeping. Can an online-TTP protocol be made crash-tolerant simply through message logging, regardless of whether the delay model is synchronous or asynchronous? The answer depends on whether the non-fault-tolerant protocol keeps the TTP state-relinquishing or state-keeping. Our protocols P1A and P1.1A suggest that it is possible to design online-TTP protocols that keep TTP state-relinquishing even with asynchronous delays. In making them crash-tolerant, we have to make the (on-line) TTP a state-keeping one (see P2A and P2.1A in Table 3). Thus, the answer to the above question is yes, provided the issue of state-relinquishing vs. state-keeping TTP is given due consideration.

Optimising the cost of message logging. The approach suggested by [LNJ00] is as follows. It defines the point of no return for a user process, and if the process would synchronously log all received messages before entering this point, then logging of other received messages can be done asynchronously. Their definition of point of no return suggests that there is only one such point for each process (Section 3.2 of [LNJ00]). We remark here that our protocol P2.1 without check-pointing and message logging is identical to the time-optimal protocol (scheme 3) of [PSW98] which is non-fault-tolerant, for the asynchronous delay model. Applying the proposed approach to make this protocol crash-tolerant does not work: the point of no return for honest P_A turns out to be its sending M_A to P_B (see figure 2, round 1); suppose that N_A crashes in round 3 (see figure 4) after P_A has sent $Ack_A^2(B)$ but before $Ack_B(A)$ and M_B were to be logged by the asynchronous logging scheme; note that $Ack_A^2(B)$ contains K_A (see Section 4.2). Honest U_A suffers loss of fairness, even if U_B does not misbehave; to avoid loss of fairness, P_A must also log all received messages before sending $Ack_A^2(B)$. It appears that some protocols can have *multiple points* at which pessimistic logging is essential. Given that the number of messages received by a user process in a given execution is small, attempts to minimise the overhead of logging may not be worth the effort after all; further, every message received (and logged) may be useful later in any dispute resolution.

References

- [ASW98] Asokan, N., Matthias Schunter and Michael Waidner. Asynchronous protocols for optimistic fair exchange. In Proc. IEEE Symposium on Research in Security and Privacy, pp. 86-99, 1998.
- [ASW99] N. Asokan, V. Shoup and M. Waidner. Optimistic Fair-Exchange of digital Signatures. IBM Research Report, October 1999.
- [BGMR90] M Ben-Or, O Goldreich, S Micali, R L Rivest. A Fair Protocol for Signing Contracts. *IEEE Transactions on Information Theory*, 36(1), pp. 10-46, Jan 90.

- [BGI01] B Barak, O Goldreich, R Impagliazzo, S Rudich, A Sahai, S Vadhan, and K Yang. On the (im)possibility of obfuscating programs (extended abstract). In J. Kilian, editor, *Advances in Cryptology*, CRYPTO '01, Lecture Notes in Computer Science. Springer-Verlag, 2001.
- [CS96] T. Coffey and P. Saidha. Non-Repudiation with mandatory proof of receipt, *Computer Communication Review*, 26 (1), pp. 6-17, Jan 1996.
- [CT00] C S Collberg and C Thomborson, Watermarking, Tamper-Proofing and Obfuscation, Technical Report #170, University of Auckland, New Zealand.
<http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson2000a/index.html>
- [DGLW96] R. H. Deng, L.Gong, A A Lazer, and W.Wang. Practical protocols for certified electronic mail. *Journal of Network and System Management*, 4(3), 1996.
- [FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, April 1985.
- [GJM99] J.A. Garay, M. Jakobsson and P. McKenzie, "Abuse-Free Contract Signing", Proc. *CRYPTO99*, LNCS1666, pp.449-466, 1999.
- [J_98] R. Jerdonek, P.Honeuman, K.Coffman, J.Rees, and K.Wheeler, "Implementation of a provably secure Smartcard-based Key Distribution Protocol", Proc. Third SmartCard Research and Advanced Application Conference, 1998.
- [LNJ00] P. Liu, P. Ning and S. Jajodia. Avoiding Loss of Fairness Owing to Process Crashes in Fair Data Exchange Protocols. Proc. *Int. Conf. on Dependable Systems and Network*. pp. 631-40, June 2000.
- [PSW98] B. Pfitzmann, M. Schunter and M. Waidner. *Optimal Efficiency of Optimistic Contract Signing*. Proc. *ACM Symp. on Principles of Distributed Computing*. New York, 1998.
- [RR00] Indrakshi Ray, Indrajit Ray: An Optimistic Fair Exchange E-commerce Protocol with Automated Dispute Resolution. EC-Web 2000: 84-93.
- [RRN00] Indrajit Ray, Indrakshi Ray, Natarajan Narasimhamurthy: A Fair-exchange E-commerce Protocol with Automated Dispute Resolution. DBSec 2000: 27-38.
- [SR96] V Shoup and A Rubin, "Session Key Distribution using Smart Cards". Proc. of *Advances in Cryptology – EUROCRYPT96*, pp. 321-331, Spain, May 1996.
- [TCPA01] Trusted Computing Platform Alliance, <http://www.trustedpc.org>
- [TTCB01] Service and Protocol Architectures for the MAFTIA Middleware, MAFTIA Deliverable Report D23, Paulo Verissimo, Nuno Neves (eds.), Technical Report DI/FCUL TR-01-1, University of Lisboa, January 2001
- [VPG99] H. Vogt, H. Pagnia, and F. C. Gartner. Modular fair exchange protocols for electronic commerce. In Proc. of the *15th Annual Computer Security Applications Conference*, pages 3--11, Dec. 1999.
- [VPG01] H. Vogt, H. Pagnia, and F. C. Gartner. Using Smart cards for Fair-Exchange. *WELCOM 2001*, LNCS 2232, Springer, pp. 101-113, 2001.
- [ZG97] J. Zhou and D. Gollmann. Evidence and non-repudiation. *Journal of Network and Computer Applications*, 20(3):267-281, 7/1997.

Appendix A1

Protocol P1: Restricted Abuser, Synchronous and non-fault-tolerant

The Protocol

In presenting the protocol, we assume the use of a primitive *timed-receive*(*M*) which blocks until either *M* is received or an associated timer expires. We use *receive*(*M*) as the conventional blocking primitive. Both primitives return *M* only if *M* is received with authentic signature of origin. The Boolean *received*(*M*) returns true if *M* has been obtained through one of these primitives. Procedure *Terminate*(*M*) is used to terminate the execution based on the contents of *M*: if *M* is an *abort-token*, execution is terminated exceptionally; otherwise normal termination results. The pseudo-code executed by P_A is given below. That for P_B can be obtained by inter-changing A and B.

```

PA:
begin
/* PART 1
  {If not (VA(IA)) then exit; // IA does not pass verification, so terminate..
  cobegin      /* spawn two concurrent threads; thread 1:
    { send MA to PB; timer = 2Δ;
      timed_receive(AckB(A));
      if received(AckB(A)) then {store(AckB(A));}
    }
    || /* thread 2:
    { timer = 2Δ; timed_receive(MB);
      if received(MB) then {store(MB); send AckA(B) to PB;}
    }
  coend; /* concurrent threads terminate; act in one of three ways
/* PART 2
#1  if (received(MB) and received(AckB(A)) then // normal termination
    {decrypt ΦB using KB; deliver IB to UA;}
#2  else if received(MB) then
    {send ResA to TTP; receive(M) from TTP; Terminate(M);}
#3  else {send ReqA to TTP; receive(M) from TTP; Terminate(M);}
  }
end;

```

Figure A1.1. The Pseudo-code given to P_A .

Observations

State of P_A	$\{received(M_B), received(Ack_B(A))\}$	State of P_B	$\{received(M_A), received(Ack_A(B))\}$
$S_A(1,1)$	{true, true}	$S_B(1,1)$	{true, true}
$S_A(1,0)$	{true, false}	$S_B(1,0)$	{true, false}
$S_A(0,1)$	{false, true}	$S_B(0,1)$	{false, true}
$S_A(0,0)$	{false, false}	$S_B(0,0)$	{false, false}

Table A1.1. Expressing the states of P_A and P_B when their concurrent threads terminate.

To make some useful observations, let us consider the state of P_X at the end of part 1, i.e., when the concurrent threads of P_X terminate, in a given execution. The state of P_X at this instance of time is expressed in terms of the (Boolean) values the variables *received*(*M_Y*) and *received*(*Ack_Y*(*X*)) evaluate at that time. P_X can be in one of four states identified in Table A1.1; e.g., $S_A(0,0)$ indicates the state of P_A in which *received*(*M_B*) = *false* = *received*(*Ack_B*(*A*)). Let us suppose that U_A is honest and consider the states of P_A and P_B when they complete part 1. The following observations can be made:

O1. If P_A is in $S_A(0,0)$, then P_B can be in either $S_B(0,0)$ or $S_B(1,0)$: since P_A has not received M_B , it is not possible for $received(Ack_A(B))$ to be true for P_B , and hence P_B cannot be in any of $S_B(*,1)$, where $*$ denotes either 1 or 0. Further, if P_A is in $S_A(0,1)$ means that P_B can only be in $S_B(1,0)$. Note that when P_A sends Req_A to TTP, it is in $S_A(0,0)$ or in $S_A(0,1)$ (see case #3 of Fig. A1.1); so, P_B cannot be in $S_B(1,1)$ and cannot execute case #1 of its pseudo-code, i.e., cannot terminate normally without receiving M_{TTP} from TTP.

O2. P_A in $S_A(1,0)$ implies that P_B can be in any one of the four states $S_B(*,*)$. When P_A is in $S_A(1,0)$ it sends Res_A to TTP (see case #2 of Fig. A1.1). So, if P_A sends Res_A , P_B could be sending Req_B , or sending Res_B , or terminating normally without receiving M_{TTP} .

O3. P_A in $S_A(1,1)$ means that P_B cannot certainly be in $S_B(0,*)$. That is, if P_A terminates normally without receiving M_{TTP} , P_B should not be sending Req_B to TTP.

Protocol for TTP

TTP initiates the exchange phase at its clock time T_{EX} by sending Π_A (Π_B) which is executed by P_A (P_B). As stated earlier, Π_X is embedded with L, N, V_X, Δ , and keys K_A and K_B . Δ is set to D for protocol P1. At its clock time $T_{EX} + 4\Delta$, (state-relinquishing) TTP terminates after resolving or aborting the exchange or sending no further message if no message is received from any user process since T_{EX} . Figure A1.2 presents the pseudo-code.

```
TTP:
when (clock =  $T_{EX}$ ) do {
  send  $\Pi_A$  to  $P_A$ ; send  $\Pi_B$  to  $P_B$ ; Set_of_M  $M_{Bag_L} = \{ \}$ ;
  repeat {receive(M); deposit M in  $M_{Bag_L}$ ;} until clock <  $T_{EX} + 4\Delta$ ;
  if  $M_{Bag_L} \neq \{ \}$  /* either resolve or abort the exchange:
    then { if ((received( $Req_A$ ) and (received( $Req_B$ )) or (received( $Res_X$ ):  $X = A$  or  $B$ ))
      then {send  $M_{TTP}(A)$  to  $P_A$ ; send  $M_{TTP}(B)$  to  $P_B$ ;} // exchange resolved
      else if (received( $Req_X$ ):  $X = A$  or  $B$ ) // abort exchange
        then {send Abort $_{TTP}(A)$  to  $P_A$ ; send Abort $_{TTP}(B)$  to  $P_B$ ;}
    };
} /* end do
```

Figure A1.2. Pseudo-code for TTP in protocol P1.

Correctness Reasoning for Protocol Properties:

Let us fix U_A to be honest throughout this subsection. We will assume that the instructions are executed in zero time. (This means that Δ needs to be increased appropriately.) Let us suppose that P_A received Π_A at time T_A as per the TTP's clock; so, $T_A < T_{EX} + \Delta$. The concurrent threads of P_A set the timer to 2Δ . So, any of P_A 's messages to TTP, must have been received by TTP before $T_A + \Delta + 2\Delta < T_{EX} + 4\Delta$. This means that when P_A sends Req_A or Res_A to TTP, it must receive a response from TTP within a finite time.

Termination: P_A sets the timer for 2Δ when it expects to receive a message from P_B . Hence, it does not wait for ever. Any message it sends to TTP reaches TTP before $T_{EX} + 4\Delta$. Hence P_A must terminate the exchange phase within a bounded time.

No Loss of Fairness: If P_A terminates abnormally, U_B cannot receive I_A so long as the contents of Π_B is opaque to a dishonest U_B .

Suppose that P_A terminates exceptionally. This means that P_A cannot have sent Res_A to TTP; if it had, TTP would have received that Res_A before its clock time $T_{EX} + 4\Delta$ and responded with $M_{TTP}(A)$. Therefore, P_A must have sent Req_A and received an abort from TTP. The observation O1 indicates that P_B cannot terminate normally without receiving $M_{TTP}(B)$ from TTP. The code for TTP indicates that if TTP sends abort to P_A , it also sends abort to P_B . Therefore, P_B of honest U_B terminates exceptionally. Say, U_B is dishonest and P_B receives M_A . Since U_B is a restricted abuser, he cannot make P_B compute I_A from the received M_A without the protocol conditions having been satisfied, nor can he guess the keys K_A and K_B . So, U_B does not receive I_A .

Non-Triviality: Suppose that both U_A and U_B are honest. Let P_A and P_B receive Π_A and Π_B at TTP's clock time T_A and T_B respectively. Since clocks of TTP, P_A and P_B are assumed to be perfectly synchronised (assumption A2), $T_{EX} \leq T_X < T_{EX} + \Delta$, $X \in \{A, B\}$; i.e., $|T_A - T_B| < \Delta$; further, T_X is also the local clock time when P_X begins executing the received Π_X . When P_A sends M_A at T_A , M_A is received at N_B before $T_A + \Delta$ but P_B cannot receive M_A before T_B when it starts executing the received Π_B . Since $T_B < T_A + \Delta$, P_B must receive M_A no later than $T_A + \Delta < T_B + 2\Delta = T_B + 2\Delta$. P_B sends $Ack_B(A)$ immediately after receiving M_A . So, P_A must receive $Ack_B(A)$ before $T_A + 2D = T_A + 2\Delta$. This means that the concurrent threads of both P_A and P_B will have *timed-receive()* returning the expected message. So, with TTP being kept off-line, U_A and U_B are delivered of the other's item.

Non-Repudiation: If P_A obtains M_B from M_{TTP} , then it has evidence of origin provided by the TTP; otherwise, it has P_B 's evidence of origin for M_B . It also has TTP's evidence for K_A and K_B using which it computes I_B from M_B .

Appendix A2

Protocol P2: Restricted Abuser, Synchronous and Crash-tolerant

The program executed by P_A has two parts. The first part is the same as for P1 and the second contains the necessary extensions as shown in Figure A2.1. Note that P_A check-points its state before executing Part 2 to ensure that if it completes part 1 without receiving any message from P_B and then crashes, it will know during recovery that it had already completed part 1. This in turn eliminates the following behaviour of P_A : it completes part 1 in $S_A(0,0)$, sends Req_A to TTP in part 2, and crashes; after recovery, it re-executes part 1, receives delayed messages M_B and/or $Ack_B(A)$, and sends to TTP in part 2 a message different to Req_A .

```

PA:
begin {
/* PART 1
/* same as in protocol P1
/* PART 2
check-point state;
if received(MB) and received(AckB(A)) then
{ send AckA2(B) to PB; timer = 2Δ; timed_receive(AckB2(A));
if received(AckB2(A)) then Terminate(AckB2(A))
else {send First_AcksA to TTP; receive(M) from TTP; Terminate(M);}
}
else if received(MB) then
{send ResA to TTP; receive(M) from TTP; Terminate(M);}
else {send ReqA to TTP; receive(M) from TTP; Terminate(M);}
} end;

```

Figure A2.1. Pseudo-code executed by P_A in protocol P2.

Protocol for TTP

Figure A2.1 presents the code for (an off-line) TTP in two parts. Part 1 is similar to TTP code for P1 (see figure A1.2) and the second part makes TTP a state-keeping one. To this end, three Boolean variables *resolved*(L, H(N)), *aborted_A*(L, H(N)), and *aborted_B*(L, H(N)) are maintained. (For brevity, the qualifier (L, H(N)) will be omitted in subsequent descriptions.) The code shown assumes that TTP responds to user process messages until it agrees to a re-execution of the protocol which will set variable *active_run#*(L) for exchange L to a new nonce (see section 2.5.2) in the start-up phase of the next execution.

```

TTP:
begin {
// stable predicates:
boolean resolved(L, H(N)) = false; // becomes true once the exchange is resolved
boolean abortedA(L, H(N)) = false; // becomes true once abort token is given to PA
boolean abortedB(L, H(N)) = false; // becomes true once abort token is given to PB
/* PART 1:
when (clock = TEX) do {
send ΠA to PA; send ΠB to PB; Set_of_M M_BagL = { };
repeat {receive(M); deposit M in M_BagL;} until clock < TEX + 4Δ;
if M_BagL ≠ { }

```

```

    then { if ((received(ReqA) and (received(ReqB)))
            then {send MTTP(A) to PA; send MTTP(B) to PB;
                  resolved(L, H(N)) = true; }
            };
} /* end do
/* PART 2 (making TTP state-keeping)
repeat
  { receive {M};
    case M of
      ReqX:
        {if (resolved(L, H(N))) then send MTTP(X) to PX
          else {send AbortTTP(X) to PX; abortedX(L, H(N)) = true;} }
      ResX:
        {if (not abortedA(L, H(N)) and not abortedB(L, H(N)))
          then {send MTTP(X) to PX; resolved(L, H(N)) = true;}
          else {send AbortTTP(X) to PX; abortedX(L, H(N)) = true;} }
      First_AcksX:
        {if (abortedX(L, H(N)))
          then send AbortTTP(X) to PX;
          else send Ack2TTP(X) to PX;}

    endcase; } //
} until active_run#(L) ≠ N;
} end;

```

Figure A2.2. Pseudo-code for TTP in protocol P2.

Note that the Boolean variables *resolved*, *aborted_A*, and *aborted_B* are stable predicates: once they become true, they remain true forever during an execution. Important points to note are: a distinct *aborted* variable is maintained for each P_X, and TTP's response to a First_Acks_X received depends on whether *aborted_X* is true or not. They are used in correctness arguments below.

Correctness Argument for Protocol Properties:

It is easy to see that P2 meets the *termination*, *non-repudiation* and *non-triviality* guarantees, if the assumption on D (same as Δ) holds. We will show that it also preserves the *No_loss_of_fairness* property despite crashes of an honest user node. Suppose that U_A is honest and P_A terminates exceptionally. We will argue that U_B could not have received I_A so long as U_B cannot deduce K_A, K_B and V_B embedded in Π_B. Let us first make two observations:

- (i) P_A logs every received message before it acts; so, Booleans *received*(M_B), *received*(Ack_B(A)) and *received*(Ack²_B(A)) are stable predicates despite intervening crashes of N_A: once they become true, they remain true in a given execution; and,
- (ii) P_A check-points its state before it begins executing part 2 of its code; so, it is not possible for P_A to send different messages (e.g., Req_A and then Res_A) to the TTP in a given execution, irrespective of the number of times its node may crash during that execution.

P_A can terminate exceptionally only by receiving an abort token from the TTP in response to its sending Req_A, Res_A, or First_Acks_A.

- Say, P_A sent Req_A. This means that it is in S_A(0, *) when it began part 2 of its code. This means that P_B cannot be in S_B(1,1) to be able to produce and send First_Acks_B to the TTP. When Req_A reached TTP, *resolved* must be false; that is, the TTP could not have earlier resolved the exchange nor will resolve it in response to receiving Req_B in future. Further, when TTP sends the abort token, it sets the *aborted_A* to true which is never set back to false. Therefore, the TTP will not resolve any of P_B's Res_B in future. So, P_B cannot deliver I_A to U_B.

- Say, P_A sent Res_A. When Res_A reached the TTP, *aborted_B* must be true, otherwise, the TTP would have resolved the exchange. This means that P_B had already been given an abort token, and that the TTP's response for any future First_Acks_B will be an abort token as well.

- Suppose that P_A sent First_Acks_A . When TTP receives First_Acks_A , aborted_A must already be true. The code of TTP indicates that if TTP sets aborted_X to true, then it must have received Req_X or Res_X . We have shown that P_A checkpointing its state at the end of part 1 of its code forbids it from sending Req_A or Res_A and then First_Acks_A . So, if P_A terminates exceptionally, it could not have sent First_Acks_A to TTP.

Appendix A3

Using inverse and compatible keys in protocols P1.1 and P2.1

U_A and U_B decide on the TTP in step 0.1 and obtain keys K_{0A} and K_{0B} respectively from the TTP which escrows the inverse keys K_{0A}^{-1} and K_{0B}^{-1} .

The key pair $\{K, K^{-1}\}$ has this property: for any m , $eK^{-1}(eK(m)) = eK(eK^{-1}(m)) = m$.

If keys K_1 and K_2 are compatible, then a product key $K_1 \times K_2$ can be obtained with the following properties:

- (i) There exist two large numbers N_1 and N_2 such that:
 - $eK_1 \times K_2(m) \equiv eK_1(\mu) \pmod{N_1}$ if and only if $m = \mu$; and,
 - $eK_1 \times K_2(m) \equiv eK_2(\mu) \pmod{N_2}$ if and only if $m = \mu$;
- (ii) m can be obtained from $eK_1 \times K_2(m)$ using K_1^{-1} or K_2^{-1} if N_1 and N_2 are known.

As in [RRN00], N_1 and N_2 are assumed to be publicly known. We refer the reader to that paper for detailed treatment on inverse and compatible keys. Below, we state the protocol modifications for P_A .

- P_A approaches its T_A with K_{0A} which returns $\text{Sig}_{T_A}(V_A, \Sigma_A)$ with $V_A = eK_{0A}(I_A)$. (See section 2.5.2.)
- It does not receive K_A from the TTP but generates it to be compatible with K_{0A} .
- $\Phi_A = eK_{0A} \times K_A(I_A)$. P_A verifies if Φ_B (received in M_B) and V_B (received in the set-up phase) are encryptions of the same data item.
- In protocol P1.1, P_A uses K_A^{-1} in place of My_ack , and in P2.1 in place of My_ack^2 .
- In protocol P1.1, the TTP uses K_{0B}^{-1} in place of My_ack of $M_{TTP}(A)$, and in P2.1 in place of My_ack^2 of $\text{Ack}_{TTP}^2(A)$.