Trusted Delivery Layer for Electronic Commerce

Amir Herzberg
Computer Science Dept.
Bar Ilan University
http://amir.herzberg.name

Abstract

We propose a trusted delivery layer, to provide secure certified messaging services to electronic commerce applications, ensuring both fairness and notarization (timeliness). Our specifications and analysis use a simple yet well-defined model of randomized I/O machines and real time adversarial configurations and runs, which may be useful for other tasks and protocols. Our protocol is efficient and practical, and realistic in its requirements on delays and clock-synchronization.

Keywords: *secure electronic commerce; non-repudiation; timestamp; certified delivery; notarized delivery; certified mail; notarization; e-banking*

1. Introduction

Many business and commercial processes use trusted third parties (TTP) to secure sensitive transactions, where they deliver or exchange information, payments and goods. There are two main motivations:

- **Fairness:** ensure each party provides its component of the deal; components are typically goods, payments or signed obligations (contracts). Typically, both parties give their agreed-upon component to the trusted third party (often referred to as *escrow service*), who then forwards component to the appropriate recipients. This prevents one party from receiving the component from the other party, but not sending its component as agreed.
- **Non-repudiation:** one or both parties receive a proof (certification, evidence) for certain properties of the transaction, such as identities of the parties and time of transaction. Such proofs are often referred to as. The proofs can be validated by one or more trusted third parties (`judge`), and their production also often involves a trusted third party, often called a Notary (or post office, etc.).

With the increased importance of (secure) electronic commerce, many researchers investigated protocols for providing fairness and non-repudiation services. Some of the most important services are:

- **Timestamping –** where the TTP provides a proof that a document existed at specific time. In particular, by time-stamping a document signed with a particular public key, the TTP provides a proof that the document was signed while that public key was (yet) valid, allowing a recipient of the document to prove the identity of the signer even after the signer's key expires or is revoked.
- **Fair exchange (of messages)** – where two parties exchange some valuable messages, using the TTP to prevent one party from receiving the message from its peer but not sending its own. There must be some way for the TTP to validate that the messages have the agreed-upon value/contents. One important special case is **Contract signing,** which is essentially fair exchange of signatures; here the messages exchanged are well-defined digital signatures (with specific contents and public keys). Another important special case is **fair exchange of content for payment,** where one party (a repository) sends content to the other party, in exchange for payment; here the payment is well defined (usually using digital signatures of some sort); the content should also be well defined (e.g. by its hash).
- **Certified messaging (email)** – where a *sender* sends a message to the *receiver*; the protocol provides the sender and/or receiver with one of the following proofs: *Proof Of Origin* (allowing receiver to prove it received the document from the sender), *Proof Of Delivery* (allowing sender

to prove that the receiver received the document) and *Proof Of Submission* (allowing the sender to prove it sent the document to the receiver at specific time, even if the receiver failed to receive). We refer to the `proofs` by the acronyms *POO, POD,* and *POS,* respectively.

Our contributions. We argue that many secure e-commerce applications should be built on top of a `trusted delivery layer`, which will provide common fairness and non-repudiation services. We present specifications, and a very efficient, practical protocol, for certified messaging. We believe our protocol is the most efficient[1] known protocol providing fair certified mail service (with POS, POD and POO, and `optimistic` - not involving the TTP[2] in typical executions); we believe it is a good choice for the basic trusted delivery service. Our specifications and protocol include Proof Of Delivery (POD), which the sender receives if and only if the receiver receives the message together with a Proof Of Origin (POO); and Proof Of Submission (POS), which the sender *always* receives – even if the receiver is faulty.

Another contribution of this work is in presenting a precise yet realistic model of faults, message delays and clock synchronization, and, using it, well-defined real-time specifications. Most of the existing works in this area do not provide rigorous model, specifications and analysis. However, this may lead to weaknesses, as shown in the more rigorous works of [ASW00, PSW00]. Real-time specifications are more realistic than purely synchronous or asynchronous models, as in [PSW00] and [ASW00] respectively. Furthermore both works provide only fairness (fair exchange of signatures in [ASW00] and of POD vs. a POO in [PSW00]), while we provide timestamped Proof Of Submission, even for faulty receiver.

Much of the architecture, message types etc. used to implement our protocol, could be used to carry more advanced protocols when necessary. In particular, the fair exchange protocols of [ASW00,PC*03] could be used to support applications unaware of the protocol (or TTP); it uses essentially the same flows and logic as our protocol (although with some differences in the cryptographic details, and with additional, substantial overhead). Other services may be supported by changing only the content of some of our messages, e.g. ensuring abuse-freeness [GJM99].

Related Works. There have been many works on fairness and non-repudiation services, we only provide partial and concise comparison; for more bibliography of non-repudiation protocols, see [KMZ02,Z01].

An underlying delivery layer for secure e-commerce applications is part of the architecture of [LPSW00], who also addressed the initialization process, mostly focusing on the legal aspects; they considered only fairness goals. There are also several messaging systems and standards offering fair delivery services, in particular [X.400,MSP96], and ISO standards [ISO13888-1,ISO13888-3].

Much of the research on fair delivery focuses on avoiding the use of a TTP, by probabilistic and/or gradual exchange, as in [G82, EGL85]; these works are applicable only to special applications, mainly due to their high overhead and probability of failure; and certainly they cannot provide timestamping services.

Our work follows the `optimistic` approach, i.e. involves the TTP only to handle exceptions. Many works study optimistic fair delivery protocols, e.g. see [ASW97, ASW00, KMZ02, M97, PSW00, Z01, ZG96,

[1] In typical executions, our protocol uses only one signature computation and one signature validation from sender and receiver, and only three messages (flows). The most efficient known protocol, not involving the TTP in typical executions, is the one sketched in sections 8.3 and 10.1 of [ASW00]; it requires an additional signature by the sender, but is otherwise very similar to our protocol (in design and performance). Notice that their protocol provides `accountability`; in section 4.5.3 we sketch how to add accountability (with no extra public key operations).

[2] We use the acronym TTP for the Trusted Third Party. We focus on TTP providing trusted (certified, notarized, fair) delivery services; another appropriate name would be a Trusted Delivery Authority, but since so many names are used in the literature, we use the general TTP name rather than introduce yet another name.

PC*03]; these works avoid the involvement of the TTP in typical, fault-free executions[3]. Our protocol belongs to this category, and shares the basic structure of [PC*03], which is also that of the protocol sketched in sections 8.3 and 10.1 of [ASW00]; however, it is the only protocol that provides also timeliness properties (it is also more efficient).

The timeliness properties of the protocol imply, in particular, that the `proofs` produced include a digitally signed *time-stamp.* The timestamp allows the receiver (sender) to prove to a third party, e.g. judge, that the message was sent (respectively, received) by a particular sender (respectively, receiver) during a specific time interval. The signatures we use in the timestamps are simply those of the sender (respectively, receiver), except if the receiver fails, in which case the timestamp is by the TTP (on the POS). We also discuss in Section 4.6 an extension to the protocol where the timestamps are always provided by the TTP (this requires online TTP, of course), which protects the parties expired or revoked signature verification keys. This is more efficient than having the parties contact a Time Stamping Authority, e.g. using the Time Stamp Protocol (TSP) [RFC3161].

Our focus is on the most basic trusted delivery requirements, which we believe can satisfy many secure e-commerce applications. Most of the research on fairness is addressing stronger requirements, which may be required for specific applications. In particular, most works on fair exchange and contract signing, e.g. [ASW97, ASW00], ensure fair exchange of documents with pre-defined validation criteria, such as a signature by a pre-defined public key (of the other party) over a specific message. For example, a buyer can exchange his signed e-cash for an insurer's signed policy. Similarly, using the protocol of [HPS01], a buyer can exchange payment for a predefined content from a vendor (e.g. repository of free or licensed content). Our protocol, in contrast, only provides fair certified messaging, requiring the application to use a specific (policy/cash) signature validation process; we show extensions to support exchange of contracts, or exchange of payments for content, but still requiring specific validation process. However, applications using fair exchange require awareness of the validation process, and rarely use externally-generated signatures. In fact, we believe that in most applications, the `value` exchanged requires more validation than just validating signatures, e.g. to handle replays and expiration of messages and keys/certificates. Our protocol achieves substantial performance, simplicity and flexibility (e.g. crypto-primitive independence) advantages over the more general fair exchange protocols.

Organization. In Section 2, we provide an informal overview of the applications, requirements and protocol. In Section 3, we present the model and specifications. In section 4 we present the protocol, including extensions e.g. to reduce the trust required in the TTP, timestamp the origin/receiver public keys and exchange signatures (of contracts, payments) and content. In section 5 we discuss implementation issues, and in section 6 we conclude.

2. Overview

We propose to provide support for common trusted-delivery services in a Trusted Delivery Layer, which will be placed between the applications (application layer) and the transport layer (e.g. TCP). The Trusted Delivery Layer will provide both fairness and non-repudiation services. Notice that the trusted delivery layer does not need to ensure communication confidentiality and integrity; in fact it can assume such services are provided, by standard protocols such as Transport Layer Security (TLS/SSL) [R00,RFC2246] or Internet Protocol Security (IP-sec) [RFC2411]. See the layers in Figure 1.

[3] Since the TTP is not involved `online` for typical, fault-free transactions, these protocols are sometimes called `offline`; we prefer the (more common) term `optimistic`.

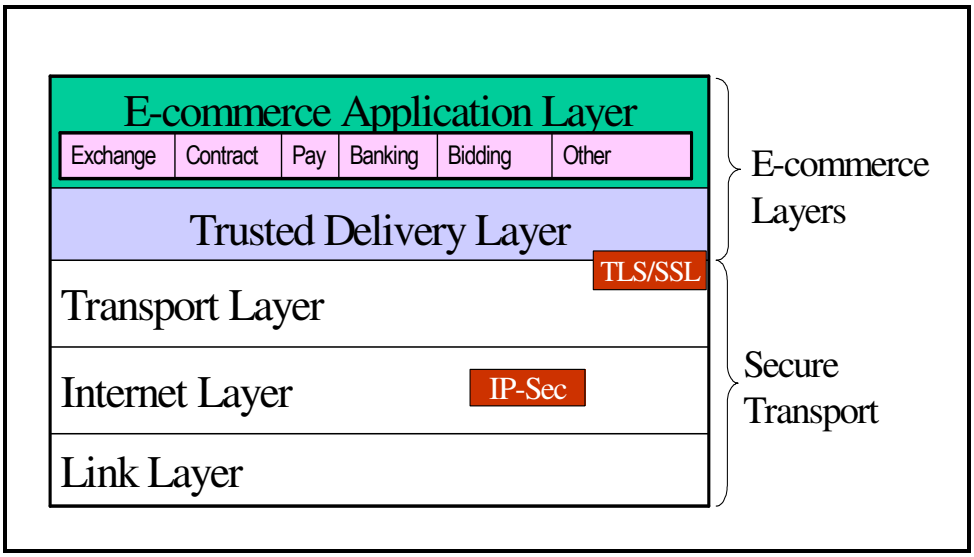

Figure 1: Secure E-Commerce and Transport Layers

We next discuss typical and important e-commerce applications that can use the Trusted Delivery Layer, to identify their fairness and non-repudiation requirements. Next, we present a high-level overview of the flows of the protocol.

2.1. Applications

A crucial business relationship is that between a client who owns some property or funds, and a financial or other institution (bank, broker, trust, etc.) that keeps the property on behalf of the client, as in **Figure 2**. The client may wish to provide orders to the server (institution) regarding its account (property), or to deposit more funds (e.g. by check) to the account.

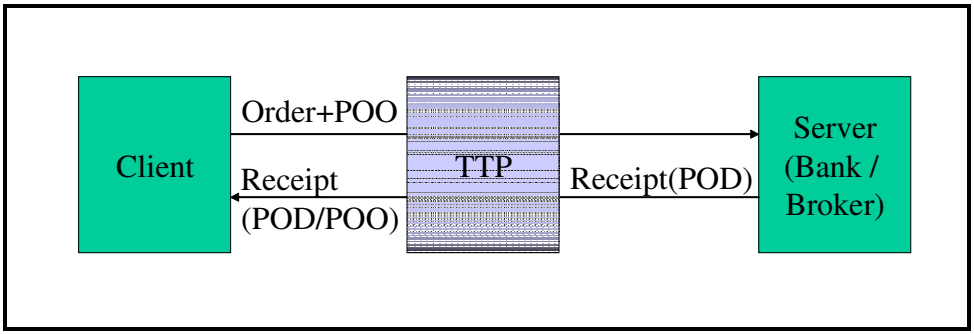

Figure 2: Electronic Banking / Brokerage Application

In traditional banking, orders are always signed, to allow the bank to prove that they were given by the client. Trusted delivery provides the digitized equivalent, which we call *Proof Of Origin (POO),* and typically includes a digital signature using the private key of the client. The proof of origin usually also includes the date and time, to avoid disputes regarding the time (e.g. for interest calculation).

Clients usually require a receipt for the order/deposit, to allow the client to prove that it gave the order or made the deposit, at a specific time. In traditional, face-to-face banking, the receipt is usually provided by the bank; we call such a receipt *Proof Of Delivery (POD).* In remote banking, and in particular electronic banking (and commerce in general), clients usually send their orders and deposits thru a trusted delivery service, which we call a *Trusted Third Party (TTP).* Both client and server (bank) trust the delivery service to reliably forward messages between them. Furthermore, clients are often concerned about availability, e.g. for brokerage services; in this case, the client may require a receipt of the submission of the order,

indicating time of submission, even if the server is unable to receive the order; we call this kind of receipt *Proof Of Submission (POS).*

Another basic application is the exchange of committed (signed) documents, or of other goods and funds. In this case, the parties are typically very concerned about the *fairness,* or atomicity, of the transaction. Namely, we want the sender to have Proof Of Delivery (POD), e.g. proving the receiver's agreement to the contract, *if and only if* the receiver has the message (contract) and Proof Of Origin (POO). In addition, parties are often concerned about timeliness, e.g. limit the amount of time during which the sender already has the POD and the receiver still waits for the message (contract) and the POO (signature on contract).

2.2. Typical Protocol Flows

We now review the proposed trusted delivery protocol. Figure 3 shows the messages exchanged during the delivery process of the protocol (ignoring initialization steps), in several scenarios. We notice that the flows are very similar to flows of several other protocols for fair exchange, contract signing and certified mail, in particular to [ASW00] (in particular to Protocol F, esp. with the modification to certified mail suggested in section 8.3) and to [PC*03]; this implies that it should be relatively easy to add to the Trusted Delivery Layer additional services such as fair exchange (possibly with invisible TTP).

The protocol is optimized for the typical scenario, where both parties are non-faulty and communication is reliable and efficient. In such no-fault scenario, illustrated in Figure 3 (a), the source and receiver can complete the protocol without the TTP. However, to ensure fairness, the source cannot directly send the message and Proof Of Origin to the receiver (who may be faulty, and not send back the signed Proof Of Delivery). Instead, the source sends a `TTP-Escrowed Proof Of Origin` (TEPOO), i.e. a message which the TTP can convert into a Proof Of Origin. Upon receiving the TEPOO, the receiver can immediately send back the Proof Of Delivery (POD). In no-faults executions, the source will send back the Proof Of Origin (POO), and the protocol completes.

If there is a loss of either the POD message from receiver to sender, or the POO message from sender to receiver, then receiver will not receive the POO from the source, as illustrated in Figure 3 (b). In this scenario, the receiver time-outs and then sends both TEPOO and POD to the TTP. The TTP, upon receiving TEPOO and POD, sends the POD to the source and sends the POO (extracted from the TEPOO) to the receiver.

If the source does not receive the POD from the receiver within the maximal round-trip delay (2Δ), then it sends the TEPOO to the TTP. As illustrated in Figure 3 (c) and (d), the TTP forwards the TEPOO to the receiver. If the receiver is non-faulty, as in (d), it responds with the POD and then the TTP proceeds just like before, i.e. sends POD to the source and POO to the receiver. However, if the receiver is faulty, as illustrated in (c), and the TTP does not receive the POD, then fair delivery is impossible. Instead, the TTP signs and sends a Proof Of Submission to the sender.

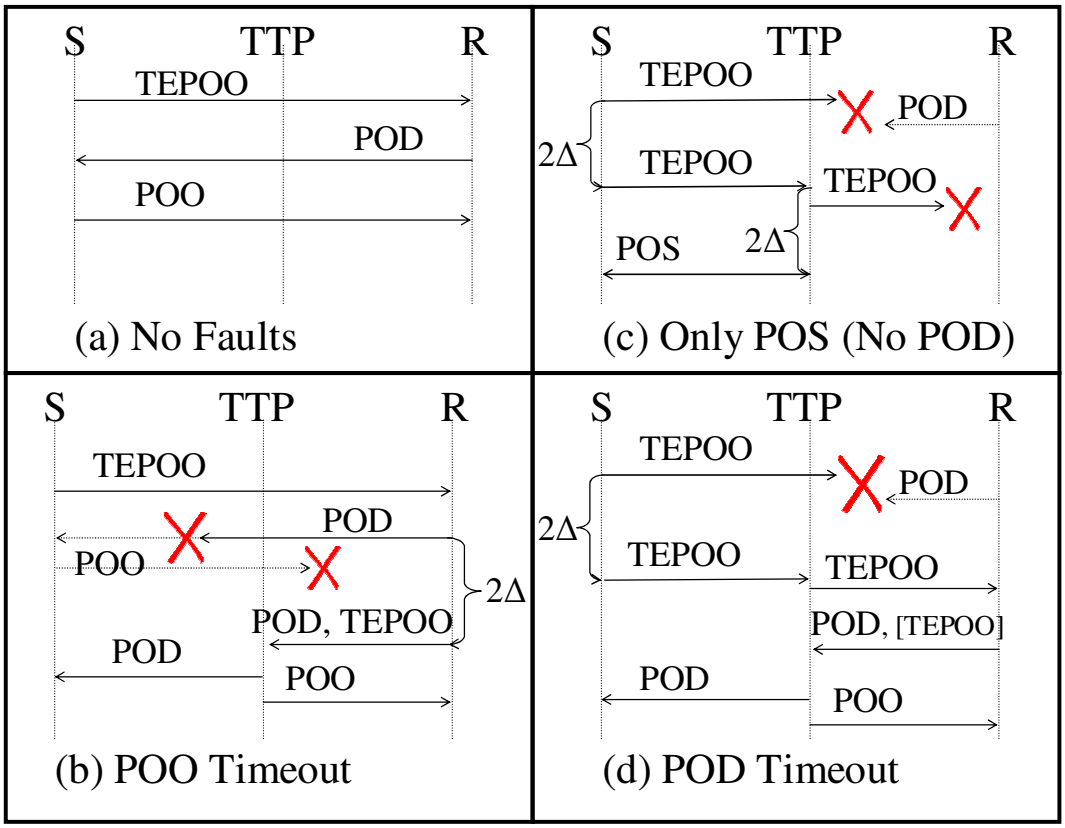

Figure 3: Typical Trusted Delivery Protocol Flows

3. Model and Specifications

We consider a set n of machines running the Trusted Delivery Layer (TDL) protocol, which we call TDL machines (defined below); without loss of generality assume each machine has a unique identifier in the set $\{1,\ldots n\}$. Each machine TDL_u has five ports, as shown in
Figure 4:

- Clock input port clk_u, which provides real-time reading (with bounded offset) periodically.
- Secure Transport Layer input ($in_{T,u}$) and output ($out_{T,u}$) ports, allowing TDL_u to communicate with other machines.
- Trusted Delivery Layer input ($in_{D,u}$) and output ($out_{D,u}$) ports, whereby the TDL_u interacts with the application.

In this section we model the system, and then present specifications for the Trusted Delivery Layer in this model. For simplicity, in this section we ignore confidentiality issues, and focus only on integrity and liveness requirements. This allows us to use a simple `adversarial model` where the adversary controls the entire environment of the protocol, including both infrastructure (clocks, communication) and application. We exclude confidentiality from this discussion since under such model, where the application is controlled by the adversary, there is no confidentiality.

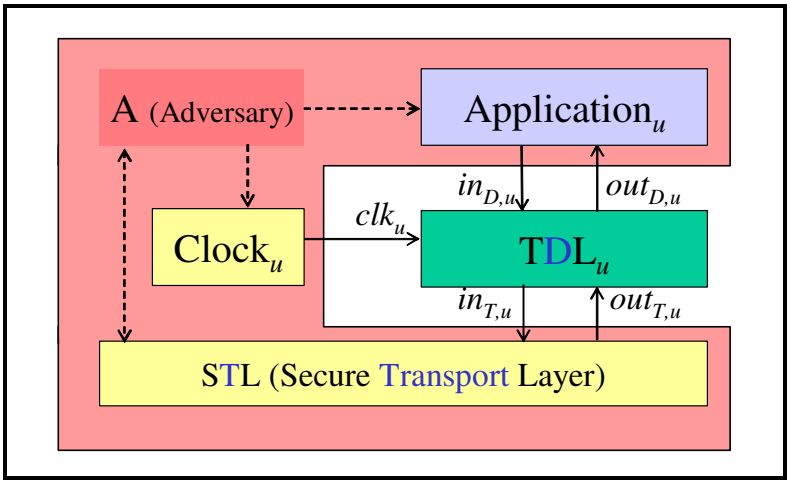

Figure 4: Interfaces of the Trusted Delivery Layer machine TDL_u

3.1. Real time single thread adversarial model

We now briefly present our model, which is an adaptation of the reactive systems models of [PW00, PWS00], and a variant of I/O automata [L96]. On the one hand, we extend these models to allow us to analyze real-time behavior, and to specify realistic bounds on message delay and clock drift. In order to do without excessive complexity, we make several restrictions and simplifications. In particular, we ignore local computation time[4]. We begin by defining our machine model.

Definition: A randomized I/O machine is a tuple $<S, Init, in, out, \delta>$ where:

- S is a finite set of states
- $Init \in S$ (initial state)
- in, out are finite sequences of non-repeating identifiers (input and output ports)
- $\delta : S \times \{0,1\}^* \times <\{0,1\}^*>^{|in|} \rightarrow S \times <\{0,1\}^*>^{|out|}$ is a (transition) function. The inputs are: the current state, random string, and a sequence of input strings, one from each of the input ports (in); some

[4] This is reasonable, since the protocols we present require minimal computation, and since we allow the adversary to control all interactions between the machines.

input ports may provide the empty input. The output is the new state and a sequence of $|out|$ output strings, one for each output port (but some of which may be empty).

We next define real time single thread adversarial configurations and their runs (executions). In such configurations, all of the environment is controlled by the adversary; we later define restrictions on the adversary as part of the specifications. Specifically, a configuration is a triple $<n,A,P>$ where A is the randomized I/O machine of the adversary, P is the randomized I/O machine of the protocol, and n is the number of protocol machines in the configuration.

Runs (executions) of the configuration are defined by a series of rounds. Each round consists of a move by the adversary and a move by one of the protocol machines selected by the adversary by identifying it on a special output port labeled *id*, as illustrated in Figure 5; we use the notation $id_i=u$ to designate that the output of port *id* of the adversary machine A at round i is the value u, where $u \in \{1,\ldots n\}$ identifies the selected protocol machine P_u.

The adversary also provides values to all the input ports of the selected protocol machine P_u such that u is the machine identified in port *id* at round i. In particular, for the TDL protocol with the interfaces shown in **Figure 4**, the input ports of protocol machine TDL_u are $\{in_{D,u}, clk_u, out_{T,u}\}$. Once the selected protocol machine made its transition, the values in its output ports become the values in the corresponding input ports of the adversary in the next round. In particular, for the TDL protocol with the interfaces shown in **Figure 4**, the output ports of protocol machine TDL_u are $\{out_{D,u}, in_{T,u}\}$.

The adversary also controls the delay of each round from the previous round, using a special output port *time*; this is used only to analyze the timing of moves and runs, and not visible to the TDL machines. The output string on port *time* is a binary encoding of a subset *TIME* of the positive real numbers (allowing a poly-time machine to specify it). Notice that this modeling of time simplifies reality, mainly by ignoring the computation time, and assuming that at any time, at most one machine is making a transition. In this sense we consider this a `single thread` model.

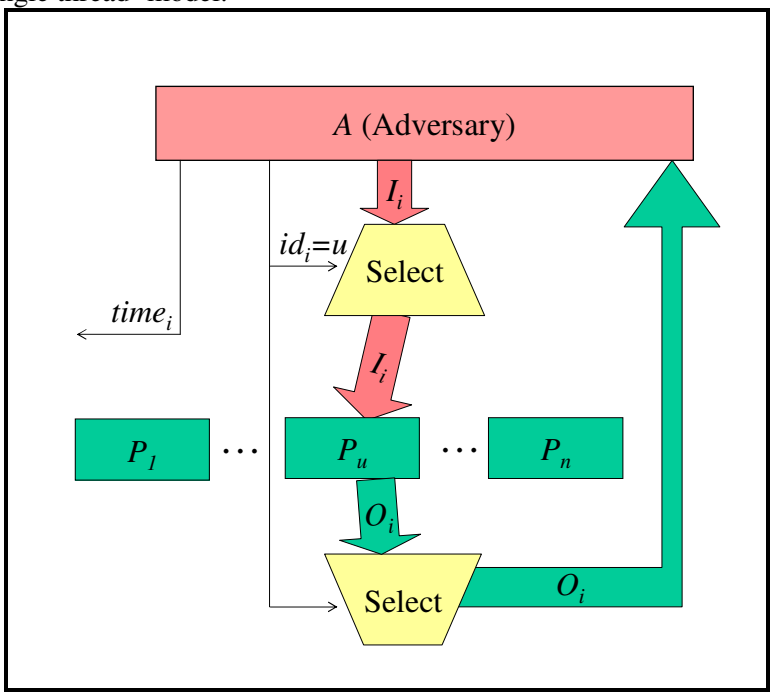

Figure 5: Round *i* of configuration $<n,A,P>$

We allow the adversary to corrupt some of the machines; let $H \subseteq \{1\ldots n\}$ denote the set of non-corrupted (honest) machines. In this work we assume that the corrupted machines are fixed in advance (rather than

selected adaptively by the adversary as in [CFGN96], or allowing processors to repeatedly switch between corrupted and correct as in proactive security [CGHN97]). Formally, we consider three types of machines: the protocol machines P; the adversary machine A; and corrupted protocol machines P_C .

We now define a real time single thread adversarial configuration $<n,H,A,P,P_C>$ and runs of it.

Definition: Consider three randomized I/O machines, $P=<S_P, Init_P, in, out, \delta_P>$, $A=<S_A, Init_A, in_A , out_A, \delta_A>$ and $P_C=<S_C, Init_C, in, out, \delta_C>$. For any positive integer n, we say that $C=<n,H,A,P,P_C>$ is a **real time single thread adversarial configuration** if: $out_A=\{\text{“time”},\text{“id”}\} \cup in$ and $in_A=out$. A **run** of C, denoted run_C, is the random variable containing the sequence $\{<S_{A,i}, t_i , p_i , I_i , S_{P,i} , S_{C,,i} , O_i>\}$, for $i:=0,1,\ldots$, where $S_{A,i} \in S_A$, $t_i \in TIME$, $p_i \in P$, $I_i \in (\{0,1\}^*)^{|in|}$, $S_{P,i} \in (S_P)^n$, $S_{C,,i} \in (S_C)^n$ and $O_i \in ((\{0,1\}^*)^{|out|})^n$, produced by the following process:

1. *Initialization :* $S_{A,0} := Init_A$; $p_0:=0$; $O_0[0]:= \Phi$; For $p:=1,\ldots n$ do: $S_{P,0}[p]:= Init_P$, $S_{C,0}[p]:= Init_C$;
2. For $i:=1,\ldots$do:
   {
   $<S_{A,i} , t_i , p_i , I_i>=\delta_A(S_{A,i-1} , r_{A,i} , O_{i-1}[p_{i-1}])$ where $r_{A,i} \in_R \{0,1\}^*$;
   For all $p \in P/\{p_i\}$ do: $\{O_i[p]:= \Phi^{|out|}$; $S_{P,i}[p]:= S_{P,i-1}[p]$; $\}$;
   If $p_i \in H$ then $<S_{P,i}[p_i], O_i[p_i]> := \delta_P(S_{P,i-1}[p_i], r_{P,i} , I_i)$ where $r_{P,i} \in_R \{0,1\}^*$;
   else $<S_{C,,i}[p_i], O_i[p_i]> := \delta_C(S_{C,i-1}[p_i], r_{C,,i} , I_i)$ where $r_{C,,i} \in_R \{0,1\}^*$;
   }

If e is the i^{th} event (element) in run $\{<S_{A,i}, t_i , id_i , I_i , S_{P,i} , S_{C,,i} , O_i>\}$ then the **real-time** of e is the sum of t_j for all $j \le i$.

Notice that run_C, is a well defined probability distribution (with $r \in_R X$ denoting uniformly random choice of r from set X).

We next specify the underlying secure transport layer, restricting the behavior of the adversary.

3.2. Secure Transport Layer Specifications

We now present the specifications (assumptions) of the underlying secure transport layer. Our assumptions (specifications) are to be interpreted as `with high probability over the runs of the system`, although for brevity we usually do not explicitly mention the negligible probability of misbehavior.

We begin by stating our assumptions on the clocks. (These assumptions are required for the notarization properties; fairness can be achieved in completely asynchronous setting.)

> **Assumption 1 (clock):** For every $u \in \{1\ldots n\}$, there is at least one clk_u event, and at most two - clk_u event, during any time interval of length $\mathbf{\Delta/2}$. Furthermore, every clk_u event is of the form $clk_u(t)$, where t is within $\mathbf{\Delta/2}$ of real time, i.e. if $clk_u(t)$ occurs at real time t' then $|t-t'|< \mathbf{\Delta/2}$.

It follows that every non-corrupted processor u can maintain a clock variable c_u such that at every time t holds $|t-c_u[t]|<\Delta$, i.e. c_u is always within offset of at most Δ from real time.

TDL machines communicate via the secure transport layer (STL), by sending messages with in_T events and receiving messages with out_T events. Each send (receive) event indicates the intended receiver (respectively, sender). We next state our assumption of authentic and reliable communication, with maximal delay Δ. We state the assumption for all machines, even adversary controlled, but of course adversary controlled machines may ignore messages.

> **Assumption 2 (authentic, reliable communication with maximal delay Δ):** if an $in_{T,u}(m,v)$ event occurs at time t, with $v \in M$, then an $out_{T,v}(m,u)$ event occurs during $[t,t+\Delta]$. Similarly, if an $out_{T,v}(m,u)$ event occurs at time t, then an $in_{T,v}(m,u)$ event occurs during $[t-\Delta , t]$.

3.3. Trusted Delivery Layer Specifications

We now present specifications of a trusted delivery layer (TDL). The TDL interacts with the application using the in_D and out_D events, as illustrated in
Figure 4. Most existing works consider only the events associated with the actual delivery process. However, for complete specifications of a realistic system, we also consider initialization events, such as generation of the private/public keys, and agreement by the parties on the details of all participants (by name and public keys). Specifically, we consider the following pairs of request (in_D) and response (out_D) events:

- *Init*: an initialization request in each party (sender *s*, receiver *r* or trusted third party *ttp*). The machine generates a pair of private and public keys; the response (in the out_D event) contains the public key of the machine. Only one *Init* request and response is required for each party, and could be used for multiple executions of the protocol; however in this manuscript we only consider a single execution.
- *Agreement*: a request in each party, identifying the parties involved in the current execution of the TDL protocol. Namely, each $in_{D,x}(\text{“Agreement”},a)$ event (request) specifies an agreement $a=<s,k_s,r,k_r,ttp,k_{ttp}>$, where $s,r,ttp \in \{1,\ldots n\}$ identify the source, receiver and TTP respectively, and k_s, k_r and k_{ttp} are the corresponding public keys. If the keys correspond to locally generated keys (at the Init request), the response repeats the parameters, i.e. $out_{D,x}(\text{“Agreement”},a)$.
- *Deliver:* the deliver request and response events, which occur only at the sender. The *Deliver* request has three parameters: $in_{D,s}(\text{“Deliver”}, a, m)$, where *m* is a message and *a* is an agreement. The *Deliver* response event has three additional parameters: $out_{D,s}(\text{“Deliver”},top,a,m,I,p)$, where *top* is the type of proof provided (either POS, for Proof Of Submission, or POD, for Proof Of Delivery), *I* is a time interval, and *p* is the proof.
- *Receive:* a receiver event with five parameters: $out_{D,s}(\text{“Receive”},a,m,I,p)$, where *m* is a message, *a* is an agreement, $I \subseteq [0,\infty)$ is a time interval and *p* is the proof of origin.

Assumption 3 (TDL Usage Specifications): for all $u,v \in \{1,\ldots n\}$ holds:
1. There is exactly one $init_u$ and $init_v$ events, and both have the same security parameter.
2. There is a single $Agreement_u$ event.
3. The execution does not contain both $Deliver_u$ and $Deliver_v$ events.

Definition: A TDL-Protocol is a a randomized I/O machine $P=<S_D, Init_D, inports_D,, outports_D, \delta_D>$ with $inports_D=\{in_D, out_T, clk\}$, $outports_D=\{out_D, in_T\}$ (as in **Figure 4**).

We now define a correct execution of a TDL protocol. The definition covers what we consider as the basic and most important TDL correctness requirements. More advanced TDL services and protocols may require additional requirements. Some of such advanced properties are abuse-freeness [GJM99], or hiding the identity and/or involvement of the TTP during the validation process. Also recall that in this section we consider only correctness (integrity) requirements. The protocol we present does *not* meet this requirement, since validation makes use of the public key of the TTP (which is in the agreement).

Definition (correct execution): A run e of configuration $<n,H,A,P,P_C>$ is $<"TDL",1^k, V, \Delta_{TDL}>$*-correct,* where:

- $V(\{POD,POO,POS\}, a, m, I, p)$ is a poly-time predicate (validity test), where $a,m,p \in \Sigma^*$ (agreement, message and proof, respectively) and $I \subseteq [0,\infty)$ (time interval), and
- Δ_{TDL} is a monotonously-increasing function over positive real numbers,

if the following hold for every $s,r,ttp \in \{1,\ldots n\}$ and $x \in H$:

- **Initialization:** if a single $in_{D,x}("Init",1^k)$ event occurs at time t, then a single $out_{D,x}("Init",1^k,k_x)$ event occurs during $(t,t+\Delta_{TDL}(\Delta)]$. No other $out_{D,x}$ event (except $out_{D,x}("Init",1^k,k_x)$) will occur.
- **Agreement:** if a single $in_{D,x}("Agreement",a)$ event occurs at time t, following an $out_{D,x}("Init",1^k,k_x)$ event, where $a=<1^k,s,k_s,r,k_r,ttp,k_{ttp}>$ and $x \in \{s,r,ttp\}$, then a single $out_{D,x}("Agreement",a)$ event occurs during $(t,t+\Delta_{TDL}(\Delta)]$.
- **POS/POD produced (liveness):** if:
  - e contains a single $out_{D,s}("Agreement",a)$ event, at some time t_{A1}, where $a=<s,k_s,r,k_r,ttp,k_{ttp}>$ and $s \in H$;
  - Either $r \in H$ and e contains a single $out_{D,r}("Agreement",a)$ event, or $ttp \in H$ and e contains a single $out_{D,ttp}("Agreement",a)$ event, at time t_{A2}
  - And an $in_{D,s}("Deliver",a,m)$ event occurs at time $t_s > max(t_{A1},t_{A2})$.

  Then e contains an $out_{D,s}("Deliver",top,a,m,I,p)$ event, where $top \in \{POS,POD\}$ and $I \subseteq [t_s,t_s+\Delta_{TDL}(\Delta)]$, which occurs during $[t_s, t_s+\Delta_{TDL}(\Delta)]$, s.t. $V(top,a,m,I,p)=True$. Furthermore, if $r \in H$ and e contains an $out_{D,r}("Agreement",a)$ event, then $top=POD$.
- **Proof Of Delivery (POD) and Submission (POS):** If e contains an $out_{D,s'}("Deliver",top,a,m,I,p)$ event, where $top \in \{POS,POD\}$ and $a=<s,k_s,r,k_r,ttp,k_{ttp}>$, then:
  - **Output valid POD/POO:** If $s' \in H$ then $s=s'$ and $V(POD,a,m,I,p,)=True$. Also, e contains an $in_{D,s}("Agreement",a)$ event and an $in_{D,s}("Deliver",a,m)$ event, the latter occurring during I.
  - **Receiver sends valid POD:** If $r \in H$ and $V(POD,a,m,I,p)=True$, then during I there was an $out_{D,r}("Receive",a,m,I,p)$ event; and before that there is an $in_{D,r}("Agreement",a)$ event.
  - **POD → POO (fairness):** If $r \in H$ and either $s \in H$ or $ttp \in H$, then e contains an $out_{D,r}("Receive",a,m,I,p)$ event.
- **Proof Of Origin (POO):** If e contains an $out_{D,r'}("Receive",a,m,I,p)$ event, where $a=<s,k_s,r,k_r,ttp,k_{ttp}>$, then:
  - **Output POO is valid:** If $r' \in H$ then $r=r'$ and $V(POO,a,m,I,p)=True$. Also, e contains an $in_{D,r}("Agreement",a)$ event.
  - **Sender sends valid POO:** If $s \in H$ and $V(POO,a,m,I,p)=True$, then during I there was an $in_{D,s}("Deliver",a,m)$ event; and before that there is an $in_{D,s}("Agreement",a)$ event.
  - **POO → POD (fairness):** If $s \in H$ and either $r \in H$ or $ttp \in H$, then e contains an $out_{D,r}("Receive",a,m,I,p)$ event.

Note: the above definitions assume specific contents to the agreements; this content appears appropriate for most existing protocols, however, it is straightforward to extend the definitions to allow more general contracts (as long as the parties can validate the contract is Ok).

We now define a computationally-correct TDL protocol, following the general definition of integrity requirements (Definition 4.2) in [PSW00].

Definition: A TDL protocol P is $<"TDL",V, \Delta_{TDL}>$-**computationally-correct** if for any n and poly-time adversary A satisfying assumptions 1-3 above, the probability that a random run e of configuration $C=<n,H,A,P,P_C>$ is not correct is negligible in the security parameter k, i.e. for all positive polynomials Q exist some k_0 such that for every $k \geq k_0$ holds

$$1\text{-}Prob(e \in_R run_C \text{ is } <"TDL",1^k, V, \Delta_{TDL}>\text{-}correct) < 1/Q(k)$$

4. Trusted Delivery Protocol and Analysis

In this section we present the Trusted Delivery Layer (TDL) protocol and analyze it. We begin by describing the cryptographic functions used by the protocol; we then describe the protocol itself, analyze it, and finally discuss extensions that reduce the trust in the TTP (improving confidentiality and integrity).

4.1. Cryptographic Functions Used

The TDL protocol requires only two basic cryptographic schemes, and both are (probabilistic) public key schemes: a public key signature scheme $PKS=<Sign,Verify,PKS.Gen>$ and a public key cryptosystem $PKC=<Enc,Dec,PKC.Gen>$. For provable security, the signature scheme should be secure against adaptive chosen message attack, and the cryptosystem should be secure against adaptive chosen ciphertext attack; for definitions see e.g. [BG02].

However, we make one additional requirement on the cryptosystem used. Namely: decryption is a deterministic function and that the decryption process (function) outputs not only the original message, but also any randomness used in the encryption process; in a chosen ciphertext attack, whenever the adversary requests decryption of ciphertext, the adversary also receives the randomness used in the encryption process.

Specifically, let $Enc_e(m;z)$ denote the encryption of message m using public key e and randomness z; we restrict our attention to public key cryptosystems where $Dec_d(Enc_e(m;z))=<m,z>$, where d is the (private) decryption key corresponding to the (public) encryption key e. This property holds for most practical public key cryptosystems; notice we do not require confidentiality of the randomness.

Cryptosystems with the above property are defined and analyzed in [ADR02]. In fact, our results seem to hold for the weaker requirement i.e. that the encryption scheme is *committing encryption*, as we define and construct (from any public key cryptosystem and commitment scheme) in another work [GH03]; in the final version we will present this weaker requirement.

4.2. Protocol

The two main components of the TDL protocol are the state machine and the validation function V. We first define the validation function V. The first parameter of the validation function is the type of proof: POO (Proof Of Origin), POD (Proof Of Delivery) and POS (Proof Of Submission). We define the function for each of these inputs:

$$V(POO,a,m,I<\sigma,rc,z>)=\begin{cases}True\ if\ Verify_{a.s.vk}(<"POO",a,m,I,Enc_{a.ttp.ek}(rc;z)>,\sigma)\\ False\ otherwise\end{cases}$$

$$V(POD,a,m,I,<\sigma,rc,z>)=\begin{cases}True\ if\ Verify_{a.r.vk}(<"POD",a,m,I,Enc_{a.ttp.ek}(rc;z)>,\sigma)\\ False\ otherwise\end{cases}$$

$$V(POS,a,m,I,\sigma)=\begin{cases}True\ if\ Verify_{a.ttp.vk}(<"POS",a,m,I>,\sigma)\\ False\ otherwise\end{cases}$$

The last parameter to the validation function *V* is the `proof`. The POS (Proof Of Submission) is simply a signature σ by the TTP on the agreement *a*, message *m,* and time interval *I* (during which the message was sent). The identities of the parties, and their public keys, are given in the agreement *a;* in particular the verification key of the TTP is *a.ttp.vk*.

The POO (Proof Of Origin) and POD (Proof Of Delivery) both consist of a signature σ by the sender (respectively, the receiver), and by the pair *rc* (random challenge) and *z* (randomizer). Both *rc* and *z* are chosen randomly by the sender. The sender first sends the TEPOO message, which provides only the signature σ, but not *rc* and *z*. It is important to understand that **while σ is a valid signature, the receiver cannot use it,** since the agreement between the parties honors only proofs that pass the validation function *V*. To provide such complete POO, both the sender and the TTP can provide *rc* and *z*; the TTP finds *rc* and *z* by decrypting $Enc_{a.ttp.ek}(rc;z)$, while the sender simply provides the (previously chosen and stored) values of *rc* and *z*.

Similarly, the POD (Proof Of Delivery) requires both the signature σ and the pair *rc* and *z*. However, this time the goal is different, and in fact *rc* and *z* are chosen by the sender – while the signature is by the receiver. We require *rc* and *z* as part of the POD to verify that the sender sent a valid encryption of them in the TEPOO message. Namely, the POD is valid only if the TTP can in fact provide the receiver with the POO, if the receiver sends it the TEPOO message (after POO time-out, i.e. 2Δ time units after sending POD to the sender; see Figure 3 (b)).

We now describe the state machine of the protocol; a simplified state diagram is shown in Figure 6. The name of each state indicates the event which causes transition out of this state (in a `typical` execution); state which are dedicated to the sender, receiver and TTP are designated by a prefix S-, R- and T-, respectively (and by color). We next describe each state and its transitions; for length restrictions, we describe only states involved in a typical execution (other states are quite similar).

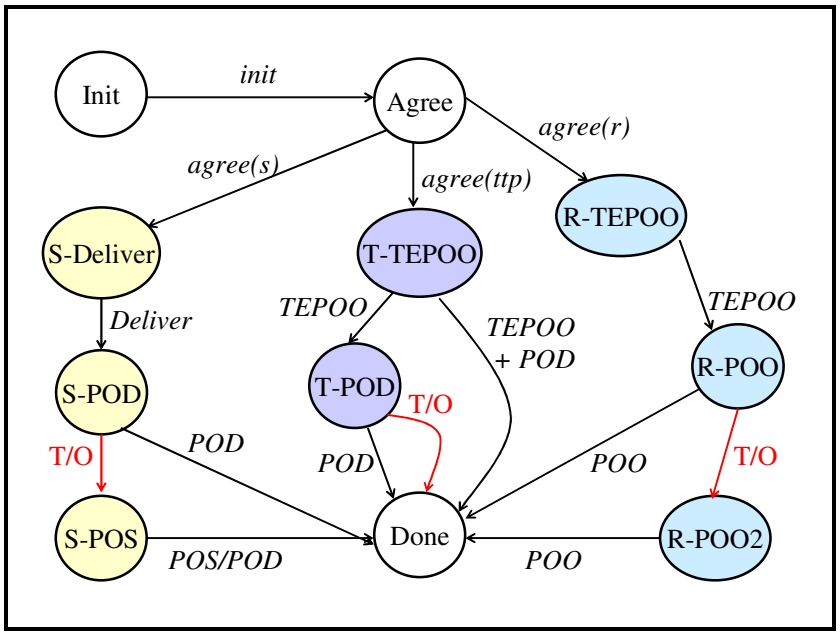

Figure 6: Simplified State Diagram of TDL-Protocol

The first two states are common to all processors. In the first state (Init), the processor (x) waits for the $in_{D,x}(\text{“Init”},1^k)$ event from the application. The processor then generates two private-public key pairs: $\langle x.ek, x.dk\rangle$ for encryption and decryption, and $\langle x.sk, x.vk\rangle$ for signing and verification. (For readability, variables maintained by x are prefixed by x, e.g. $x.ek$ denotes the (public) encryption key of x.) The processor responds to the application by providing the public keys, in an $out_{D,x}(\text{“Init”},\langle x.ek,x.vk\rangle)$ event.

State Init: On $in_{D,x}(\text{“Init”},1^k)$:
$\langle x.ek,x.dk\rangle := PKC.Gen(1^k)$;
$\langle x.sk,x.vk\rangle := PKS.Gen(1^k)$;
$out_{D,x}(\text{“Init”},\langle x.ek,x.vk\rangle)$;
State ← *Agree;*

The application collects the public keys from the three participants (sender, receiver and TTP), and inform all of them of the `agreement`, by invoking the $in_{D,x}(\text{“Agreement”}, a)$ event, where a is the agreement. Each processor finds its role (Sender, Receiver or TTP) by comparing its identity (in special variable, $x.id$) with the identities of the sender, receiver and TTP processors in the agreement (variables $a.s$, $a.r$ and $a.ttp$, respectively). The processor validates that its public key appears in the agreement correctly; if so it responds with a matching $out_{D,x}(\text{“Agreement”}, a)$ event. The next state depends on the role: S-Deliver for the sender (wait for *Deliver* request from the application), or *R-TEPOO, T-TEPOO* states for the receiver, TTP respectively (wait for TEPOO message from the sender).

State Agree: On $in_{D,x}(\text{“Agreement”}, a)$ *s.t.* $x.id \in \{a.s,a.r,a.ttp\}$ *and* $\langle x.ek,x.vk\rangle = a.k_{x.id}$:
$x.a := a$; // *save agreement in variable*
$out_{D,x}(\text{“Agreement”}, a)$;
State ← *{S-Deliver for x.id=s; R-TEPOO for x.id=r; T-TEPOO for x.id=ttp};*

The protocol now waits for a *Deliver* request at the sender; in particular, the sender is in *S-Deliver.* Once such a deliver event occurs, with the expected agreement (and parties), the source signs the TEPOO message and sends it to the receiver. The TEPOO message by itself is not sufficient for the validation function (described below). However, it contains a random challenge $x.rc$, encrypted with the public key of the TTP; let $x.z$ be the random bits used in the encryption process. By providing the random challenge $x.rc$ together with the random bits $x.z$, the source and the TTP can transform the TEPOO message to the POO

(Proof Of Origin) message. After sending the *TEPOO* message to the receiver, the sender moves to state *S-POD,* waiting for the POD from the receiver. We save several of the computation results in variables for possible reuse in the following states.

State S-Deliver: On $in_{D,x}$(*"Deliver",a,m) s.t. x.a=a and a.s=x.id:*
x.m:=m;
x.I := (x.t-Δ,x.t+5Δ] ; // valid time interval
x.rc :=random(); // random challenge
x.z:=random();// randomness used for encryption
$x.erc := Enc_{a.ttp.ek}(x.rc; x.z)$;
x.poo:=<"POO", a, m, x.I, x.erc>;
x.tepoo:=<"TEPOO", a, $Sign_{x.sk}(x.poo)$*, x.poo>*
x.timeout:=x.t+2Δ; // x.t is the current time according to *x*'s clock (sampled every *Δ/2).*
$in_{T,x}$*(x.tepoo,a.r);*
State ← S-POD;

In typical (non-faulty) executions, the receiver next receives the TEPOO message from the sender via the transport layer. The receiver waits for this in the R-TEPOO state; when the TEPOO message is received, correctly signed and with the expected agreement, time interval and identities, the receiver responds with the (signed) POD message, and moves to state R-POO (wait for the POO message from the source).

State R-TEPOO: On $out_{T,x}$*("TEPOO", a, σ, poo, s)*
s.t.: x.a=a; a.r=x.id; a.s=s; $Valid_{a.s.vk}(poo, \sigma)$*=True;*
and poo=<"POO", a, m, I, erc> for some m, I, erc, s.t. [x.t,x.t+3Δ] ∈I, |I|=6Δ;
then*:*
x.m:=m; x.poo:=poo; x.σ:=σ; x.erc:=erc;
$x.spod := Sign_{x.sk}$*("POD", a, m, I, erc);*
x.timeout:=x.t+2Δ;
$in_{T,x}$*("POD", x.spod, s);*
State ←R-POO;

In typical (non-faulty) executions, the sender next receives the POD message, to which it waits in the S-POD state. The sender validates the POD and sends back the (saved) random challenge *rc* and randomizer *z,* which will allow the receiver to transform the TEPOO to a valid POO. The sender now delivers the POD to the application and terminates (moves to state DONE).

State S-POD:
On $out_{T,x}$*("POD", σ, r) s.t.: a.r=r; a.s=x.id;* $Valid_{a.r.vk}$*(<"POD", x.a, x.m, x.I, x.erc>, σ)*
then*:*
$in_{T,x}$*("POO", x.rc, x.z); // allow receiver to transform TEPOO to POO*
$out_{D,x}$*("POD", σ, x.I, x.erc, x.rc, x.z);*
State ←DONE;
On Time-Out *(x.timeout<x.t):*
$in_{T,x}$*(x.tepoo, a.ttp);*
State ←S-POS;

In typical (non-faulty) executions, the receiver, now in R-POO state, next receives the POO message. The receiver validates that the received *rc, z* pair matches correctly the encrypted challenge in the (signed) TEPOO; in this (typical) case, it outputs the complete POO and terminates.

State R-POO:
On $out_{T,x}$*("POO", rc, z, s) s.t.: a.r=x.id; a.s=s;* $x.erc = Enc_{a.ttp.ek}(rc;z)$*;*
$out_{D,x}$*("POO", x.σ, x.poo, rc, z);*

State ←DONE;
On Time-Out *(x.timeout<x.t):*
$in_{T,x}$("POD+TEPOO", x.a, x.spod, x.poo, x.σ);
State ←T-POO2;

The additional states are similar, and will be included in the full version.

4.3. Performance

The protocol, as presented above, is very efficient. In particular:

- In a typical (no-faults) execution as in Figure 3 (a), there are only three transmissions; the TTP is not involved at all; and each party performs just one public key encryption, signature and verification operation. The execution terminates within two round-trip times.
- In the `worst` execution, each processor sends two messages; the source and receiver may perform one additional public key verification, and the TTP performs at most one decryption, two verifications and one signature. The execution terminates within four time-outs (Δ) plus two round trip times.

4.4. (Sketch of) Analysis

Theorem: Protocol *TDL* is *<"TDL",V, 6Δ>*-**computationally-correct.**

Proof (sketch): The Initialization and Agreement properties follow immediately from Assumption 3 (application specification) and from the protocol (and do not involve any cryptography). The liveness (POS/POD produced) property follow from assumptions 1 and 2 (transport layer), and the protocol.

The validity of proofs the Trusted Delivery Layer outputs to the application (POD, POO and POS) follows immediately from the protocol.

The `receiver sends valid POD` property follows immediately from the protocol, since the signature scheme used is secure against chosen message attack. This is by a standard reduction argument. Namely, assume otherwise; then there is some adversary ADV which is able to (with high probability) produce an execution where the sender outputs valid POD without the receiver performing the appropriate *Receive* event. We use ADV and the protocol machines to create an adversary ADV' which forges a signature (using chosen message attack). Namely ADV' executes the original protocol, except that whenever the receiver is supposed to sign, it instead uses asks the `signing oracle` (of the chosen message attack) to sign instead. When the sender produces a valid POD without a corresponding *Receive* event, this gives a valid signature on a message which was not asked from the `signing oracle`, contradicting the security of the signature scheme. The `sender sends valid POO` property follows exactly in the same way.

The fairness properties (POD→POO, POO→POD) follow similarly, but in a more complex reduction argument, which also uses the properties of the public key cryptosystem; in particular we use the fact that it is resilient to adaptive chosen ciphertext attack. ♦

4.5. Reducing the Trust in the TTP

Notarized delivery requires the origin and destination to place considerable trust in the TTP. The TTP is responsible for validating signatures by source and receiver, validation of the time periods specified in messages, and delivering the messages and proofs to the appropriate parties. A failure of any of these functions, malicious or benign, can appear to the application as a failure of either origin or destination.

We next sketch three extensions of the protocol that reduce the trust required in the TTP.

4.5.1. Confidentiality

In this work, we only provide informal discussion of confidentiality, since rigorous treatment requires very different specifications (see e.g. [PSW00] which provide both integrity and confidentiality specifications and analysis for certified mail). However, the solution is so simple that its security appears obvious.

To protect the confidentiality of the message, the application at the sender should encrypt it using the key of the receiver, before delivering it via the NDL-protocol. To preserve the notarization (non-repudiation) properties, encrypt here (also) using a public key cryptosystem where $Dec_d(Enc_e(m;z))=<m,z>$, where d is the (private) decryption key of the receiver, corresponding to the (public) encryption key e of the receiver, and z is the random bits used for encrypting message m; see our discussion in Section 4.1. The validation process includes encryption.

We comment that most time-stamping proposals, e.g. [HS91], protect confidentiality of the document from the notary by time-stamping the one-way, collision resistant hash of the message. This is a natural heuristics, and secure under `random oracle` analysis, but not using standard definitions of one-way and collision resistant hash functions. Therefore, there is an advantage to using the committed-encryption technique above, even if only a timestamp is needed (no receiver).
Notice now the sender signs a message encrypted using the public key of the receiver. Encrypting a message and then signing it (*EtS)* conflicts with the widely accepted design principle `*Sign then Encrypt*` (*StE*) of [AN95, AN96,Sy96]. We agree that one must be cautious in signing an encrypted document. However, in our case, since the signature includes the public encryption key (as part of the agreement), the attacks described in [AN95, AN96] do not apply, and security follows from the analysis in [ADR02].
Notice that some applications where confidentiality is a concern, e.g. bidding, may require also *non-malleability* [DDM00], i.e. prevent eavesdropper (or corrupted TTP) from sending bids which depend on the (unknown) value in an encrypted and/or committed bid. In this case, use non-malleable encryption and/or commitment schemes; see [DDM00].

4.5.2. Integrity and Availability: Distributed TTP

The TDL-protocol extends naturally to support multiple, distributed Trusted Third Parties. A trivial solution is to run multiple executions of the protocol with multiple TTPs, and define a proof to be valid if and only if signed by a sufficient number of TTPs. However, this has the undesirable property that the validation process requires the public keys of all TTPs; validation also requires multiple signature verification operations.

We can avoid these drawbacks, by distributing the TTP functionality within the TDL-Protocol. In fact, the almost only function that needs to be distributed is the decryption operation; we can use one of the (many) known threshold (distributed) or proactive public key decryption algorithms.

4.5.3. TTP Accountability

We now briefly sketch how our protocol can be extended to provide accountability as in [ASW00] but not requiring additional public key operations (as they require). Accountability, as introduced by [ASW00], implies that even if the TTP is corrupt, and therefore gives the receiver a POO without ensuring the sender has a POD, then the sender will be able to prove this cheating, or detect in real time that the TTP is faulty (by not receiving required responses from the TTP in time).

It is not difficult to add this feature to our protocol. To achieve this, we do two changes:

- The sender adds another `hidden random challenge` to the signed message (TEPOO) but this time this second challenge, *rc2,* is only committed-to – namely, only the sender can expose it, not the TTP. The sender exposes both `regular` *rc* and *rc2* when sending the POO message directly to the receiver.
- To make a valid POO, the TTP now needs not only to expose *rc* but also to *sign* the POO message.

It the TTP is cheating, and providing a POO to the receiver and a POS (not a POD) to the sender, and later the receiver presents the POO (signed by the TTP) to a judge, then the sender would be able to prove the cheating by the TTP by presenting the conflicting POS, also signed by the TTP.

4.6. Fair exchange of contracts, payments and content

Our protocol focuses on certified messaging, namely, it ensures fair exchange of the proof of origin (POO) with the proof of delivery (POD). The `proofs` are part of the protocol definition, specifically defined by the validity predicate *V* and by the agreement events from the application (specifying the parties and their public keys). This allows fair exchange of signed documents, such as contacts (both/all parties receive the signed contract), payments and other obligations. In our protocol, the parties must agree to treat the POO and POD as the exchanged signatures. Most works on fair exchange and contract signing do not require the application to adopt a specific validation predicate, and ensure fair exchange of arbitrary signed[5] documents, e.g. [ASW97, ASW00, EGL85, GJM99, PC*03]; we notice that some of these protocols, e.g. [PC*03], share much of the flows of our protocol (although with substantially more computations).

A related problem, studied e.g. in [HPS01], is fair exchange of payment for content. Namely, one party is a content repository, e.g. of music or software, and the other party is a client. The client wishes to purchase specific content *c* (e.g. identified by its hash *h(c)*) from the repository. The repository wants to ensure that the client receives the content only if the repository receives the payment, while the client wants to ensure that the repository receives the payment only if the client receives the content.

Achieving fair exchange of payment for content requires the following minor changes to our protocol; we present the changes required to support the case where the repository is the origin and the client is the receiver (the other case is similar):

1. The hash of the content, *h(c),* and its price (and other terms), should be agreed upon between the parties (added to the agreement *a*).
2. The message *m* should be $m:=Enc_{rc}(c)$, i.e. the encryption of the content using the random challenge (*rc*) as a key (of a symmetric cipher).
3. We add the following `Proof Of Delivery of Content` (PODOC) to the validation function, where *a.h* denotes the hash of the content, *h(c),* as specified in the agreement *a*:

$$V(PODOC, a, m, c, I, <\sigma, rc, z>) =$$

$$= \begin{cases} True\ if\ Verify_{a.r.vk}(<"POD", a, m, I, Enc_{a.ttp.ek}(rc; z)>, \sigma)\ and\ a.h = h(c)\ and\ m = Enc_{rc}(c) \\ False\ otherwise \end{cases}$$

Namely, the proof of delivery of content is valid only if the origin sent the agreed-upon content (identified by hash) properly encrypted using the random challenge *rc* as a key (where *rc* is exposed to the received when as part of the proof of origin).

Notice that the above process requires the origin to expose the content *c* as part of the proof of delivery of content, i.e. to get paid. This could be avoided in non-faulty executions, by having the receiver send a special additional flow which will provide a proof of delivery of content, which will not require sending the actual content; we omit the (simple) details.

4.7. Time-stamp and Real-time Key validation by TTP

We briefly comment that in some applications, there may be a concern that the signing key of the sender or receiver may expire or be revoked. The standard solution is for a TTP to provide a timestamp for the POO (or POD) proving the receiver (resp. the source) already had validated it *before* the signing key expired or

[5] The documents signed are arbitrary, but these methods require specific signature algorithms.

was revoked. Support for this case can be trivially added to our protocol, simply by doing all communication via the TTP and having the TTP sign the resulting POD and POO. Clearly, this is a rather trivial protocol; however it is probably a good idea to include it as one of the services offered by the Trusted Delivery Layer. The resulting protocol is simple and yet substantially improves on the `standard` approach of having each party contact a Time Stamping Authority to validate its key before usage, as suggested in [RFC3161] (PKI Time Stamping Protocol). The resulting protocol is similar to the protocol of [R99], but their protocol reduces the trust in the integrity of the time provided by the TTP by preserving causality order between timestamps.

5. Implementation issues

We believe that the design and protocol presented in the previous section are practical; however, there are still substantial challenges for a successful implementation. In this section we briefly discuss some of these issues.

Message space. The protocol as described did not place restrictions on the message space. However, in reality, every protocol and implementation has specific limits on acceptable messages. This includes application-specific limits, such as the kinds of messages understood by the application, as well as technical limits, such as maximal message length. It is important for such limits to be well defined in advance; otherwise a corrupt sender may send a message which the receiver cannot handle but receive a Proof Of Submission (POS) for it from the TTP. We did not include these restrictions in the protocol (e.g. in the agreement) since it is sufficient for the application-level agreement (e.g. the manual agreement) to specify such additional constraints. An implementation may want to make such limits explicit.

Server implementation. The protocol as described instructs the TTP, in several cases, to send a message to either sender or receiver (not as a response). In a practical implementation, it may be better to use a classical server design for the TTP. This is easily achieved by adding periodic `query TTP` mechanisms to sender and receiver. Notice the TTP needs to maintain a small amount of state information, even in the existing protocol (in a server implementation it will also need to keep messages until they are read).

Service levels. The protocol as described appears appropriate for many secure e-commerce applications. However, some applications and scenarios may require additional (or reduced) trusted delivery services. We envision an actual implementation to allow the parties to specify, e.g. in the agreement, their choice of additional services and of appropriate `plug-in` protocols. Notice that many of these services can be achieved with only limited changes to the presented protocol (see suggestions in the previous section). In particular, such additional (and reduced) services may include the following (all motivated by existing works):

- The variations discussed in sections 4.5-4.7.
- TTP-signatures only (no PKI) service: this service is appropriate for scenarios where the receiver and sender do not have certified public keys. Instead, they authenticate themselves to the TTP, and only rely on signatures of the TTP. This kind of certified mail service is described in [AGHP02].
- Abuse-freeness [GJM99]: this is a stronger version of the fairness property, where the receiver (sender) should not be able to convince any outside observer the sender (respectively, receiver) is currently committed to a specific `proof`, if the receiver (respectively, sender) will provide its `corresponding proof`. This is required, for example, if the sender and receiver are negotiating a deal, and the receiver attempts to present the about-to-be-signed agreement to a competitor of the sender, and thereby try to get a better deal. We note that abuse-freeness seem to require substantial additional complexity, and is not required for many applications; therefore it should be implemented and used only `as needed`.

Timing model. In this work, we adopted a realistic timing model with asynchronous communication with bound Δ on the delay and on the bias between clocks of two different processors. However, this model is

still a simplification; in particular, in reality, the bounds for clock bias is not necessarily related to the bound on the delay. Real systems are also multi-threaded (see more below).

Multiple executions. The protocol and specifications were designed for a single execution and agreement. An implementation would need to support multiple agreements, and one or more executions for each agreement, probably by using unique identifier for each execution.

Ignored exceptions. The protocol, and even the specifications, ignored several exception situations that a realistic implementation should address. In particular, we should properly respond to incorrect and invalid (e.g. not properly signed) messages, which are simply ignored in the current protocol. Also, as currently described, the protocol does not use a time-out mechanism while waiting for response from the TTP; this is since the liveness specification considers only cases where at most one party fails. For a realistic implementation and specification, we will require termination (with appropriate alert indication) even if two (out of the three) parties are faulty.

6. Conclusions and Further Research

Trusted delivery services are required in many commercial scenarios, e.g. for business-to-business commerce and for banking and payment applications. We identify two main trusted delivery services: fair delivery and timely delivery. These services are essential for many e-commerce applications. We presented a simple, efficient and practical protocol that provides both fairness and timeliness services.

Our work is trying to provide solid foundations for practical electronic commerce protocols and systems. We expect further work to provide additional solid mechanisms for secure e-commerce applications, especially in the banking and finance areas, where the importance of security is well recognized. Such work may use trusted delivery services based on the current work.

There are many directions for further research based directly on the current work, such as rigorous analysis of specific implementation (possibly using automated verification), analysis and specification of the confidentiality properties, extensions to support adaptive or mobile adversary (proactive security), standardization, deployment and applications.

We hope our real-time adversarial model may be useful to analyze lower layer security and fault-tolerance protocols (e.g. TCP, TLS), higher layer secure e-commerce protocols; we also hope it will be possible to prove general composition theorems as in [PSW00]. Finally, real systems are multi-threaded (even in a single processor), motivating appropriate extensions to our (single-threaded) model.

References

[ADR02] Jee Hea An, Yevgeniy Dodis and Tal Rabin, On the Security of Joint Signature and Encryption, in Theory and Application of Cryptographic Techniques, pp. 83-107, 2002.

[AGHP02] Martín Abadi, Neal Glew, Bill Horne and Benny Pinkas, Certified Email with a Light On-Line Trusted Third Party: Design and Implementation, Proceedings of the Eleventh International World Wide Web Conference (May 2002), 387-395.

[AN95] Ross Anderson and Roger Needham. Robustness principles for public key protocols. In Proc. Int'l. Conference on Advances in Cryptology (CRYPTO 95), volume 963 of Lecture Notes in Computer Science, pages 236--247. Springer-Verlag, 1995. http://citeseer.nj.nec.com/article/anderson95robustness.html

[AN96] Abadi, M. and Needham, R. 1996. Prudent engineering practice for cryptographic protocols. IEEE Trans. Softw. Eng. 22, 1 (Jan.), 6-15. http://citeseer.nj.nec.com/abadi96prudent.html

[ASW97] N. Asokan, M. Schunter, and M. Waidner. Optimistic Protocols for Fair Exchange. In Proceedings of 4th ACM Conference on Computer and Communications Security, Zurich, April 1997.

[ASW00] N. Asokan, V. Shoup, and M. Waidner. Optimistic Fair Exchange of Digital Signatures. IEEE J. Selected Areas in Comm., 18(4):593-610, April 2000.

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. *Adaptively secure multi-party computation*. In Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC '96), pages 639--648. ACM, 1996.

[CGHN97] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. *Proactive security: Long-term protection against break-ins*. RSA Laboratories' CryptoBytes, 3(1), 1997.

[DDN00] D. Dolev, C. Dwork, and M. Naor. Non-Malleable Cryptography. SIAM Journal on Computing, 30(2):391--437, 2000.

[EGL85] Shimon Even, Oded Goldreich, Abraham Lempel: A Randomized Protocol for Signing Contracts; Communications of the ACM 28/6 (1985) 637--647.

[G82] Oded Goldreich. A protocol for sending certified mail. Technical report, Computer Science Department, Technion, Haifa, Israel, 1982.

[GB01] Shafi Goldwasser and Mihir Bellare, Lecture Notes on Cryptography, Available online at http://www.cs.ucsd.edu/users/mihir/papers/gb.html, 1996-2001.

[GH03] Yitchak Gertner and Amir Herzberg, Commited Encryption, work in progress, 2003.

[GJM99] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In Advances in Cryptology: Proceedings of Crypto'99, volume 1666 of Lecture Notes in Computer Science, pages 449--466. Springer-Verlag, 1999.

[HPS01] B. Horne, B. Pinkas, and T. Sander. Escrow services and incentives in peer-to-peer networks. In Proc. of The ACM Conference on Electronic Commerce (EC'01), Tampa, Florida, October 2001.

[HS91] Stuart Haber and W.-Scott Stornetta. How to Time-Stamp a Digital Document. Journal of Cryptology, 3(2):99--111, 1991. http://citeseer.nj.nec.com/haber91how.html

[ISO13888-1] ISO/IEC 3rd CD 13888-1. Information technology – security techniques – Non-repudiation, Part 1: General model. ISO/IEC JTC1/SC27 N1274, March 1996.

[ISO13888-3] ISO/IEC 2nd CD 13888-3. Information technology – security techniques – Non-repudiation, Part 3: Using asymmetric techniques. ISO/IEC JTC1/SC27 N1379, June 1996.

[J98] Mike Just. Some Timestamping Protocol Failures. In Internet Society Symposium on Network and Distributed System Security, 1998. Available at http://citeseer.nj.nec.com/just98some.html.

[KMZ02] An Intensive Survey of Non-repudiation Protocols. Steve Kremer, Olivier Markowitch & Jianying Zhou . To appear in Computer Communications Journal. Elsevier. 2002.

[L96] Nancy Lynch, Distributed Algorithms, Morgan Kaufman, San Francisco, 1996.

[LPSW00] Gerard Lacoste, Birgit Pfitzmann, Michael Steiner, Michael Waidner (Editors), SEMPER - secure electronic marketplace for Europe, Vol. 1854, Springer-Verlag, ISBN = "3-540-67825-5".

[M97] Silvio Micali, Certified E-Mail with Invisible Post Offices - or - A Low-Cost, Low-Congestion, and Low-Liability Certified E-Mail System; presented at 1997 RSA Security Conference, San Francisco.

[MSP96] National Security Agency. Secure Data Network System : Message Security Protocol (MSP), January 1996.

[PC*03] Jung Min Park, Edwin Chong, Howard Siegel, Indrajit Ray, Constructing Fair-Exchange Protocols for E-commerce Via Distributed Computation of RSA Signatures, to be published in PODC'03, Boston, July 2003.

[PSW00] Birgit Pfitzmann, Matthias Schunter, Michael Waidner, Provably Secure Certified Mail, IBM Research Report RZ 3207 (#93253), IBM Research Division, Zurich, Feb. 2000.

[R00] Eric Rescorla. SSL and TLS: Designing and Building Secure Systems. Addison-Wesley, 2000.

[R98] Tal Rabin. A Simplified Approach to Threshold and Proactive RSA. In H. Krawczyk, editor, Advances in Cryptology--CRYPTO'98, Lecture Notes in Computer Science Vol. 1462, pp. 89--104, Springer-Verlag, 1998.

[R99] Meelis Roos, Integrating Time-Stamping and Notarization, Master Thesis, Tartu University, Faculty of Mathematics, May 1999.

[RFC2246] T. Dierks, C. Allen, The TLS Protocol: Version 1.0, Network Working Group, Internet Engineering Task Force (IETF). Available online at http://www.ietf.org/rfc/rfc2246.txt.

[RFC2411] R. Thayer, N. Doraswamy and R. Glenn, IP Security Document Roadmap, Network Working Group, Internet Engineering Task Force (IETF). Available online at http://www.ietf.org/rfc/rfc2411.txt. November 1998.

[RFC2560] Michael Myers, R. Ankney, A. Malpani, S. Galperin, and Carlisle Adams. RFC2560: X.509 Internet Public Key Infrastructure Online Certicate Status Protocol - OCSP, June 1999, http://www.ietf.org/rfc/rfc2560.txt.

[RFC3161] C. Adams, P. Cain, D. Pinkas, R. Zuccherato, RFC 3161: Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP), IETF Network working group, August 2001, http://www.ietf.org/rfc/rfc3161.txt.

[Sy96] P. Syverson. Limitations on Design Principles for Public Key Protocols. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, pages 62--73, Oakland, CA, May 1996. http://citeseer.nj.nec.com/syverson96limitation.html

[Z01] Jianying Zhou. "Non-repudiation in Electronic Commerce". Computer Security Series, Artech House, August 2001

[ZG96] Jianying Zhou and Dieter Gollmann. Observations on non-repudiation. In Kim Kwangjo and Matsumoto Tsutomu, editors, Advances in Cryptology - Asiacrypt 96, vol. 1163 of LNCS, pp. 133--144. Springer-Verlag, 1996.