

School of Computing Science,
University of Newcastle upon Tyne



A Family of Trusted Third Party based Fair-Exchange Protocols

Paul D. Ezhilchelvan and Santosh K. Shrivastava

Technical Report Series

CS-TR-928

September 2005

Copyright©2005 University of Newcastle upon Tyne
Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK.

A Family of Trusted Third Party based Fair-Exchange Protocols

Paul D Ezhilchelvan and Santosh K Shrivastava

School of Computing Science,
University of Newcastle, Newcastle upon Tyne, 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 report presents a family of fair exchange protocols for two participants which make use of the presence of a trusted third party, under a variety of assumptions concerning participant misbehaviour, message delays and node reliability. The development is systematic, beginning with the strongest set of the assumptions and gradually weakening the assumptions to the weakest set. The resulting protocol family exposes the impact of a given set of assumptions on solving the problem of fair exchange. Specifically, it highlights the relationships that exist between fairness and assumptions on the nature of participant misbehaviour, communication delays and node crashes. The report also shows that the restrictions assumed on a dishonest participant's misbehaviour can be realized through the use of smartcards and smartcard-based protocols.

Keywords and Phrases: Fair Exchange, Security, Trusted Third Party (TTP), Smartcards, Crash tolerance, Distributed Systems.

(This Technical Report without the appendices is to appear as a regular paper in the IEEE Transactions on Dependable and Secure Computing.)

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 in the literature (e.g., [ZG96, ASW97, FR97, BDM98]); 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 report we develop a number of fair exchange protocols under a variety of assumptions concerning user misbehaviour, communication delays and node failures. 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 contribution of this report is to highlight the relationships that exist between fairness and the assumptions concerning abuse restriction, communication delays, and node reliability.

This enables a reader to gain a deeper understanding of the impact that a given set of assumptions has on solving the fair-exchange problem. Such an understanding simplifies deriving a protocol for one set of assumptions from those developed with different sets of assumptions. This paper is a revised and extended version of [ES04] which, to the best of our knowledge, is the only paper that comprehensively studied the relationships between such diverse combinations of assumptions and the fair-exchange protocols. The second significant contribution is the development of a restricted abuse model and a realization of that model by making use of the smartcard technology.

Several useful observations are made by comparing our protocols with the related ones in the literature. For example, we note that the use of optimistic message logging for crash tolerance is more subtle than that suggested in [LNJ00] - the first paper to consider fair-exchange with fault-tolerance. Further, contract-signing protocols are observed to transform easily into fair-exchange protocols under our restricted abuse model. This means that many contract-signing protocols presented in the literature (e.g., [ASW98, GJM99]) can be used to derive fair-exchange protocols for different sets of assumptions. These observations constitute the third and final contribution.

The report is organised as follows. We first describe the problem of fair-exchange in detail 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 smartcards in realizing the restricted abuser model defined in section 2. Section 6 surveys the literature for related and similar work and section 7 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 which the other user cannot generate autonomously. User U_x , $X \in \{A, B\}$, advertises that I_x meets specification Σ_x and offers 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 node N_x . Our distributed exchange system (Figure 1) has a third node hosting the trusted third

party (TTP) process. The TTP is assumed to be reliable and secure against intrusions and Trojan horse attacks, and is also trusted by both the users. The exchange preserves fairness as well as non-repudiation. (These properties will be precisely defined shortly.)

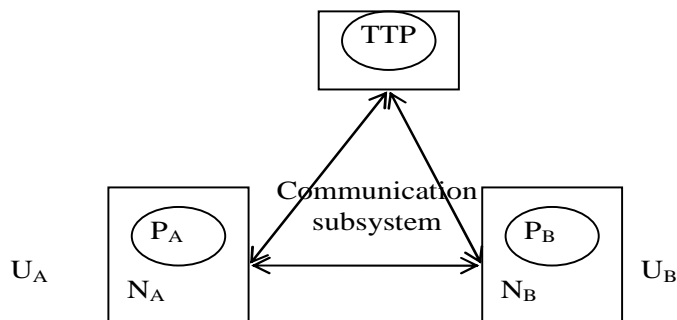


Figure 1. *The 2-user Fair-Exchange System.*

2.1. Classifying User Misbehaviour

Generally, the problem of fair exchange is solved in a context where a dishonest user U_x *totally controls* the behaviour of P_x 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* defined below.

Restricted Abuser:

A dishonest user U_x is a restricted abuser if

- i. P_x can execute a piece of code Π_x which will remain obfuscated to U_x before, during and after the execution;
- ii. U_x can interfere with an execution of Π_x only by crashing the execution platform or by delaying, blocking, or tampering with any message which the execution of Π_x outputs or is destined to receive.

Suppose, for example, that user U_B is a restricted abuser and Π_B contains some secret encryption keys and some input-verification procedures. By the definition above, U_B cannot obtain the encryption keys in Π_B at any time (due to (i)), nor can a modification of Π_B by U_B result in an incorrect input data being verified to be correct (due to (ii)).

We note that code obfuscation, used in software watermarking and tamper-proofing (see [CT00] for a survey), cannot be guaranteed to be totally secure [BGI01]: there exists a non-zero probability that an obfuscated program ceases to be a virtual black box to an abuser. Therefore, realising the restricted-abuse model requires that P_X receive Π_X from the TTP *via* a secure channel and have it executed in a tamper-proof execution platform. In section 5, we describe how smartcards can be used to meet these requirements. We also observe here that there is much interest in developing tamper-proof computing subsystems by the industry led Trusted Computing Group [TCG05]. So, it is of practical interest to develop fair-exchange protocols under the restricted abuse model and to expose thereby any benefits which the model has to offer.

2.2. Classifying Communication Delays

Inter-process communication is assumed to be resilient to network failures and intrusions. This in turn assumes that any message corruption is detected using encryption and reduced to a message loss, and that message losses are tolerated by a bounded number of retransmissions.

Note that a network intruder is here 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 at best block a given transmission or prevent it from being accepted at the destination. We consider two types of network intrusions: intruder gives up delaying a given message transmission after a known period of time or after some unknown 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 unknown (but finite).

2.3. Classifying Node Behaviour

A fair-exchange protocol is fault-tolerant if it ensures fairness and non-repudiation to an honest participant despite the possibility that the participant's node can suffer failures of the assumed type. In other words, guarantees afforded to an honest user do not change when his node is unreliable. We consider two types of behaviour for user nodes:

Reliable: An honest user's node does not fail.

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

The traditional view taken in the TTP-based protocols (e.g., [ZG96, ASW97, ASW98, PSW98, BDM98]) is that the user nodes are reliable; the cause of any node crash is attributed to the user who is seen to have misbehaved and is therefore not entitled to any fair-exchange guarantee. We classify such protocols as non fault-tolerant (see also [LNJ00]). Note that whether the protocol is crash-tolerant or not, the TTP is (assumed to be) reliable and secure.

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: An 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 and the delivered I_Y meets Σ_Y , or
- an *exceptional* termination where P_X informs U_X that the exchange attempt is unsuccessful.

Fairness: If P_X of honest U_X terminates normally and if U_Y is honest, U_Y also terminates normally. If P_X of honest U_X terminates exceptionally, U_Y – honest or not – cannot receive I_X .

When both the users are honest, both are guaranteed to have the same type of termination. (This property is referred to as the *goods atomicity* in [T97].) If only one user is honest and if he has exceptional termination, then the dishonest user cannot receive the expected item.

Non-repudiation: When P_X delivers I_Y to honest U_X , it also provides irrefutable evidence that I_Y was sent by U_Y .

Non-Triviality: When both U_A and U_B are honest, they are guaranteed to have normal termination, provided that certain specified conditions hold.

Without *non-triviality*, the other properties are trivially met if P_A and P_B always terminate exceptionally. Since non-triviality is defined only when both the users are honest, guaranteeing it cannot be directly affected by the abuse model considered, but only by the assumptions chosen regarding the other two aspects. It turns out that when the communication is synchronous and the nodes are reliable, non-triviality is unconditionally guaranteed; for all other combinations of assumptions, as we explain below, some specific sets of conditions need to be satisfied.

In the 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. Consider an honest user or the TTP that is waiting too long for a message from a user process. It cannot resolve whether the source process is honest and the message is delayed due to a crash or is dishonest and is never 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 all. (Similar arguments give rise to the well-known FLP impossibility result [FLP85].) Therefore, meeting the termination property would mean that a protocol execution may have to be terminated exceptionally even if both the users are honest. So, the *necessary condition* for non-triviality is that the honest users re-execute the protocol after every exceptional termination. If each execution is termed as an *attempt*, an honest user must be prepared to make as many attempts as necessary until he has normal termination.

The *sufficient condition* for non-triviality varies with the chosen combination of assumptions: there must be an exchange attempt in which

(i) user nodes do not crash, for the combination of synchronous communication and crash-recovery nodes;

(ii) message delays do not increase during the protocol execution, for the combination of asynchronous communication and reliable nodes; and,

(i) and (ii), for the combination of asynchronous communication and crash-recovery nodes.

2.5. Assumptions, Notations and the Exchange Preliminaries

In all our protocols, the TTP sets up the context and initiates the exchange. Consequently, the protocols are structured into two phases: *start-up* phase (nearly common to all protocols) and *exchange* phase (specific to the selected combination of assumptions). It is assumed that:

A1: Processing within functioning nodes is synchronous. Delays for task scheduling and processing are bounded by a known constant, which, for simplicity, is taken to be zero in relation to communication delays; when a protocol instruction involves computationally intensive operations (e.g., taking a checkpoint), the execution delay is assumed to be counted in the communication delay estimates.

A2: Clocks of functioning nodes are perfectly synchronised. 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

The following notations are frequently used in the paper.

V_A : procedure to verify whether I_A satisfies its description Σ_A advertised by U_A . Similarly, V_B is the procedure to verify whether I_B satisfies its description Σ_B advertised by U_B .

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

$\text{Sig}_X(M)$: signature of P_X , $X \in \{A, B\}$, on M using the private key of U_X ; $\text{Sig}_{\text{TTP}}(M)$: 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 uniquely identifies the exchanging of I_A and I_B .

N : a large random number (*nonce*) generated securely by the TTP to uniquely identify messages of a given attempt.

H : one-way and collision-resistant hash function: it is not feasible to compute from $H(N)$, N nor another N' such that $H(N) = H(N')$. $H(N)$ is smaller in size compared to N , and is therefore used in place of N when the latter needs to be included in a message.

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 $\text{Sig}_Z(M)$ as the evidence of origin for M . A recipient accepts a received M only if the accompanying $\text{Sig}_Z(M)$ is found authentic and if the contents of M have been formed as per the protocol-specific conditions. While the latter will be stated clearly, the verification details are not made explicit in our presentations for reasons of brevity. Thus a *received* message will refer, from now on, only to a message received with an authentic evidence of origin and with appropriate contents, not to the one that was received and found inappropriate or not authentic. Further the pair $\{M, \text{Sig}_Z(M)\}$ is simply written as M which will be indicated by its significant fields. Finally, sending of a message M , say, by P_A to P_B , will be denoted as: $P_A \rightarrow P_B: M$.

2.5.2. Exchange Preliminaries – the start-up phase

The *start-up* phase has two sub-phases, each containing two steps.

Step 0.1: Users decide between themselves the *relinquish time* T_R . This is the time by which they hope to complete the exchange and the TTP is instructed not to keep the exchange-related state information after T_R . They inform the TTP of T_R and, in return, obtain label L and $\text{Sig}_{TTP}(A, B, L, T_R)$ from the TTP. Using the \rightarrow notation, Step 0.1 can be expressed as:

$TTP \rightarrow U_A: \{A, B, L, T_R\};$	$TTP \rightarrow U_B: \{A, B, L, T_R\};$
--	--

Step 0.2: Each U_X approaches a trusted authority TA_X to generate a verification procedure using which any party can verify if I_X satisfies the advertised specification Σ_X . For example, if I_B is a piece of software S , TA_B is a software licensing authority (trusted by U_A) who evaluates S against the specification Σ_B . If satisfied, TA_B computes an evidence of evaluation $E_B = H(S)$; V_B is then 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 . Thus, the step involves:

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

Remarks. TA_A and TA_B also retain the information they receive until time T_R . Note that there is no implication that TA_A and TA_B be the same as the TTP itself. Imposing such a requirement would mean that the functionality of the TTP be quite diverse.

Steps 0.1 and 0.2 need to be re-executed for a new, agreed value for T_R , if all exchange attempts made before T_R terminate exceptionally and if users still want to continue trying. They need not be executed during the second and subsequent exchange attempts before a given T_R . Every exchange attempt involves executing steps 1.1 and 1.2 described below.

Step 1.1: Users first decide between themselves on a (future) time T_{EX} for the TTP to initiate the *exchange* phase. They then exchange what they received from their respective TA and T_{EX} :

$U_A \rightarrow U_B: \{V_A, \Sigma_A, TTP, L, \text{Sig}_{TA_A}(V_A, \Sigma_A, TTP, L, T_R), T_{EX}\};$	$U_B \rightarrow U_A: \{V_B, \Sigma_B, TTP, L, \text{Sig}_{TA_B}(V_B, \Sigma_B, TTP, L, T_R), T_{EX}\};$
---	---

Step 1.2: Users inform the TTP to initiate an exchange attempt at T_{EX} :

$U_A \rightarrow TTP: \{L, A, B, T_{EX}, \{V_A, \Sigma_A\}, \{V_B, \Sigma_B\}, \text{rtt}_{AB}\};$	$U_B \rightarrow TTP: \{L, A, B, T_{EX}, \{V_A, \Sigma_A\}, \{V_B, \Sigma_B\}, \text{rtt}_{BA}\};$
--	--

The last field rtt_{XY} is U_X 's estimation of the round trip time between its node and N_Y . Upon receiving the messages sent in step 1.2, the TTP verifies whether all fields except the last one are identical, L refers to an exchange between A and B , and T_{EX} is future and also 'sufficiently' ahead of T_R . (The minimum expected value for $(T_R - T_{EX})$ varies with the protocols.)

If the verifications are affirmative, the TTP generates a nonce N for the exchange attempt. It then sends Π_A to N_A and Π_B to N_B which include the parameters L and $H(N)$. The nature and the complete contents of Π 's sent by the TTP will vary with the combination of chosen assumptions and will be described as a part of the protocol descriptions.

2.5.3. TTP Involvement

A protocol is said to keep the TTP *off-line* [BDM98] if it is possible for honest user processes to achieve normal termination without interacting with the TTP after an exchange attempt has been initiated. Such a protocol is also called *optimistic* in the literature (e.g., [ASW98]). If the TTP is not off-line, it is said to be *on-line*.

We will say that a protocol uses a *state-relinquishing* TTP if it allows the users to specify some finite value for (the *relinquish* time) T_R in the start-up phase (Step 0.1). As in [PSW98], a protocol is said to use a *state-keeping* TTP if it does not allow a finite value to be specified for T_R in the start-up phase; it may require the TTP to respond to messages from user processes for an unspecified amount of time and hence T_R is expected to be set to ∞ in Step 0.1. From the cost point of view, the TTP is preferred to be off-line and state-relinquishing.

2.5.4. Protocol Naming Notations

Before presenting the protocol family, the following conventions are used for denoting each protocol. PR is the short form for protocol which by default is non-fault-tolerant; CT is indicative of a crash-tolerant protocol. The assumptions regarding abuse model (α) and communication model (χ) are indicated by the first and the second suffix after PR, respectively; i.e., as PR_ $\alpha\chi$ or CT_ $\alpha\chi$.

- $\alpha = R$ or $\alpha = M$ indicates that the abuse is restricted or malicious respectively.
- $\chi = S$ or $\chi = A$ means that the communication is synchronous or asynchronous, respectively.

For example, PR_RS denotes a non-fault-tolerant protocol for restricted abuser and synchronous communication, and CT_MA a crash-tolerant protocol for malicious abuser and asynchronous communication. When there are several protocols for a given set of assumptions, they are numbered as PR_ $\alpha\chi$ _#1, PR_ $\alpha\chi$ _#2 and so on.

3. Non-Fault-Tolerant Protocols

3.1. Protocol (PR_RS) for Restricted Abuser and Synchronous Communication

The TTP sends code Π_A to P_A and Π_B to P_B . P_A and P_B perform the exchange phase of the protocol PR_RS by executing the code given to them. Embedded in Π_X are L , $H(N)$, V_X , Δ , and keys K_A and K_B , each with the TTP's evidence of origin. Δ is set to D (the known bound on message delays – see Section 2.2). K_A and K_B are symmetric and random session keys. Recall that when a dishonest U_X is only a restricted abuser, he cannot obtain K_A and K_B from Π_X nor

undetectedly 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 .

The message exchange between P_A and P_B are depicted in Figure 2, where message shown along an out-going arrow is sent only on the condition that every incoming message shown has been received. More precisely, P_X starts the rounds only after receiving Π_X (from the TTP) and a valid I_X (from U_X); it sends M_X (in round 1), and sends $Ack_X(Y)$ (in round 2) if M_Y is additionally received.

Table 1 summarises the messages sent, together with their destinations. A message is indicated by its significant fields, with the first three fields being the label L , the sender, and $H(N)$. The detailed description of the protocol for P_A is presented below, and that for P_B can be obtained by symmetry.

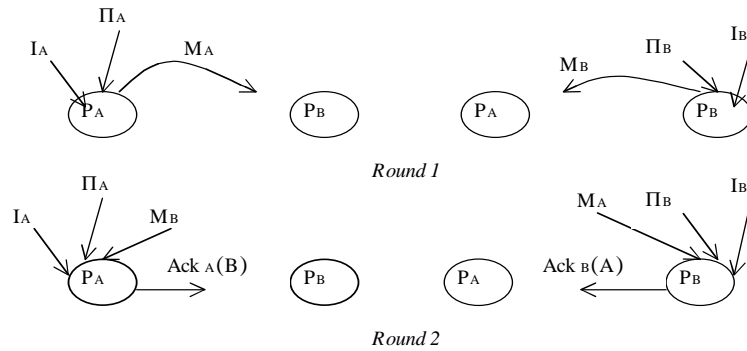


Figure 2. Protocol PR_RS : Message exchange rounds between user processes.

On receiving Π_A , P_A sets a timeout for 2Δ and verifies whether I_A (input by U_A) passes V_A embedded within Π_A . If I_A does not pass the verification, P_A halts the execution. If I_A is verified to be valid, P_A encrypts I_A using K_A and the encrypted item is denoted as Φ_A . It then forms a message M_A whose contents include Φ_A and are encrypted with K_B . (See rows 1 and 2 of Table 1.) M_A is sent to P_B . Similarly, M_B is sent by P_B to P_A .

If P_A receives M_B before the timeout (of 2Δ) expires, it decrypts the received M_B using K_A , and then decrypts the contained Φ_B with K_B to obtain I_B . Recall that P_A is in possession of V_B

(see Step 1.1), and can therefore verify whether I_B satisfies V_B . If I_B is found valid, P_A sends an acknowledgement $Ack_A(B)$ to P_B and waits to receive $Ack_B(A)$ from P_B before the timeout expires. Note that P_A receiving M_B does not lead to resetting of the timeout; it sets the timeout only once and completes both the rounds when or before that timeout expires.

When P_A completes both the rounds, it is in one of the four following states: $S_A(1,1)$ where it has received both M_B and $Ack_B(A)$, $S_A(0,0)$ where it has received neither, $S_A(1,0)$ where it has received M_B not $Ack_B(A)$, and $S_A(0,1)$ where it has received only $Ack_B(A)$. If P_A is in state $S_A(1,1)$, it computes I_B from M_B – which is feasible as it has both the keys K_A and K_B . If it is in $S_A(1,0)$, it asks the TTP to *resolve* the exchange by sending message Res_A that contains both M_A and M_B . (See row 4 of Table 1.) If the state is $S_A(0,1)$ or $S_A(0,0)$, P_A *requests* the TTP to *abort* the exchange by sending message Req_A that contains M_A . (See also row 5 of Table 1.)

No.	Messages from/to P_A	Messages from/to P_B
1	$\Phi_A = eK_A(I_A)$	$\Phi_B = eK_B(I_B)$
2	$M_A = eK_B(L, A, H(N), \Phi_A)$; $P_A \rightarrow P_B$	$M_B = eK_A(L, B, H(N), \Phi_B)$; $P_B \rightarrow P_A$
3	$Ack_A(B) = (L, A, H(N), H(M_B), My_ack)$; $P_A \rightarrow P_B$	$Ack_B(A) = (L, B, H(N), H(M_A), My_ack)$; $P_B \rightarrow P_A$
4	$Res_A = (L, A, H(N), M_A, M_B, Resolve_request)$; $P_A \rightarrow TTP$	$Res_B = (L, B, H(N), M_B, M_A, Resolve_request)$; $P_B \rightarrow TTP$
5	$Req_A = (L, A, H(N), M_A, Abort_request)$; $P_A \rightarrow TTP$	$Req_B = (L, B, H(N), M_B, Abort_request)$; $P_B \rightarrow TTP$
6	$M_{TTP}(A) = (L, TTP, H(N), M_B, My_ack)$; $TTP \rightarrow P_A$	$M_{TTP}(B) = (L, TTP, H(N), M_A, My_ack)$; $TTP \rightarrow P_B$
7	$Abort_{TTP}(A) = (L, TTP, H(N), Abort_granted, A)$; $TTP \rightarrow P_A$	$Abort_{TTP}(B) = (L, TTP, H(N), Abort_granted, B)$; $TTP \rightarrow P_B$

Table 1. Description of messages used in the exchange phase of Protocol PR_{RS} .

T_R specified in Step 0.1 can be finite and the TTP initiates the exchange at its clock time T_{EX} only if $T_R - T_{EX} > 4\Delta$. If the exchange has been initiated at T_{EX} , the TTP initialises the variable *outcome* = *unknown* and executes the following three steps at $T_{EX} + 4\Delta$.

Step T1: if the TTP has received Req from both P_A and P_B , or Res from at least one process, it sets $outcome = resolved$ and resolves the exchange by sending to both the processes a message $M_{TTP}(X)$ whose contents can be seen in the row 6 of Table 1.

Step T2: if the TTP has received Req from only one process, it sets $outcome = aborted$ and aborts the exchange: an abort token $Abort_{TTP}(X)$ is sent to both P_A and P_B . Row 7 of Table 1 shows the contents of $Abort_{TTP}(X)$.

Step T3: The TTP terminates the execution for this exchange attempt.

An honest P_A terminates normally by receiving either M_B and $Ack_B(A)$ or $M_{TTP}(A)$; exceptionally by receiving $Abort_{TTP}(A)$. Appendix A1 presents the pseudo-code and argues that the four properties of fair-exchange (Section 2.4) are met based on the following observations:

- A dishonest user, say U_B , is a restricted abuser. He cannot therefore obtain K_A or K_B from Π_B . Also, he cannot deceive P_B into accepting an I_B for which $V_B(I_B)$ is not true, nor can he force P_B to deliver I_A against the protocol conditions.
- An honest P_A sends $Ack_A(B)$ to P_B only if it receives M_B in a timely manner. So, if U_B obtains I_A , then P_A must be able to send the request Res_A (containing M_B) to the TTP.
- Since P_A completes both the rounds on a timeout of 2Δ , the TTP will receive any message from P_A before $T_{EX}+4\Delta$ as per its clock. The TTP's response is identical to both P_X .

Remarks. *Post-Exchange Allegations.* When P_X terminates normally, the I_Y which U_X receives must meet Σ_Y advertised by U_Y . This is because if U_Y is dishonest, he cannot deceive P_Y into accepting an I_Y for which $V_Y(I_Y)$ is not true. So, if a user alleges that the item he received does not conform to the advertised specification, the allegation is unfounded and warrants no investigation from the protocol's perspective.

Distinct Keys. Observe that the TTP supplies the key pair, K_A and K_B , to both P_A and P_B . A dishonest user cannot deduce any of the keys from the TTP-supplied code Π_X . So, there is no reason why these keys need to be distinct, and a single key, $K \equiv K_A \equiv K_B$, can be used in their place: both P_A and P_B use K to generate Φ_X and encrypt M_X (see rows 1 and 2 of Table 1). We used distinct keys only to simplify the derivation of other protocols from PR_RS.

3.2. Malicious Abuser, Synchronous Communication (PR_MS_#1 and PR_MS_#2)

Derived from PR_RS are two protocols denoted as PR_MS_#1 and PR_MS_#2. In PR_MS_#1, U_X may receive an item that does not conform to the specification Σ_Y advertised by U_Y . Consequently, following a ‘normal’ termination, U_X may need to contact the TTP to resolve a dispute. The TTP restores fairness to an honest user by contacting only the trusted authorities TA_A and TA_B ; specifically, it requires no cooperation from the other user. PR_MS_#2, on the other hand, leaves no room for post-exchange disputes to arise.

3.2.1 Protocol PR_MS_#1 (With Post-Exchange Dispute Resolution)

The time taken to resolve a dispute is assumed to be bounded by a known constant DR . Since the communication model is synchronous, such an assumption is justified provided that dispute resolution terminates, i.e., does not involve dishonest users or crash-prone nodes.

The TTP initiates an exchange only if $T_{EX} + 6\Delta + DR < T_R$. The exchange phase of PR_MS_#1 is derived from PR_RS, accounting for the fact that a dishonest user, say U_B , can now obtain the keys K_A and K_B from Π_B supplied by the TTP. The following four changes are needed in the contents of messages used, and Table 2 presents the full list of the messages:

1. Π_A is a message containing all parameters as in PR_RS except V_A and K_B : $\Pi_A = (L, H(N), \Delta, K_A)$; similarly, $\Pi_B = (L, H(N), \Delta, K_B)$. Further, the TTP should not have used K_A and K_B before.
2. Since Π_A does not contain K_B , M_A is no longer encrypted with K_B as in PR_RS (see row 2 of Table 1). The plain contents of M_A and M_B are shown in row 2 of Table 2.
3. P_A includes K_A in $Ack_A(B)$ (in place of My_ack) so that if P_B has both M_B and $Ack_A(B)$ it can terminate normally without having to contact the TTP. Row 3 of Table 2 shows the modified contents of $Ack_A(B)$ and $Ack_B(A)$.
4. Finally, $M_{TTP}(A)$, by which the TTP instructs P_A to resolve the exchange, now has K_B (again in place of My_ack). Row 6 of Table 2 shows the contents of $M_{TTP}(X)$.

No.	Messages from/to P _A	Messages from/to P _B
1	$\Phi_A = eK_A(I_A)$	$\Phi_B = eK_B(I_B)$
2	$M_A = (L, A, H(N), \Phi_A); P_A \rightarrow P_B$	$M_B = (L, B, H(N), \Phi_B); P_B \rightarrow P_A$
3	$Ack_A(B) = (L, A, H(N), H(M_B), K_A); P_A \rightarrow P_B$	$Ack_B(A) = (L, B, H(N), H(M_A), K_B); P_B \rightarrow P_A$
4	$Res_A = (L, A, H(N), M_A, M_B, Resolve_request); P_A \rightarrow TTP$	$Res_B = (L, B, H(N), M_B, M_A, Resolve_request); P_B \rightarrow TTP$
5	$Req_A = (L, A, H(N), M_A, Abort_request); P_A \rightarrow TTP$	$Req_B = (L, B, H(N), M_B, Abort_request); P_B \rightarrow TTP$
6	$M_{TTP}(A) = (L, TTP, H(N), M_B, K_B); TTP \rightarrow P_A$	$M_{TTP}(B) = (L, TTP, H(N), M_A, K_A); TTP \rightarrow P_B$
7	$Abort_{TTP}(A) = (L, TTP, H(N), Abort_granted, A); TTP \rightarrow P_A$	$Abort_{TTP}(B) = (L, TTP, H(N), Abort_granted, B); TTP \rightarrow P_B$

Table 2. Description of messages used in the exchange phase of Protocol PR_MS_#1.

Correctness: Regarding *termination*, the arguments for PR_RS hold only if both the users are honest. If U_B is dishonest and if P_A terminates ‘normally’, U_A might find the item it received not passing the verification test V_B which it agreed with U_B in the start-up phase. Consider the following scenario. Malicious U_B obtains K_B from Π_B , generates M'_B using $I'_B \neq I_B$, and transmits M'_B in place of M_B . Say, M'_B is received with authentic evidence of origin. So, P_A accepts the received M'_B . Since Π_A does not contain K_B (see modification 1 above), P_A 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 P_A accepts M'_B , it will send $Ack_A(B)$ that contains K_A , thus letting P_B terminate normally without ever contacting the 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. Note that if U_B is a restricted abuser, the above scenario cannot arise, as U_B cannot obtain K_B from Π_B .

In summary, the exchange phase of PR_MS_#1 only guarantees that P_A delivers to an honest U_A what a dishonest U_B *actually* sent in exchange for I_A , not necessarily what U_B has pledged to send. If the former is different from the latter, the TTP restores fairness to U_A by obtaining I_B from the trusted agent TA_B whom U_B employed to generate V_B . This arrangement corresponds to a weaker form of fairness enforcement in the hierarchy of [VPG99]: fairness is guaranteed with the help of a trusted authority and without any cooperation from U_B .

Dispute Resolution. When P_X delivers I to U_X , U_X verifies whether $V_Y(I)$ is true; if $V_Y(I)$ is not true, then U_X sends a dispute resolution call to the TTP with the evidence that P_Y sent M_Y containing the disputed I . The TTP responds to a dispute resolution call from U_X by verifying the evidence supplied, contacting TA_Y to obtain I_Y , and forwarding I_Y to U_X . The step T3 of sub-section 3.1 is modified as below:

Step T3: the TTP terminates execution at $T_{EX} + 6\Delta$, only if no call for dispute resolution is received. If a call is received, it initiates the procedure for dispute resolution.

The arguments for *non-triviality* remain the same as for PR_RS since they concern only when both users are honest. Since dispute resolution restores fairness, the definition of normal termination (in sub-section 2.4) can be weakened to one of U_X receiving an item that either conforms to or *can be made to conform to* the advertised Σ_Y . This means that the arguments of PR_RS for *fairness* also hold here; the arguments for *non-Repudiation* are also the same as for PR_RS, except that the evidence of origin for K_B may come from P_B (in $Ack_B(A)$) as well.

3.2.2. Protocol PR_MS_#2 (No Post-Exchange Dispute)

The core idea behind eliminating the need to handle any post-exchange dispute is to enable P_X to verify an encrypted item without decrypting it. We achieve this, as in the protocols of [RR00] and [RRN00], by making use of *inverse* and *compatible* keys and modifying PR_RS appropriately. The modification is simple and preserves the 2-round structure. The protocol of [RRN00] keeps the TTP on-line and that of [RR00], like PR_MS_#2, keeps the TTP off-line but requires 4 rounds when both the users are honest.

Inverse and Compatible Keys.

The inverse 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:

1. There exist two large numbers N_1 and N_2 such that:

$$eK_1 \times K_2(m) \equiv eK_1(\mu) \pmod{N_1} \text{ if and only if } m = \mu; \text{ and,}$$

$$eK_1 \times K_2(m) \equiv eK_2(\mu) \pmod{N_2} \text{ if and only if } m = \mu;$$

2. 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 [RRN00] for a detailed treatment on inverse and compatible keys. Below, we state the modifications on PR_RS to derive PR_MS_#2.

Step 0.1: In this step, U_A and U_B *additionally* (see also sub-section 2.5.2) obtain keys K_{0A} and K_{0B} respectively from the TTP which escrows the inverse keys K_{0A}^{-1} and K_{0B}^{-1} .

Step 0.2: U_A approaches its TA_A also with K_{0A} ; TA_A computes $V_A = eK_{0A}(I_A)$ and returns its response as in sub-section 2.5.2; similarly, TA_B computes $V_B = eK_{0B}(I_B)$.

The modifications on PR_RS for P_A are as follows, and those for P_B can be derived by substituting the subscripts A and B by B and A respectively.

- P_A does not receive K_A from the TTP but generates it to be compatible with K_{0A} .
- $\Phi_A = eK_{0A} \times K_A(I_A)$. $M_A = (L, A, H(N), \Phi_A)$. (See rows 1 and 2 of Table 3.)
- By the properties of compatible keys, P_A verifies if Φ_B (contained in M_B) and V_B (obtained in the set-up phase) are encryptions of the same data item. If so, it sends $Ack_A(B)$ after including K_A^{-1} in place of My_ack – see row 3 in Table 3.
- As soon as P_A computes I_B after receiving both M_B and $Ack_B(A)$, it verifies whether I_B satisfies Σ_B . If P_B has not included the correct K_B^{-1} in its $Ack_B(A)$, I_B will not satisfy Σ_B . In that case, P_A assumes that it did not receive $Ack_B(A)$ and, as in PR_RS, it sends Res_A to the TTP.
- The TTP includes K_{0B}^{-1} in place of My_ack in its $M_{TTP}(A)$. (See row 4 of Table 3.)

No.	Messages from/to P_A	Messages from/to P_B
1	$\Phi_A = eK_{0A} \times K_A(I_A)$ // P_A generates K_A to be compatible with K_{0A}	$\Phi_B = eK_{0B} \times K_B(I_B)$ // P_B generates K_B to be compatible with K_{0B}
2	$M_A = (L, A, H(N), \Phi_A)$; $P_A \rightarrow P_B$	$M_B = (L, B, H(N), \Phi_B)$; $P_B \rightarrow P_A$
3	$Ack_A(B) = (L, A, H(N), H(M_B), K_A^{-1})$; $P_A \rightarrow P_B$	$Ack_B(A) = (L, B, H(N), H(M_A), K_B^{-1})$; $P_B \rightarrow P_A$
4	$M_{TTP}(A) = (L, TTP, H(N), M_B, K_{0B}^{-1})$; $TTP \rightarrow P_A$	$M_{TTP}(B) = (L, TTP, H(N), M_A, K_{0A}^{-1})$; $TTP \rightarrow P_B$

Table 3. Messages of Protocol PR_RS which get modified for Protocol PR_MS_#2.

3.3. Restricted and Malicious Abusers, Asynchronous Communication (PR_RA and PR_MA)

PR_RS, PR_MS_#1 and PR_MS_#2 use an off-line and state-relinquishing TTP. As per [PSW98], the nature of the TTP makes them inappropriate for the asynchronous model for the following reasons (see also Section 6). In an exchange attempt, an off-line TTP may or may not receive an abort/resolve request from user(s); it will receive no request if users manage to have normal termination without contacting it. When the communication is asynchronous, the TTP cannot know how long a user request, if there is one, will take to arrive. So, the TTP needs to keep the states until it has received a request or it has known that there is no request from the users. The latter is possible only if each user informs the TTP of its normal termination. Since a dishonest user need not inform the TTP and an honest user's message can take an arbitrary amount of time, a finite value for T_R cannot be guaranteed in the start-up phase.

A (contract-signing) protocol with an *off-line and state-keeping* TTP is presented in [PSW98]. We here derive PR_RA and PR_MA_#2 with an *on-line and state-relinquishing* TTP and PR_MA_#1 with an *on-line and state-keeping* TTP. The derivation is from their respective synchronous counterparts, and involves the following three modifications:

- In Step 1.2, the TTP obtains round trip time (rtt) measurements from each P_X (as rtt_A and rtt_B) and estimates $2\Delta = \text{maximum}\{2\Delta, rtt_A, rtt_B, rtt_{AB}, rtt_{BA}\}$. (Note that Δ is included in the Π_X sent by the TTP to signal the start of the exchange phase.)
- Round 1 of the exchange phase remains the same; in round 2 (see Figure 2), P_X sends $\text{Ack}_X(Y)$ to the TTP, not to P_Y , if it is satisfied with the round 1 message it received; it then waits for either $M_{TTP}(X)$ or $\text{Abort}_{TTP}(X)$ from the TTP
- $T_R = \infty$ only for PR_MA_#1 as it is not possible to estimate the bound DR .

TTP:

```

when (clock =  $T_{Ex}$ ) do {
  send  $\Pi_A$  to  $P_A$ ; send  $\Pi_B$  to  $P_B$ ; Set_of_M M_BagL = { };
  repeat {receive(M); deposit M in M_BagL;} until clock <  $T_{Ex} + 4\Delta$ ;
  if (AckA(B) ∈ M_BagL and AckB(A) ∈ M_BagL)
    then {send  $M_{TTP}(A)$  to  $P_A$ ; send  $M_{TTP}(B)$  to  $P_B$ ;} // exchange resolved
  else {send AbortTTP(A) to  $P_A$ ; send AbortTTP(B) to  $P_B$ ;} /* end do

```

Figure 3. Pseudo-code for the TTP in PR_RSA, PR_MA_#1 and PR_MA_#2.

The TTP's code for the exchange phase is presented in Figure 3. Note that the TTP resolves the exchange only if both the processes are satisfied with what they received in the first round; also, $Ack_X(Y)$ acts as a signal to the TTP that P_X is satisfied with the round-1 message it received. $M_{TTP}(X)$ contains appropriate information needed for processes to decrypt the round-1 message. This information is ' My_ack ', K_Y , and K_{0Y}^{-1} for PR_RA, PR_MA_#1 and PR_MA_#2, respectively – exactly as in PR_RS, PR_MS_#1 and PR_MS_#2 respectively. The contents of $M_{TTP}(X)$ are shown below for each protocol. (The messages Res_A , Req_A , Res_B and Req_B of the synchronous protocols are not needed.)

$M_{TTP}(A) = (L, TTP, H(N), H(M_B), My_ack);$ $TTP \rightarrow P_A$ // PR_RA	$M_{TTP}(B) = (L, TTP, H(N), H(M_A), My_ack);$ $TTP \rightarrow P_B$ // PR_RA
$M_{TTP}(A) = (L, TTP, H(N), H(M_B), K_B);$ TTP $\rightarrow P_A$ // PR_MA_#1	$M_{TTP}(B) = (L, TTP, H(N), H(M_A), K_A);$ TTP $\rightarrow P_B$ // PR_MA_#1
$M_{TTP}(A) = (L, TTP, H(N), H(M_B), K_{0B}^{-1});$ $TTP \rightarrow P_A$ // PR_MA_#2	$M_{TTP}(B) = (L, TTP, H(N), H(M_A), K_{0A}^{-1});$ $TTP \rightarrow P_B$ // PR_MA_#2

Correctness:

The properties of *termination*, *fairness* and *non-repudiation* are met for the reasons that (i) P_X terminates only by receiving a message from the TTP and the contents of the TTP's message alone decide the type of termination; and, (ii) the TTP in PR_MA_#1 is state-keeping ($T_R = \infty$) and can therefore restore fairness whenever it is approached by an honest P_X for a post-exchange dispute resolution. If honest users re-execute the protocol 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 the TTP do not exceed the Δ determined by the 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

4.1. Restricted Abuser and Synchronous Communication (CT_RS)

We make PR_RS crash tolerant by incorporating three features: (i) state-keeping TTP, (ii) check-pointing by protocol processes, and (iii) an additional round in the exchange phase.

State-keeping TTP: In step 0.1, T_R is set to ∞ . The TTP executes steps T1 and T2 of Section 3.1 at $T_{EX}+4\Delta$, except that it does not respond to P_X that has sent no message to it. In Step T3, it does not terminate but waits to respond to a message from P_X . The response to P_X is either an abort or a resolve message. (The latter is denoted as $Ack_{TTP}^2(X)$, not as $M_{TTP}(X)$.) Thus, the TTP always helps a recovered P_X to terminate. Appendix 2 presents the TTP's code in detail.

Pessimistic (synchronous) check-pointing: P_X of an honest user logs every received message and check-points its state before the received message is processed. Thus, in Figure 2, an honest P_X receives *and* logs the messages of the incoming channels before it sends out a message.

Even with check-pointing by P_X and state-keeping by TTP, PR_RS is not crash-tolerant. Consider the following scenario. Let N_A crash after $Ack_A(B)$ is sent but before $Ack_B(A)$ is received (see Figure 2). P_A must have logged I_A , Π_A , and M_B and check-pointed the sending of M_A , prior to processing the received M_B . Let dishonest U_B block all messages P_A sent to P_B but retains a copy of them. P_B , having received no message from P_A within 2Δ time, sends Req_B for which the TTP will respond by setting *outcome* = *aborted* and sending $Abort_{TTP}(B)$. Say U_B blocks $Abort_{TTP}(B)$ as well and crashes N_B . Say, P_A recovers meanwhile. Having received only M_B , P_A will send Res_A but the TTP's response to P_A will be to send $Abort_{TTP}(A)$. Let U_B re-boot N_B and adjust the clock to make it appear as if Π_B has just been received. He replays the arrival of the blocked messages from 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 CT_RS – Incorporating the Third Round

To make PR_RS crash-tolerant, we additionally need to make two provisions: (a) when P_B has M_A and $Ack_A(B)$, it can deliver I_A to U_B without consulting the TTP only if it knows that P_A also has M_B and $Ack_B(A)$; and, (b) even after the TTP has aborted the exchange for P_B , it can resolve the exchange for P_A if P_A 'appeals' with valid credentials. A third round (shown in Figure 4) is added to implement (a) and (b).

P_X check-points its state soon after it completes the first two rounds. Only if it has received both M_Y and $Ack_Y(X)$, it enters the third round by sending a second acknowledgement $Ack_X^2(Y)$

to P_Y . (Otherwise, it acts as in PR_RS.) Once P_X enters the third round, it expects to receive $Ack^2_Y(X)$ from P_Y within 2Δ time. If $Ack^2_Y(X)$ is not received until the timeout expires, P_X appeals to the TTP by sending $First_Ack_{S_X}$ that contains both $Ack_X(Y)$ and $Ack_Y(X)$.

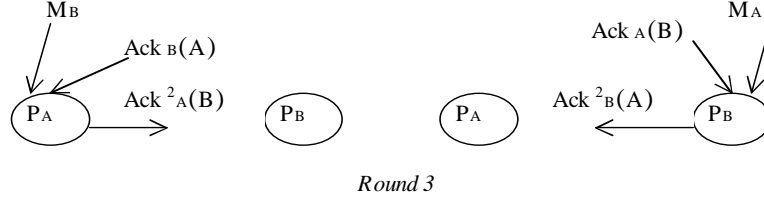


Figure 4. Additional message round for CT_RS.

When the TTP receives $First_Ack_{S_A}$, it sends a resolve message $Ack^2_{TTP}(A)$ to P_A only if it had not earlier sent $Abort_{TTP}(A)$ to P_A . (The decision to send $Ack^2_{TTP}(A)$ to P_A is not influenced by whether or not $Abort_{TTP}(B)$ had been earlier sent to P_B .) The condition for P_A to terminate normally is: $received(Ack^2_{Z_A}(A))$ where $Z_A \in \{B, TTP\}$; that for P_B is: $received(Ack^2_{Z_B}(B))$, $Z_B \in \{A, TTP\}$. The pseudo-code and the correctness arguments are given in Appendix A2. Table 4 shows the three messages used in addition to those used in PR_RS.

No.	Messages from/to P_A	Messages from/to P_B
1	$Ack^2_A(B) = (L, A, H(N), H(Ack_B(A)), My_ack^2)$; $P_A \rightarrow P_B$	$Ack^2_B(A) = (L, B, H(N), H(Ack_A(B)), My_ack^2)$; $P_B \rightarrow P_A$
2	$First_Ack_{S_A} = (L, A, H(N), Ack_A(B), Ack_B(A))$; $P_A \rightarrow TTP$	$First_Ack_{S_B} = (L, B, H(N), Ack_B(A), Ack_A(B))$; $P_B \rightarrow TTP$
3	$Ack^2_{TTP}(A) = (L, TTP, H(N), H(Ack_A(B)), My_ack^2)$; $TTP \rightarrow P_A$	$Ack^2_{TTP}(B) = (L, TTP, H(N), H(Ack_B(A)), My_ack^2)$; $TTP \rightarrow P_B$
4	$M_{TTP}(A)$ not used	$M_{TTP}(B)$ not used

Table 4. Additional Messages of CT_RS.

4.2. Malicious Abuser, Synchronous Communication (CT_MS_#1 and CT_MS_#2)

Protocols CT_MS_#1 and CT_MS_#2 are obtained from PR_MS_#1 and PR_MS_#2 respectively by adding the third round in the same manner employed for obtaining CT_RS from PR_RS. The contents of a few messages need to be changed and these changes arise due to the following reason. Recall that $Ack_X(Y)$ sent in the second round (also the final round) of

PR_MS_#1 and PR_MS_#2 contains keys for decrypting the messages received in the earlier round. These keys must now be exchanged in the third round, and the $Ack_X(Y)$ is sent containing no keys. Also, $Ack_{TTP}^2(X)$ will now include the key that was contained in $M_{TTP}(X)$ as the latter is not used in the crash-tolerant versions. These changes are presented in Tables 5 and 6.

No.	Messages from/to P_A	Messages from/to P_B
1	$Ack_A(B) = (L, A, H(N), H(M_B), My_ack); P_A \rightarrow P_B$	$Ack_B(A) = (L, B, H(N), H(M_A), My_ack); P_B \rightarrow P_A$
2	$Ack_A^2(B) = (L, A, H(N), H(Ack_B(A)), K_A); P_A \rightarrow P_B$	$Ack_B^2(A) = (L, B, H(N), H(Ack_A(B)), K_B); P_B \rightarrow P_A$
3	First_Acks _A = (L, A, H(N), Ack _A (B), Ack _B (A)); $P_A \rightarrow TTP$ // as in CT_RS	First_Acks _B = (L, B, H(N), Ack _B (A), Ack _A (B)); $P_B \rightarrow TTP$ // as in CT_RS
4	$Ack_{TTP}^2(A) = (L, TTP, H(N), H(Ack_A(B)), K_B); TTP \rightarrow P_A$	$Ack_{TTP}^2(B) = (L, TTP, H(N), H(Ack_B(A)), K_A); TTP \rightarrow P_B$
	$M_{TTP}(A)$ not used	$M_{TTP}(B)$ not used

Table 5. Additions to and Changes in the Messages of PR_MS_#1 for deriving CT_MS_#1.

No.	Messages from/to P_A	Messages from/to P_B
1	$Ack_A(B) = (L, A, H(N), H(M_B), My_ack); P_A \rightarrow P_B$	$Ack_B(A) = (L, B, H(N), H(M_A), My_ack); P_B \rightarrow P_A$
2	$Ack_A^2(B) = (L, A, H(N), H(Ack_B(A)), K_A^{-1}); P_A \rightarrow P_B$	$Ack_B^2(A) = (L, B, H(N), H(Ack_A(B)), K_B^{-1}); P_B \rightarrow P_A$
3	First_Acks _A = (L, A, H(N), Ack _A (B), Ack _B (A)); $P_A \rightarrow TTP$ // as in CT_RS	First_Acks _B = (L, B, H(N), Ack _B (A), Ack _A (B)); $P_B \rightarrow TTP$ // as in CT_RS
4	$Ack_{TTP}^2(A) = (L, TTP, H(N), H(Ack_A(B)), K_{0B}^{-1}); TTP \rightarrow P_A$	$Ack_{TTP}^2(B) = (L, TTP, H(N), H(Ack_B(A)), K_{0A}^{-1}); TTP \rightarrow P_B$
	$M_{TTP}(A)$ not used	$M_{TTP}(B)$ not used

Table 6. Additions to and Changes in the Messages of PR_MS_#2 for deriving CT_MS_#2.

4.3. Malicious and Restricted Abusers, Asynchronous Communication, on-line TTP (CT_RA, CT_MA_#1 and CT_MA_#2)

These protocols are obtained by making PR_RA, PR_MA_#1 and PR_MA_#2 crash-tolerant. This is done by (i) having process P_X pessimistically checkpoint its state before processing a received message, and (ii) making the TTP state-keeping. The latter is done as follows: for any message received from P_X after $T_{EX} + 4\Delta$, the TTP sends $M_{TTP}(X)$ to P_X if both $Ack_A(B)$ and

$\text{Ack}_B(A)$ have been received before $T_{\text{EX}} + 4\Delta$; otherwise it sends $\text{Abort}_{\text{TTP}}(X)$. Checkpointing permits a recovered P_X to resume execution correctly and the state-keeping by the TTP allows the recovery time to be arbitrary. Note that the third round is not needed because P_X can terminate only by receiving a message from the (on-line) TTP.

4.4. Summary and Extensions

PR_{RS} is the basic protocol from which we derive every other protocol. Its TTP is offline and state-relinquishing. The restricted abuse assumption was relaxed in two ways: by adding post-exchange resolution procedures ($\text{PR}_{\text{MS}_{\#1}}$) and by the use of compatible and inverse keys ($\text{PR}_{\text{MS}_{\#2}}$). From this core sub-family of triplets, many other triplets are derived as shown in Figure 4, where a circle represents a protocol-triplet for both the abuse models ($\alpha = *$). Note that the bound DR on dispute resolution delays must be known for the TTP to be state-relinquishing; otherwise, T_R needs to be ∞ , i.e., the TTP be state-keeping. (See Sections 3.2.1 and 3.3.)

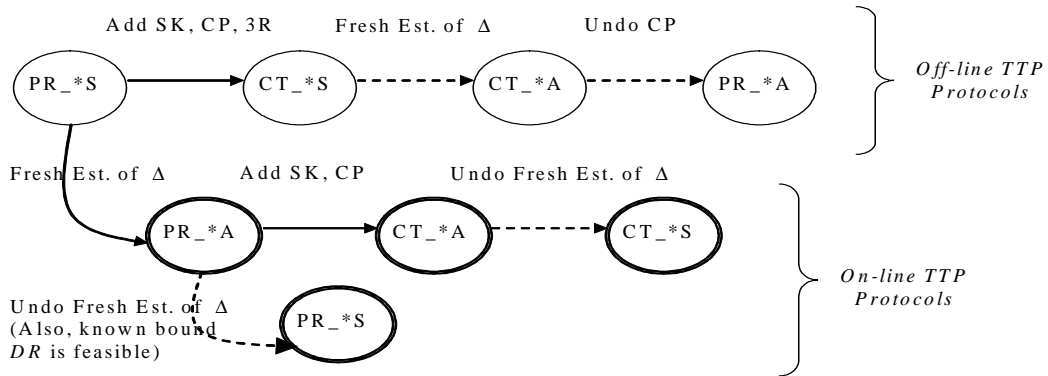


Figure 5. Protocol Derivation Methods.

The triplet PR_{*S} is enhanced in section 3.3 to PR_{*A} by making the TTP online and estimating Δ freshly before each exchange initiation to ensure *non-triviality*. (The protocol-triplets with on-line TTP are indicated by thick circles in Figure 5.) Retaining off-line TTP, PR_{*S} are enhanced to CT_{*S} in sections 4.1 and 4.2. It required adding three features: state-keeping TTP (**SK**), P_X to checkpoint its state (**CP**), and a third round (**3R**). In section 4.3, the

PR_*A (with online TTP) are transformed to CT_*A by adding just SK and CP. Note that adding 3R is not needed when the TTP is on-line and solely dictates how P_X should terminate.

The dotted arrows in Figure 5 represent derivations not described in the earlier sections. Consider first the protocols with offline TTP (shown in the upper half of Figure 5). Deriving CT_*A from CT_*S is straightforward for the following reason. A recovered node resuming the protocol execution after an arbitrary amount of repair-time appears to its environment as a reliable node whose response is arbitrarily delayed. So, CT_*S will cope with network asynchrony so long as Δ is estimated freshly before each exchange initiation to ensure *non-triviality*. Reduction of CT_*A to PR_*A removes check-pointing (CP) since nodes do not crash. For protocols with online TTP, reducing CT_*A to CT_*S and PR_*A to PR_*S would mean that the fresh evaluation of Δ is not necessary as the bound D on message delays is known in the synchronous model. Further, the bound DR can also be known. So, the post-exchange dispute resolution no longer requires the TTP to be state-keeping in PR_*S, since nodes do not crash.

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) which are embedded in the code (Π_A or Π_B) supplied by the TTP. In this section, we show that how the model of malicious abuser can be reduced to one of restricted abuser. We make use of the smartcard technology and also a smartcard-based protocol of Shoup and Rubin [SR96] designed for secure generation of session keys for distributed processes. The latter has been proven correct in [SR96, B03] and its implementation can be seen in [J_98]. (For an easy understanding of the protocol, we refer the reader to a deconstructed version in [B03].) Our attempt at restricting abuse involves a small modification on the use of the original Shoup and Rubin protocol but preserves the smartcard to be a stateless probabilistic device.

Figure 6 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, as in [SR96], that C_X

has a hardwired, long-term key K_{1X} stored in it, and is shared only with the TTP; U_X is assumed not to be able to guess this key stored in his own card. We also assume that the smartcard devices are tamper-resistant and do not crash. Finally, the communication between P_X and the local smartcard device is assumed reliable (i.e., what is sent is received uncorrupted) and synchronous.

The principles behind abuse restriction are two-fold. Shoup and Rubin protocol assumes the availability of a TTP which, upon being requested by one of the users, initiates an execution of the protocol. At the end of the execution, the user processes (P_A and P_B in our case) obtain a shared symmetric key (say, K) from their local smartcard C_X . In our system, a dishonest user U_X could control P_X . So, C_X returns to P_X the K encrypted with its hard-wired long-term key K_{1X} . (This is the only modification on the use of Shoup and Rubin protocol.) The shared K encrypted with K_{1X} is denoted as K_{2X} . Note that the TTP knows K_{1X} for $X= A$ or B , and hence it can know K , given K_{2X} . Thus, the TTP, C_A and C_B together form the *logical TTP* for PR_RS and CT_RS in the following sense. The TTP and C_X enable P_X towards using a key K that can also be used by P_Y . U_X cannot deduce K since P_X receives and stores K only in the encrypted form - encrypted with K_{1X} that U_X cannot access.

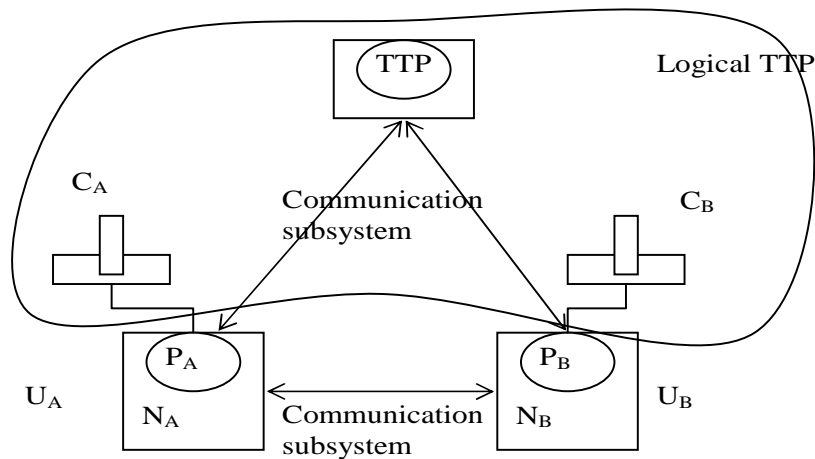


Figure 6. Smartcard Based Fair-Exchange System.

Secondly, the supplied K is used in place of K_A and K_B . (See also the remark *Distinct Keys* in Section 3.1.) Since P_A and P_B do not know K , they cannot perform the protocol-specific

cryptographic operations, such as forming Φ_X , by themselves. So they have such operations done for them by the local C_X . To this end, C_X supports three APIs. Moreover, to preserve C_X to be a stateless device, these APIs are invoked by P_X with all relevant information so that C_X does not have to store any protocol specific information (including the K it generated). The three APIs are *verify_local()*, *verify_remote()*, and *decrypt()*. They enable P_A (i) to have I_A verified, (ii) to have M_A constructed, and (iii) to verify and decrypt M_B , respectively. They are described below.

5.1. Modifications and Additional Support

Recall that in protocols PR_RS and CT_RS, the TTP sends Π_A to P_A which contains: L , $H(N)$, V_A , Δ , K_A , and K_B . In the smartcard-based system, L , $H(N)$ and Δ remain the same; instead of V_A , $V_{1A} = eK_{1A}(L, H(N), U_A, V_A)$ is sent; and, the information SR_A necessary for P_A to execute Shoup-Rubin protocol is sent in place of K_A and K_B . Thus, $\Pi_A = (L, H(N), V_{1A}, \Delta, SR_A)$ and $\Pi_B = (L, H(N), V_{1B}, \Delta, SR_B)$. P_A and P_B execute the protocol which results in P_A receiving K_{2A} from C_A and P_B receiving K_{2B} from C_B , if processes execute the protocol properly and do not prematurely timeout on each other.

verify_local() :

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

Using its hard-wired key K_{1A} , C_A decrypts the first and the second input parameters to obtain K and $(L, H(N), U_A, V_A)$, respectively. If the third input parameter (I_A) passes V_A , then C_A constructs M_A with the contents encrypted with K and returns M_A . The contents of M_A are those obtained in the decryption of the second input parameter (V_{1A}) and the verified I_A .

verify_remote() :

$P_A \rightarrow C_A: (K_{2A}, M_B);$
 $C_A \rightarrow P_A: \text{ack}_A = eK(L, H(N), U_B, OK) \text{ if } V_B(I_B), \text{ or } \text{ack}_A \text{ otherwise;}$

C_A obtains K from K_{2A} ; using K , it decrypts M_B to $(L, H(N), U_B, V_B, I_B)$. If I_A passes V_A , C_A returns ack_A .

If P_A receives ack_A , it sends $\text{Ack}_A(B)$ to P_B in protocol PR_RS with ack_A in place of *My_ack* (see row 3, Table 1); in protocol CT_RS, $\text{Ack}_A(B)$ sent will be as in Table 1 and ack_A will replace

My_ack^2 in $Ack^2_A(B)$ (see row 1, Table 4). If, on the other hand, P_A receives $nack_A$, it assumes that it did not receive M_B and acts as per the protocol – i.e., sends Req_A to the TTP. The messages Res_A , Req_A and $First_Acks_A$ will additionally include K_{2A} as shown in Table 7.

If the TTP decides to resolve the exchange after the due verification process, it computes $ack_{TTP}(A)$ for P_A which is the same as $ack_B = eK(L, H(N), U_A, OK)$; similarly, and $ack_{TTP}(B)$ for P_B is the same as $ack_A = eK(L, H(N), U_B, OK)$. It uses $ack_{TTP}(X)$ in place of My_ack in $M_{TTP}(X)$ for PR_RS (see row 6, Table 1); for CT_RS , it uses $ack_{TTP}(X)$ in place of My_ack^2 in $Ack^2_{TTP}(X)$ (see row 3 of Table 4). The resulting contents of $M_{TTP}(X)$ and $Ack^2_{TTP}(X)$ are shown in rows 4 and 5 of Table 7 respectively. (Recall that $M_{TTP}(X)$ is not used in CT_RS .)

No.	Messages from/to P_A	Messages from/to P_B
1	$Req_A = (L, A, H(N), M_A, Abort_request, K_{2A}); P_A \rightarrow TTP$	$Req_B = (L, B, H(N), M_B, Abort_request, K_{2B}); P_B \rightarrow TTP$
2	$Res_A = (L, A, H(N), M_A, M_B, Resolve_request, K_{2A}); P_A \rightarrow TTP$	$Res_B = (L, B, H(N), M_B, M_A, Resolve_request, K_{2B}); P_B \rightarrow TTP$
3	$First_Acks_A = (L, A, H(N), Ack_A(B), Ack_B(A), K_{2A}); P_A \rightarrow TTP$ /* used only in CT_RS	$First_Acks_B = (L, B, H(N), Ack_B(A), Ack_A(B), K_{2B}); P_B \rightarrow TTP$ /* used only in CT_RS
4	$M_{TTP}(A) = (L, TTP, H(N), M_B, ack_{TTP}(A)); TTP \rightarrow P_A$ /* used only in PR_RS	$M_{TTP}(B) = (L, TTP, H(N), M_A, ack_{TTP}(B)); TTP \rightarrow P_B$ /* used only in PR_RS
5	$Ack^2_{TTP}(A) = (L, TTP, H(N), H(Ack_A(B)), ack_{TTP}(A)); TTP \rightarrow P_A$ /* used only in CT_RS	$Ack^2_{TTP}(B) = (L, TTP, H(N), H(Ack_B(A)), ack_{TTP}(B)); TTP \rightarrow P_B$ /* used only in CT_RS

Table 7. Description of messages used when abuse is restricted by smartcards.

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

In the $decrypt()$ operation, C_A checks (i) if the first three fields $\{L, H(N), U_A\}$ of V_{1A} , and of ack_B or $ack_{TTP}(A)$ are identical, and (ii) if the first two fields $\{L, H(N)\}$ of M_B and of ack_B or $ack_{TTP}(A)$ are identical. If (i) and (ii) are verified to be true, C_A decrypts M_B and returns I_B .

6. Related Work and Discussions

To the best of our knowledge, this and [ES04] are the only papers that considered the restricted abuser model in solving the fair-exchange problem and also to show how smartcards can be used to realise that model. The smartcards and the related protocols essentially build a *trusted computing base* (TCB) within an untrustworthy host. Such an abstraction is assumed in [AGGV04] to solve a multi-party fair-exchange problem for the synchronous communication model. The smartcards are stateless probabilistic devices, and are retained so while we use them to realize our restricted abuse model. Vogt et. al., on the other hand, develop a 2-party fair-exchange protocol [VPG01] using smartcards as state-keeping, trusted computing platforms. For example, a smartcard should be able to construct signed messages describing its state and be able to authenticate the signed messages it receives, say, from a Bank.

Incorporating Crash Tolerance. The paper by Liu et. al. [LNJ00] is the first and perhaps the only other work we know of, to undertake the task of transforming non-fault-tolerant protocols into crash-tolerant ones. The authors propose a 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. Moreover, their approach is claimed to be orthogonal to the underlying delay model (see section 6.2 of [LNJ00]). To substantiate these claims, the authors have considered a protocol with an *on-line and state-keeping* TTP; this protocol is identical to our PR_MA_#1 except that the communication model is taken to be synchronous. The protocols we have developed here indicate that the issue of incorporating crash-tolerance is far more subtle than the claims of [LNJ00] and is governed by a combination of several factors. Referring to Figure 5, the following can be observed when a non-fault-tolerant protocol is made crash-tolerant:

- When the TTP is on-line, it must be made state-keeping (SK) if it is not already, in addition to CP. This ensures that a recovering honest node can learn the outcome of an exchange from the TTP and terminate. Also, our protocol family shows that there can be non-fault-

tolerant protocols with a state-relinquishing TTP; that is, there is no reason to suppose that a non-fault-tolerant protocol will only have a state-keeping TTP.

- If the TTP is off-line, the protocol itself may have to be modified. In Section 4.1.1, we have shown that adding CP and SK alone is not sufficient for offline-TTP protocols PR_*S to be made crash-tolerant; 3R is also needed.
- Whether the TTP is on-line or off-line, the delay model does matter if non-triviality is to be guaranteed. Asynchronous model requires the TTP to estimate Δ freshly before an exchange initiation to cope with increase in message transfer delays.

The semantics-based message logging scheme by [LNJ00] operates 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. The structure of CT_RS however indicates that a strategy based on *point of no return* alone is not sufficient to make an off-line protocol crash-tolerant; it should involve process check-pointing as well. In CT_RS, P_X check-points its state once it completes the first two rounds. (See sub-section 4.1.1 and also Appendix 2.) Check-pointing ensures that P_X records what, if any, was not received when the first two rounds ended. This in turn ensures that the post-recovery behaviour is consistent with the pre-crash behaviour: if an action based on the non-arrival of an expected message has been carried out in the pre-crash execution, then the arrival of that message after recovery is ignored.

There are no definitive arguments in [LNJ00] to assert that all protocols will have only a single *point of no return*. If some protocols can have *multiple points* at which pessimistic logging or check-pointing is essential, then attempts to minimise the overhead of logging may not be worth the effort after all, given that the number of messages received by a user process in a given execution is small.

Contract-signing Protocols. The 2-round structure of PR_RS of Section 3.1 is identical to that of a contract-signing scheme (Scheme 2) of [PSW98]. Similarly, Scheme 3 of [PSW98] is identical in structure to protocol PR_RA with the off-line TTP which, as per Figure 5, can be derived from

CT_RS (of Section 4.1) by adding a fresh estimation of Δ prior to each exchange initiation, and by removing check-pointing and message logging. In contract-signing, two users compute a contract as a non-repudiable agreement over a pre-agreed contractual text such that even if one user misbehaves either both or none obtain a contract. The signing process is structured so that if a user is deemed to misbehave after a certain point in the execution, the TTP generates that user's signature [GJM99] or a valid alternative [ASW98]. Such counter-measures by the TTP are simplified by the fact that the contractual text is known to the users prior to the signing process, unlike the item I_X which is a secret for U_Y prior to the exchange process. Because of this difference, contract-signing is a weaker version of the fair-exchange problem and can always be solved (by default) through fair-exchange of signatures. For this reason, the impossibility results on contract-signing apply to fair-exchange; also, schemes 2 and 3 of [PSW98] assume malicious abuser, whereas PR_RS and PR_RA with the off-line TTP require that the abuser be restricted.

These structural similarities and the reasons provided below suggest that any contract-signing protocol can lead to a fair-exchange protocol being derived for the restricted abuse model. In contract-signing, the messages that P_A sends to P_B (until a certain point in the execution) are useless to a dishonest U_B , because the received message cannot by itself form a contract and the contractual text within it is not a secret. Similarly, with the restricted abuse assumption, the messages that P_A receives from P_B are useless to U_B (though they may contain I_A): they are a black-box to U_B who cannot force P_B to decrypt them against the protocol conditions. Further, the signatures generated by the TTP as a counter-measure in contract-signing has its equivalence in the TTP sending a *resolve* message ($M_{TTP}(X)$ or $Ack_{TTP}^2(X)$) and thereby having P_X decrypt Φ_Y and deliver I_Y to U_X .

We believe that the rich variety of contract-signing protocols in the literature and their analyses (e.g. [CMSS03]) can help derive new families of fair-exchange protocols and a deeper understanding of them. For example, Scheme 1 of [PSW98], [GJM99] and [ASW98] present *asymmetric* protocols which assign non-identical roles to each user, whereas all the protocols in the family developed here are symmetric.

7. Concluding Remarks

The paper comprehensively investigated the problem of TTP-based, 2-party fair-exchange, by considering three significant factors that influence the solution space. The first factor is concerned with a dishonest user's ability to interfere with the protocol execution. In the restricted abuser model, a dishonest user cannot obtain the keys embedded in the code supplied by the TTP and also cannot force the protocol process to operate against the protocol conditions. In the worst case, he can unintentionally, rather than maliciously, block or delay the messages exchanged and crash his node. Though the protocols for this model have the same time and message complexity as the others, they warrant no post-exchange dispute resolution nor require sophisticated cryptographic keys, such as compatible and inverse keys. We also describe a scheme by which the restricted abuser model can be realized through the use of smartcards and related protocols.

The other two factors concern the ability to estimate a bound on communication delays between the parties and the possibility of an honest user's node crashing in the middle of a protocol execution. The family of protocols illustrates the impact of various assumptions that can be made regarding these factors. For example, we identify that the post-exchange dispute resolution involving (only the TTP and the Trusted authorities) can be eliminated in one of the following two ways: reducing the malicious abuse model to one of restricted abuse (using code obfuscation or smartcard-based approach of section 5), or using sophisticated keys (as in PR_MS_#2). Also, it was observed that several issues must be considered while transforming a non-fault-tolerant protocol into a crash tolerant one; particular attention needs to be paid to the question of whether the TTP of the non-fault-tolerant protocol is off-line or on-line.

Acknowledgements

This work was funded in part by the UK Engineering and Physical Sciences Research Council, under e-Science programme grants: Trusted Coordination in Dynamic Virtual Organisations, GOLD: Grid-based Information Models to Support the Rapid Innovation of New High Value Added Chemicals, and by the European Union under IST-2001-34069: "TAPAS (Trusted and QoS-Aware Provision of Application Services).

Discussions with Nick Cook helped to improve many of our initial ideas.

References

- [ASW97] N. Asokan, M. Schunter and M. Waidner, "Optimistic protocols for Fair Exchange", In the proc. of 4th *ACM Conference on Computer and Communications Security*, April 1997, pp. 8-17.
- [ASW98] N. Asokan, M. Schunter and M. Waidner, "Asynchronous protocols for optimistic fair exchange", In Proc. *IEEE Symposium on Research in Security and Privacy*, pp. 86-99, 1998.
- [AGGV04] G. Avoine, F. Gartner, R. Guerraoui and M. Vuolic, "Gracefully Degrading Fair-Exchange with Security Modules", in the Proc. of the 5th *European Dependable Computing Conference (EDCC05)*, Springer Verlag, LNCS 3463, pp. 55-71.
- [BDM98] F. Bao, R. Deng and W. Mao, "Efficient and Practical Fair-Exchange Protocols with Off-line TTP", In Proc. *IEEE Symposium on Research in Security and Privacy*, 1998, pp. 77-85.
- [B03] G. Bella, 'Inductive Verification of Smart Card Protocols', *Journal of Computer Security*, 11(1), 2003, pp. 87-132.
- [BGMR90] M. Ben-Or, O. Goldreich, S. Micali and 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*, LNCS 2139, Springer-Verlag, 2001, pp. 1 -18.
- [CMSS03] R. Chadha, J.C. Mitchell, A. Scedrov, V. Shmatikov, "Contract Signing, Optimism, and Advantage", 14th *International Conference on Concurrency Theory (CONCUR)*, LNCS, 2761, 2003, pp. 366-382.
- [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>
- [ES04] P. D. Ezhilchelvan and S. K. Shrivastava, "Systematic Development of a Family of Fair Exchange Protocols", In the Proc. of the *17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*, 2003, (Eds. Sabrina De Capitani di Vimercati, Indrakshi Ray and Indrajit Ray), Kluwer Academic Press, ISBN 1 4020 8069 7, 2004, pp. 243-258.
- [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.

- [FR97] M. K. Franklin and M. K. Reiter, "Fair-Exchange with a Semi-trusted Third Party", In the proc. of 4th *ACM Conference on Computer and Communications Security*, April 1997, pp. 1-7.
- [GJM99] J.A. Garay, M. Jakobsson and P. McKenzie, "Abuse-Free Contract Signing", Proc. *CRYPTO99*, LNCS 1666, pp.449-466, 1999.
- [J_98] R. Jerdonek, P. Honeyman, K. Coffman, J. Rees, and K. Wheeler, "Implementation of a provably secure Smartcard-based Key Distribution Protocol", Third *SmartCard Research and Advanced Application Conference*, 1998, pp. 229-235.
- [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*, PODC98, New York, 1998, pp. 113-122.
- [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, N. 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". In U. Maurer, Editor, *Advances in Cryptology – EUROCRYPT96*, LNCS 1070, pp. 321-331, Springer-Verlag, 1996.
- [T97] J. D. Tygar, "Atomicity in Electronic Commerce", in *Internet Besieged*, (Ed D. Denning and P. Denning), ACM Press, Addison Wesley, 1997, pp. 389-405.
- [TCG05] Trusted Computing Group, <http://www.trustedcomputinggroup.org/>, 2005.
- [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*, Dec. 1999, pp. 3-11.
- [VPG01] H. Vogt, H. Pagnia and F. C. Gartner, "Using Smart cards for Fair-Exchange", *WELCOM 2001*, LNCS 2232, Springer, 2001, pp. 101-113.
- [ZG96] J. Zhou and D. Gollmann, "A Fair Non-repudiation Protocol", In the Proc. of *IEEE Symposium on Research in Security and Privacy*, pp. 55-61, 1996.

Appendix A1

Protocol PR_RS: 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 and with appropriate contents. 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 interchanging 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:

Observation 1. If P_A is in $S_A(0,0)$, then P_B can be in either $S_B(0, 0)$ or $S_B(1, 0)$. This is because P_A has not received M_B . Hence, $received(Ack_A(B))$ cannot be true and P_B cannot be in $S_B(*, 1)$, where $*$ denotes either 1 or 0.

Observation 2. If P_A is in $S_A(0,1)$ then P_B can only be in $S_B(1, 0)$. 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.

Observation 3. 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 the 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} .

Observation 4. 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 the TTP.

Protocol for TTP

The 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 PR_RS. At its clock time $T_{EX} + 4\Delta$, the (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  $M_{BagL}$  = { };
  repeat {receive (M); deposit M in  $M_{BagL}$ ;} until clock <  $T_{EX} + 4\Delta$ ;
  if  $M_{BagL} \neq \{ \}$  /* 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 the TTP in protocol PR_RS.

Correctness Reasoning for Protocol Properties:

Let us fix U_A to be honest throughout our reasoning. 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 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 the TTP, must have been received by the TTP before $T_A + \Delta + 2\Delta < T_{EX} + 4\Delta$. This means that when P_A sends Req_A or Res_A to the TTP, it must receive a response from the 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 the TTP reaches the TTP before $T_{EX} + 4\Delta$. Hence P_A must terminate receiving a response from the TTP within a bounded amount of time.

Suppose that P_A terminates normally and delivers an item I to U_A . It obtains Φ_B by decrypting M_B with K_A and then I by decrypting Φ_B with K_B . P_B forms Φ_B and sends M_B only if the I_B (supplied by U_B) satisfies V_B which it received directly from the TTP. The V_B that the TTP encloses in Π_B is identical to the V_B that U_A approved of in step 1.3 (the exchange initiation request step). Even if U_B is dishonest, he is a restricted abuser (see definition in Section 2.1). Therefore, $I = I_B$ and U_A must find $V_B(I)$ true, that is, U_A must find I meeting Σ_B .

Fairness:

1. If P_A terminates exceptionally, U_B cannot receive I_A so long as the contents of Π_B is opaque to a U_B .
2. If P_A terminates normally, honest P_B also terminates normally.

Part 1. Suppose that P_A terminates exceptionally. This means that P_A cannot have sent Res_A to the TTP; if it had, the 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 the TTP. The observation 2 indicates that P_B cannot terminate normally without receiving $M_{TTP}(B)$ from the TTP. The code for the TTP indicates that if the 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 even though P_B receives M_A . Thus, U_B , honest or not, could not have received I_A .

Part 2. Suppose that P_A terminates normally. We need to show that if U_B is honest, P_B also terminates normally. The arguments for it follow from those provided below for *Non-Triviality*. They show that when both U_A and U_B are honest, U_A and U_B are certainly delivered of the each other's item.

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 the TTP's clock time T_A and T_B respectively. Since clocks of the TTP, P_A and P_B are assumed to be perfectly synchronised (assumption A2, Section 2.5), $T_{EX} \leq T_X < T_{EX} + \Delta$, $X \in \{A, B\}$; 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 $T_A + \Delta$. 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 the 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 the TTP; otherwise, it has P_B 's evidence of origin for M_B . It also has the TTP's evidence for K_B using which it computes I_B from M_B .

□

Appendix A2

Protocol CT_RS: Restricted Abuser, Synchronous and Crash-tolerant

The program executed by P_A has two parts. The first part is the same as for PR_RS and the second part 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 the 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 the TTP in part 2 a message different to Req_A .

```

PA:
begin {
/* PART 1
/* same as in PR_RS
/* 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 the TTP; receive(M) from the TTP; Terminate(M);}
    }
else if received(MB) then
  {send ResA to the TTP; receive(M) from the TTP; Terminate(M);}
else {send ReqA to the TTP; receive(M) from the TTP; Terminate(M);}
} end;

```

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

Protocol for the TTP

Figure A2.1 presents the code for (an off-line) TTP in two parts. Part 1 is similar to the TTP code for PR_RS (see figure A1.2) and the second part makes the 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 the TTP responds to messages from P_X until it receives a valid request from the users to initiate another exchange attempt (Step 1.2 of sub-section 2.5.2). The variable *active_run#* (L) for exchange L is assumed to hold the nonce for the latest exchange attempt.

```

the 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 AckTTP2(A) to PA; send AckTTP2(B) to PB;
                    resolved(L, H(N)) = true; }
            };
  } /* end do
  /* PART 2 (making the TTP state-keeping)
  repeat
    { receive {M};
      case M of
        Reqx:
          {if (resolved(L, H(N))) then send AckTTP2(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 AckTTP2(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 AckTTP2(X) to PX;}
      endcase; } //
    } until active_run#(L) ≠ N;
  } end;

```

Figure A2.2. Pseudo-code for the TTP in protocol CT_RS.

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 , it is possible for $aborted_X$ to be true while $aborted_Y$ is false, and the TTP's response to a $First_Acks_X$ received depends on whether $aborted_X$ is true or not. These points are used in correctness arguments below.

Correctness Arguments for Protocol Properties:

Termination: As in PR_RS, P_X of an honest U_X waits on a timeout whenever it awaits a message from P_Y , a response from the TTP always arrives. Hence, it must terminate in an exchange attempt.

Non-triviality is guaranteed when U_A and U_B are honest and their nodes do not crash in an exchange attempt. (See sub-section 2.4.) When there is no node crash, the arguments (for non-triviality) are the same as in PR_RS.

Non-repudiation: The arguments for PR_RS hold here as well.

Fairness: Let us fix U_A to be honest throughout.

Part 1. Suppose that P_A terminates exceptionally. We will argue that U_B , honest or not, 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 three observations:

- P_A logs every received message before it acts; so, Booleans $received(M_B)$, $received(Ack_B(A))$ and $received(Ack_B^2(A))$ are stable predicates despite intervening crashes of N_A : once they become true, they remain true in a given execution; and,
- 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 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 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 the TTP. When Req_A reached the TTP, $resolved$ must be false; that is, the the TTP could not have earlier resolved the exchange nor will resolve it in response to receiving Req_B in future. Further, when the TTP sends the abort token, it sets the $aborted_A$ to true which is never set back to false. Therefore, the 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 the TTP, $aborted_B$ must be true, otherwise, the the TTP would have resolved the exchange. This means that P_B had already been given an abort token, and that the 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 the TTP receives $First_Acks_A$, $aborted_A$ must already be true. The code of the TTP indicates that if the 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 the TTP.

Part 2. Suppose that P_A terminates normally. Assume to the contrary that the honest U_B terminates exceptionally. The arguments of Part 1 indicate that when an honest user (here U_B) terminates exceptionally, the other user process (here P_A) cannot terminate normally. But, by given, P_A terminates normally. This is a contradiction. So, honest U_B must also terminate normally.

□