# A Two-Phase Highly-Available Protocol for Online Validation of E-Tickets

Fernando Pedone
Software Technology Laboratory
HP Laboratories Palo Alto

HPL-2000-116
September 12th, 2000*

E-mail: pedone@hpl.hp.com

highly available e-services, electronic tickets, online validation protocols

E-ticket is an Internet service which, similarly to real-world tickets, gives their owners permission to enter a place of entertainment, use a means of transportation, or have access to some other Internet services. E-tickets can be stored in desktop computers or personal digital assistants for future use. Before being used, e-tickets have to be validated to prevent duplication, and ensure authenticity and integrity. This paper studies the e-ticket validation problem in contexts in which users cannot be trusted and servers may fail. The paper proposes formal definitions for the e-ticket validation problem, and shows that some intuitive guarantees cannot be implemented when failures may occur. The paper also presents two protocols for online validation of e-tickets.

## 1 Introduction

Widespread use of the Internet has recently led to the emergence of a variety of electronic services, also known as "e-services." E-ticket is an example of a class of e-services. Generally speaking, e-tickets are the Internet version of real-world tickets, and give their owners permission to enter a place of entertainment (e.g., theater, sports ground), use a means of transportation (e.g., airplane), or have access to some other Internet e-services (e.g., software upgrade). Before being used, e-tickets have to be acquired, which users can do by purchasing them from a web server, or simply receiving them from another user who previously acquired them or from a vendor, as part of a promotion. Once acquired, e-tickets can be stored in a desktop computer or in a personal digital assistant for future use.

To use an e-ticket, a user first sends it to a server for validation. The validation process, hereafter called *e-ticket validation problem*, results in the server either accepting or rejecting the e-ticket, and is intended to prevent duplication, and ensure authenticity and integrity. Preventing duplication avoids multiple use of an e-ticket by the same or different users; ensuring authenticity and integrity guarantees, respectively, that e-tickets are only accepted if they have been issued by an authorized source, and have not been tampered with [Sta99]. For reasons of privacy, it is also desirable that e-tickets be anonymous, that is, e-tickets should not contain any information associated with their owners.

This paper studies the e-ticket validation problem in contexts in which users cannot be trusted and servers may fail: the paper discusses formal specifications of the e-ticket validation problem, shows that some intuitive guarantees cannot be implemented when users are not trusted and servers may fail, and proposes two specifications of the e-tickets validation problem that admit solutions in the context considered. These specifications define the *at-most-once* and the *at-least-once* e-ticket validation problems. In executions without failures, both specifications require e-tickets to be accepted exactly once. In executions with failures, however, the former specification may result in some e-tickets never being accepted, and the latter specification may result in some e-tickets being accepted multiple times.

The paper also presents a simple protocol that solves the at-most-once e-ticket validation problem. The protocol is based on Atomic Broadcast, and performs online validation of e-tickets. Differently from offline protocols, online protocols require some synchronization among the servers to validate e-tickets, but do not rely on any level of trustworthiness on the users [AJSW97]. The protocol is highly available in that the failure of some servers does not prevent the remaining ones from validating e-tickets. Finally, the paper presents a more complex and efficient protocol for online validation of at-most-once e-tickets, and compare its cost to the simple protocol considering the degree of resilience, the latency, and the number of messages exchanged between servers.

The paper is structured as follows. Section 2 describes the system model and provides specifications for the e-ticket validation problem. Section 3 presents a simple protocol, and a more complex and efficient protocol for the at-most-once e-ticket validation problem. Section 4 compares the efficiency of the protocols. Section 5 discusses related work, and Section 6 concludes the paper. All proofs of correctness are in the Appendix.

## 2 System Model and Problem Definition

### 2.1 Processes, Communication and Failures

We consider a system composed of a set $\Pi_u = \{u_1, ..., u_m\}$ of user processes and a set $\Pi_s = \{s_1, ..., s_n\}$ of server processes. User and server processes execute a sequence of atomic events, where an event can be any change in the internal state of a process, the sending of a message, or the receiving of a message [Lam78]. A server process can also execute a crash event, after which the process does not execute any other event (i.e., crash-stop mode of failure). User processes may behave maliciously and cannot be trusted by server processes. We make no assumptions about process speeds or message transmission times. Processes are connected to each other by Reliable Channels, defined by the primitives *send* and *receive*. Reliable Channels guarantee that if a process sends a message $m$ to another process, and both sender and receiver do not crash, $m$ is eventually received.

We also assume that server processes can communicate with one another using Atomic Broadcast, defined by the primitives *broadcast* and *deliver*. Atomic Broadcast guarantees that if a server broadcasts a message $m$ and does not crash, it eventually delivers $m$ *(validity)*; if a server delivers a message $m$, then all servers that do not crash eventually deliver $m$ *(uniform agreement)*; for every message $m$, every server delivers $m$ at most once, and only if $m$ was previously broadcast by $sender(m)$ *(uniform integrity)*; and if two servers, $s_i$ and $s_j$, both deliver messages $m$ and $m'$, then $s_i$ delivers $m$ before $m$ if and only if $s_j$ delivers $m$ before $m'$ *(total order)*.

## 2.2 The E-ticket Problem

In a fully operational system based on e-tickets, users acquire e-tickets before using them. In this paper, we are interested in the validation of e-tickets, and thus, we assume that users have acquired their e-tickets by some means. To use an e-ticket, a user first has to send it to some server for validation, a process that results in the e-ticket being either accepted or rejected. We model e-ticket acceptance and rejection as local events in the servers, without further specifying their semantics. An accept event could be, for example, the sending of a message containing some access code to the user. Generally speaking, validation of e-tickets addresses two concerns: First, the same e-ticket should not be accepted more than once, which can happen, for example, when users distribute copies of their e-tickets to other users. Second, no solution to the first concern consisting in rejecting all e-tickets is admitted—that is, there must be situations where e-tickets are accepted. These two concerns correspond, respectively, to safety and liveness guarantees.[1]

We formally define the e-ticket validation problem, or *e-ticket problem* for short, as follows:

(E-1) If a server accepts an e-ticket $\tau$, then no other server accepts $\tau$, and a server does not accept the same e-ticket more than once; and

(E-2) Let $\sigma(\tau)$ be the set of servers that validate the same e-ticket $\tau$. If no server in $\sigma(\tau)$ crashes, then there is a server in $\sigma(\tau)$ that eventually accepts $\tau$.

If the servers in $\sigma(\tau)$ do not crash, properties E-1 and E-2 ensure that e-ticket $\tau$ is accepted *exactly-once*. If some server in $\sigma(\tau)$ crashes, however, there is no guarantee that $\tau$ is accepted by some server. Therefore, in the presence of crashes, properties E-1 and E-2 ensure that $\tau$ is accepted *at-most-once*.

In an attempt to enforce exactly-once semantics even in the presence of crashes, property E-2 might be re-phrased as "...if not all servers in $\sigma(\tau)$ crash, then there is some server in $\sigma(\tau)$ that accepts $\tau$," (hereafter denoted E-2'). It turns out, however, that properties E-1 and E-2' together lead to an unsolvable problem in the context defined in Section 2.1 even if only one server can crash. The intuition behind such a result is that if some server crashes, the remaining servers cannot tell whether an accept event took place at the crashed server. This situation

[1] Authentication and integrity of e-tickets are also major concerns (e.g., preventing users from forging e-tickets, and changing the e-ticket contents), but we do not elaborate on this matter in the paper. Standard security techniques such as cryptography are usually used to address such concerns [Sta99].

may lead to executions where an e-ticket is not accepted at all, violating property E-2', and executions where some e-ticket is accepted more than once, violating property E-1.

For example, consider the executions depicted in Figures 1 and 2, where servers $s_1$, $s_2$, and $s_3$ receive the same e-ticket $\tau$. In the first execution (Figure 1), server $s_1$ crashes before accepting $\tau$, and to satisfy property E-2', server $s_2$ accepts $\tau$. In the second execution (Figure 2), server $s_1$ crashes after accepting $\tau$. From $s_2$'s viewpoint, these executions are indistinguishable, and since $s_2$ accepts $\tau$ in the former execution, it also accepts $\tau$ in the latter, contradicting property E-1. Notice that even if the accept event is the sending of some message by $s_1$ to $s_2$, the problem is still unsolvable: since $s_1$ crashes, there is no guarantee that the message will be received by $s_2$.

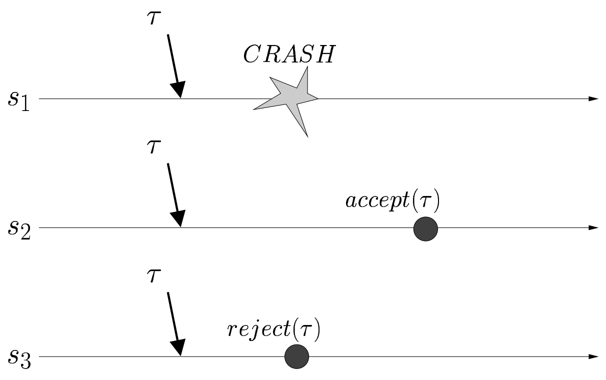

Figure 1: Execution satisfying E-1 and E-2'

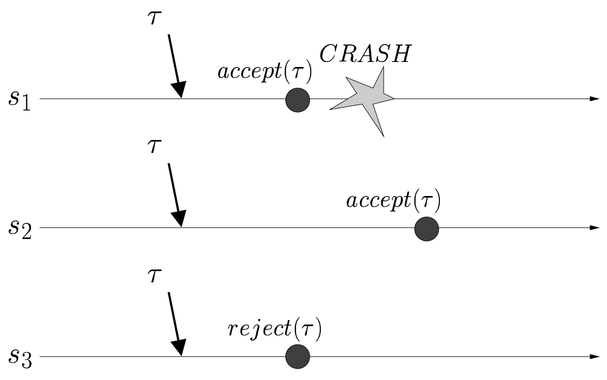

Figure 2: Execution violating E-1 and E-2'

Property E-1 can be modified and combined with property E-2', leading to the following problem:

(E-1') If a server accepts an e-ticket $\tau$ and does not crash, then no other server accepts $\tau$, and a server does not accept the same e-ticket more than once; and

(E-2') Let $\sigma(\tau)$ be the set of servers that receive the same e-ticket $\tau$. If not all servers in $\sigma(\tau)$ crash, then there is a server in $\sigma(\tau)$ that eventually accepts $\tau$.

The execution depicted in Figure 2 does not violate property E-1', and so, the argument presented for properties E-1 and E-2' no longer holds. Furthermore, as for properties E-1 and E-2, if the servers in $\sigma(\tau)$ do not crash, for any e-ticket $\tau$, properties E-1' and E-2' guarantee that $\tau$ is accepted exactly-once. If some servers in $\sigma(\tau)$ crash, however, the same e-ticket may be accepted more than once. Therefore, in the presence of crashes, properties E-1' and E-2' ensure that $\tau$ is accepted *at-least-once.*

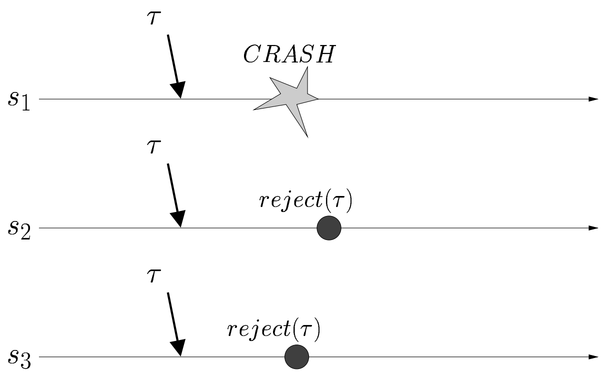

Figure 3: At-most-once e-ticket problem

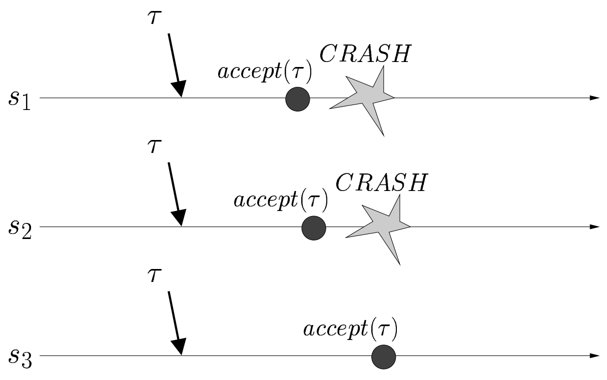

Figure 4: At-least-once e-ticket problem

Figures 3 and 4 depict, respectively, worst-case executions of the at-most-once and the at-least-once e-ticket problems. Protocols solving these problems in presence of crashes do not necessarily lead to all e-tickets being never accepted (Figure 3), or being accepted multiple times (Figure 4). However, some e-tickets may be never accepted or accepted multiple times if crashes occur. In the rest of the paper, we focus on solvability issues about the at-most-once e-ticket problem, as defined by properties E-1 and E-2.

## 3 Solving the E-ticket Validation Problem

### 3.1 A Simple E-ticket Protocol

The e-ticket validation problem can be solved by a simple protocol based on Atomic Broadcast (hereafter, SE protocol): when a server $s_i$ receives an e-ticket $\tau$ from some user, $s_i$ broadcasts $\tau$, and waits for the delivery of a message with $\tau$. If the first message delivered by $s_i$ is the message $s_i$ broadcast, $s_i$ accepts $\tau$; otherwise $s_i$ rejects $\tau$. This protocol solves the at-most-once e-ticket problem: property E-1 comes from uniform agreement and total order of Atomic Broadcast, and property E-2 comes from validity and uniform integrity of Atomic Broadcast.

Although simple, in most practical cases, the SE protocol does not solve the e-ticket validation problem efficiently (see Section 4 for details about the efficiency of e-ticket protocols). The reason being that the SE protocol orders all e-tickets in the system, but order is only needed to resolve cases where the same e-ticket is submitted multiple times, which only occurs in rare occasions. Thus, users who use their e-tickets only once end up penalized by the protocol. We present next an optimized protocol for the common case where e-tickets are used only once.

## 3.2 The Two-Phase E-ticket Protocol

The two-phase e-ticket protocol (hereafter, 2PE protocol) is an optimistic protocol in the sense that when e-tickets are used only once, the validation process is very efficient (i.e., e-tickets are validated in Phase 1), but when users try to use the same e-ticket multiple times, the validation process becomes inefficient (i.e., e-tickets are validated in Phase 2). The notion of efficiency is taken relative to the SE protocol, that is, the validation in Phase 1 of the 2PE protocol is more efficient than the validation using the SE protocol, but the validation in Phase 2 of the 2PE protocol is less efficient than the validation using the SE protocol.

**2PE Protocol Overview.** Once a server $s_i$ receives an e-ticket $\tau$ from some user, $s_i$ sends $\tau$ to all servers to find out whether $\tau$ has already been accepted by some other server. When a server $s_j$ receives $\tau$ from $s_i$, if $s_j$ has not received $\tau$ before, $s_j$ sends an ACK message to $s_i$; if $s_j$ has received $\tau$ before, $s_j$ sends a NACK message to $s_i$. Server $s_i$ waits for replies from a majority of servers—to ensure termination, at least a majority of servers should not crash (i.e., $f < n/2$). If $s_i$ does not receive any NACK messages, $s_i$ accepts $\tau$ in Phase 1; otherwise, $s_i$ proceeds to Phase 2.

Phase 2 has to handle two cases: (a) if every server that validates $\tau$ proceeds to Phase 2—that is, no server accepts $\tau$ in Phase 1, and does not crash, then some server accepts $\tau$ in Phase 2, and (b) if some server accepts $\tau$ in Phase 1, every server executing Phase 2 rejects $\tau$.[2] Servers in Phase 2 initially broadcast a message with $\tau$, and then execute a deterministic procedure whose only input is the messages they deliver. This guarantees that all servers in Phase 2 reach the same decision on which one accepts $\tau$. Moreover, the deterministic procedure is such that if some server that validates $\tau$ does not execute Phase 2, servers in Phase 2 reject $\tau$; but if all servers that validate $\tau$ execute Phase 2, one is chosen to accept $\tau$.

Figures 5 and 6 depict executions of the 2PE protocol. In Figure 5, server $s_1$ receives e-ticket $\tau$ from user $u_1$ and sends it to all servers. Server $s_1$ receives ACK messages from servers $s_1$, $s_2$, and $s_3$. Therefore, $s_1$ accepts $\tau$. Server $s_5$ receives the same e-ticket $\tau$ from user $u_2$, sends $\tau$ to all servers, and receives a NACK message from $s_3$. Thus, server $s_5$ executes Phase 2, and rejects $\tau$. In Figure 6, neither $s_2$ nor $s_5$ gather a majority of ACK messages in Phase 1. Thus, both servers start Phase 2, $s_5$ accepts the e-ticket sent by $u_2$, and $s_2$ rejects the e-ticket sent by $u_1$.

[2]Notice that while the SE protocol could be used to solve case (a), it could not be used to solve case (b).

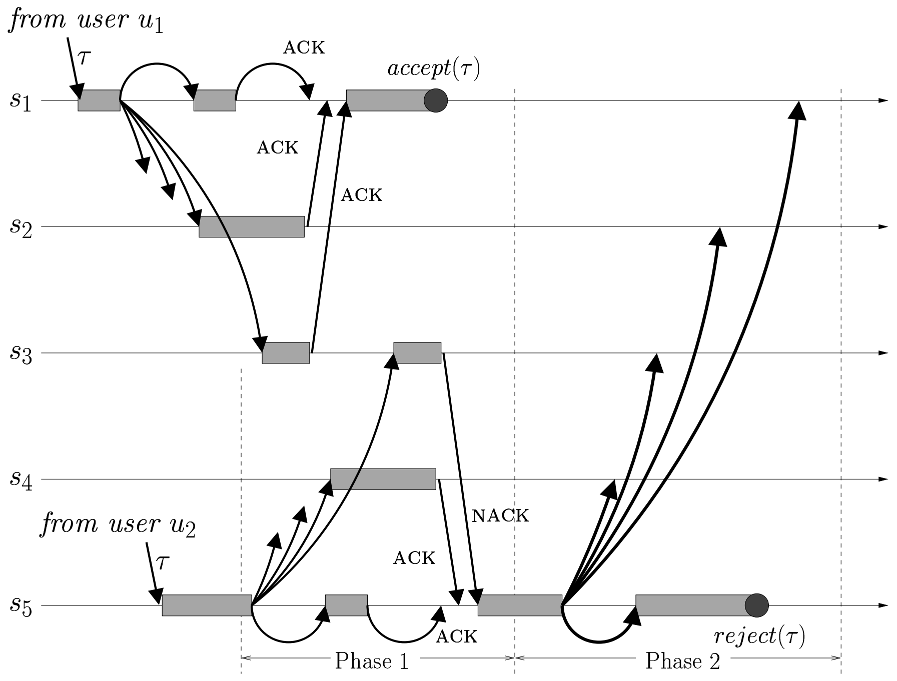

Figure 5: E-ticket accepted in Phase 1

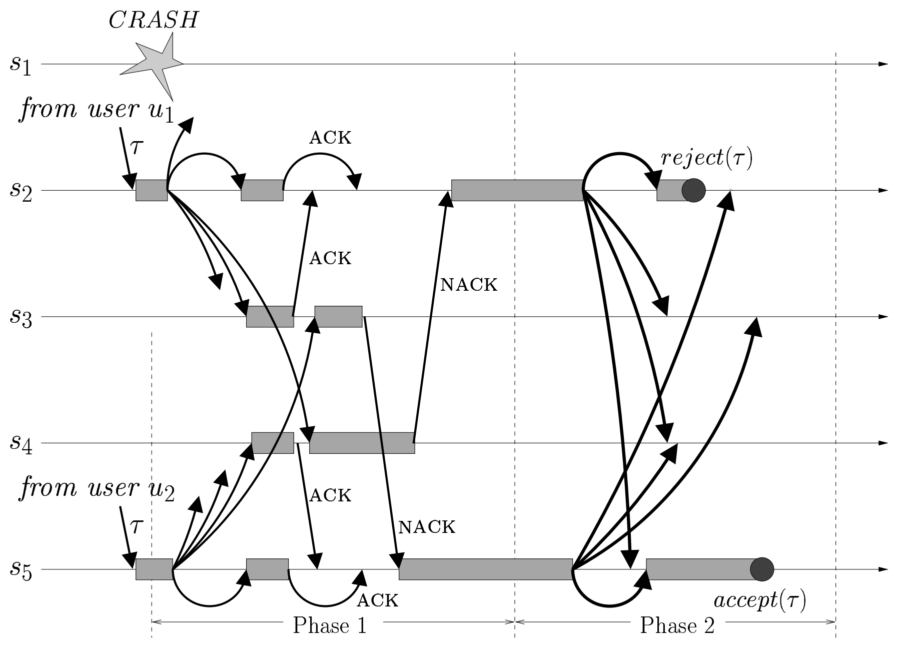

Figure 6: E-ticket accepted in Phase 2

**2PE Protocol in Detail.** Algorithm 1 presents a detailed description of the two-phase e-ticket protocol. To validate an e-ticket $\tau$ sent by some user $u$, server $s_i$ sends message $(s_i, \tau, \text{NEWTKT})$ to all servers (line 12). When a server $s_i$ receives a message $(s_j, \tau, \text{NEWTKT})$ from server $s_j$ (line 13), if $s_i$ has received some message of the type $(s_k, \tau, \text{NEWTKT})$ before, where $s_k \neq s_j$ (line 14), $s_i$ sends $(s_k, \tau, \text{NACK})$ to $s_j$ (line 15); otherwise, $s_i$ sends $(s_j, \tau, \text{ACK})$ to $s_j$ (line 18). Upon receiving a reply message (i.e., a message of the type $(*, \tau, \text{ACK})$ or $(*, \tau, \text{NACK})$) (line 19), $s_i$ updates set $\mathit{Replies}_i^\tau$ (line 20), which stores the identifiers of every server $s_k$ contained in each reply message received by $s_i$ for e-ticket $\tau$. Once $s_i$ receives $\lceil (n+1)/2 \rceil$ reply messages, and $\mathit{Replies}_i^\tau = \{s_i\}$ (line 20), that is, all message received by $s_i$ are of the type $(s_i, \tau, \text{ACK})$, $s_i$ accepts $\tau$. If there is a message $(s_j, \tau, \text{NACK})$ among the messages received by $s_i$, $s_i$ starts Phase 2 of the protocol (lines 26-36).

In Phase 2, $s_i$ broadcasts message $(s_i, \tau, \mathit{Replies}_i^\tau)$ (line 27), and waits for the delivery of any message of the type $(*, \tau, \mathit{Replies}_*^\tau)$ (line 29). Server $s_i$ stores in $\mathit{Srvs}_i^\tau$ the identifiers of the servers whose messages it already delivered (line 30), and remains in the *repeat* loop until: (a) it delivers the message it broadcast, or (b) it delivers a message $(s_k, \tau, \mathit{Replies}_k^\tau)$ that allows some server $s_k$ to accept $\tau$, that is, $\mathit{Replies}_k^\tau \setminus \mathit{Srvs}_k^\tau = \emptyset$ (line 31). The condition for $s_i$ to accept an e-ticket is deliver the message $(s_i, \tau, \mathit{Replies}_i^\tau)$ it broadcast such that $\mathit{Replies}_i^\tau \setminus \mathit{Srvs}_i^\tau = \emptyset$ (line 32). Validated e-tickets are stored in $\mathit{vTkts}$ so that they are not accepted again (line 36).

## 3.3 Correctness of the 2PE Protocol

In this section, we present the intuition behind the correctness of the 2PE protocol. All correctness proofs can be found in the Appendix.

**Property E-1.** Algorithm 1 guarantees that there are not two servers $s_i$ and $s_j$ that accept the same e-ticket $\tau$. E-tickets can be accepted in Phase 1 or Phase 2 of the protocol. We consider next these two cases:

(a) Assume that $s_i$ accepts $\tau$ in Phase 1. Thus, $s_i$ has gathered a majority of ACK messages. Therefore, $s_j$ cannot gather such majority of ACK messages, and so, $s_j$ cannot accept $\tau$ in Phase 1. To accept $\tau$ in Phase 2, $s_j$ has to deliver the message $(\tau, s_j, \mathit{Replies}_j^\tau)$ it broadcast such that $\mathit{Replies}_j^\tau \setminus \mathit{Srvs}_j^\tau = \emptyset$. By the fact that $s_i$ accepts $\tau$ in Phase 1, there is at least one server from whom $s_j$ receives an ACK message, and so, $s_i \in \mathit{Replies}_j^\tau$. $\mathit{Srvs}_j^\tau$ contains the identifiers of the servers whose messages were delivered by $s_j$ in Phase 2, and so, to have $\mathit{Replies}_j^\tau \setminus \mathit{Srvs}_j^\tau = \emptyset$, $s_i$ has to be in $\mathit{Srvs}_j^\tau$, which does not happen since $s_i$ does not execute Phase 2, and thus, does not broadcast message $(s_i, \tau, \mathit{Replies}_i^\tau)$.

(b) Assume now that $s_i$ accepts $\tau$ in Phase 2. So, $s_i$ delivered message $(s_i, \tau, \mathit{Replies}_i^\tau)$ such that $\mathit{Replies}_i^\tau \setminus \mathit{Srvs}_i^\tau = \emptyset$. Server $s_j$ does not accept $\tau$ in Phase 1, since as shown before, if $s_j$ accepts $\tau$ in Phase 1, $s_i$ cannot accept $\tau$ in Phase 2, our hypothesis. It follows that $s_j$ does not accept $\tau$ in Phase 2 either. The reason is that the mechanism used to determine the server that accepts $\tau$ in Phase 2 (lines 28-35) is deterministic and has as its only input the messages delivered by the Atomic Broadcast primitive, which guarantees that no two servers deliver messages in different orders. Therefore, if $s_i$ exits the *repeat* loop (lines 28-31) after the $l$-th iteration, $l > 0$, and accepts $\tau$, $s_j$ also exits the *repeat* loop after the $l$-th iteration, determines that $s_i$ will accept $\tau$ and rejects $\tau$.

Furthermore, when a server $s_i$ validates an e-ticket $\tau$, $s_i$ includes $\tau$ in the set $\mathit{vTkts}_i$ (line 36), so that this $\tau$ is not accepted again.

**Property E-2.** Algorithm 1 guarantees that if the servers in a set $\sigma(\tau)$ receive the same e-ticket $\tau$, and no server in $\sigma(\tau)$ crashes, then there is a server in $\sigma(\tau)$ that eventually accepts $\tau$. First, if a server $s_i$ in $\sigma(\tau)$ can contact a majority of servers before all the other servers in $\sigma(\tau)$, $s_i$ accepts $\tau$ in Phase 1. Thus, it remains to be shown that even if no server in $\sigma(\tau)$ accepts $\tau$ in Phase 1, there is a server that accepts $\tau$ in Phase 2.

Every server $s_i$ in $\sigma(\tau)$ exits the *repeat* loop at lines 28-31 when $s_i$ delivers the message it broadcast or delivers some message $(s_j, \tau, \mathit{Replies}_j^\tau)$ broadcast by $s_j$ such that $\mathit{Replies}_j^\tau \setminus \mathit{Srvs}_i^\tau = \emptyset$. In the latter case, after $s_j$ delivers message $(s_j, \tau, \mathit{Replies}_j^\tau)$, $\mathit{Srvs}_j^\tau = \mathit{Srvs}_i^\tau$, and so, $\mathit{Replies}_j^\tau \setminus \mathit{Srvs}_j^\tau = \emptyset$. Therefore, $s_j$ will exit the *repeat* loop, evaluate the *if* statement at line 32, and execute the *then* branch, accepting $\tau$. So, consider that every server $s_i$ exits the *repeat* loop when $s_i$ delivers the message it broadcast, and let $s_j$ be the server whose message is the last one to be delivered. Therefore, when $s_j$ exits the *repeat* loop, $\mathit{Srvs}_j^\tau = \sigma(\tau)$, and so, $s_j$ evaluates the *if* statement at line 32 and accepts $\tau$.

**Algorithm 1** Online e-ticket validation (for every server $s_i$)

```
1: Initialization:
2:   $rTkts_i \leftarrow \emptyset$                                  {received e-tickets set}
3:   $vTkts_i \leftarrow \emptyset$                                  {validated e-tickets set}
4:   $aTkts_i \leftarrow \emptyset$                                  {acked e-tickets set}

5: Phase 1:
6: when receive $\tau$ from $u$                                      {Task 1}
7:   if $\tau \in rTkts_i$ then                                      {if $\tau$ has already been received...}
8:     $reject(\tau)$                                                {...reject it,}
9:   else                                                            {else start $\tau$ validation:}
10:    $rTkts_i \leftarrow rTkts_i \cup \{\tau\}$                    {make sure $\tau$ will not be accepted again,}
11:    $Replies_i^\tau \leftarrow \emptyset$                         {get ready to count reply messages, and}
12:    send $(s_i, \tau, \text{NEWTKT})$ to all                      {contact other servers}

13: when receive $(s_j, \tau, \text{NEWTKT})$ from $s_j$             {Task 2}
14:   if $[\exists\, s_k$ s.t. $(s_k, \tau) \in aTkts_i]$ then       {if $s_i$ has received $\tau$ from $s_k$ before...}
15:     send $(s_k, \tau, \text{NACK})$ to $s_j$                     {...send $s_k$ to $s_j$,}
16:   else                                                           {else $\tau$ has been received for the first time:}
17:     $aTkts_i \leftarrow aTkts_i \cup \{(s_j, \tau)\}$            {keep record of $\tau$, and}
18:     send $(s_j, \tau, \text{ACK})$ to $s_j$                      {send an ack message to $s_j$}

19: when (receive $(s_j, \tau, \text{ACK})$ or $(s_j, \tau, \text{NACK})$ from $s_k$) and $(\tau \notin vTkts_i)$   {Task 3}
20:   $Replies_i^\tau \leftarrow Replies_i^\tau \cup \{s_j\}$        {gather replies for $\tau$}
21:   if [for $\lceil (n+1)/2 \rceil$ servers $s_k$: received $(*, \tau, \text{ACK})$ or $(*, \tau, \text{NACK})$ from $s_k$] then
22:     if [for $\lceil (n+1)/2 \rceil$ servers $s_k$: received $(*, \tau, \text{ACK})$ from $s_k$] then
23:       $accept(\tau)$
24:     else
25: Phase 2:
26:       $Srvs_i^\tau \leftarrow \emptyset$                         {senders of delivered messages}
27:       broadcast$(s_i, \tau, Replies_i^\tau)$                     {broadcast the acks received}
28:       repeat                                                     {repeat until can reach a decision}
29:         wait until deliver$(\tau, s_j, Replies_j^\tau)$          {only delivers change the state}
30:         $Srvs_i^\tau \leftarrow Srvs_i^\tau \cup \{s_j\}$        {keep track of message senders}
31:       until (delivered $(s_k, \tau, Replies_k^\tau)$) and $(k = i$ or $Replies_k^\tau \setminus Srvs_i^\tau = \emptyset)$
32:       if (delivered $(s_i, \tau, Replies_i^\tau)$) and $(Replies_i^\tau \setminus Srvs_i^\tau = \emptyset)$ then
33:         $accept(\tau)$
34:       else
35:         $reject(\tau)$
36:     $vTkts_i \leftarrow vTkts_i \cup \{\tau\}$                   {keep track of validated e-tickets}
```

## 4 Evaluating the 2PE Protocol

In the following, we evaluate the 2PE protocol, and compare it to the SE protocol presented in Section 3.1. Evaluating the SE protocol boils down to evaluating implementations of Atomic Broadcast in our model. We also consider an implementation of Generic Broadcast [PS99], which can be used instead of Atomic Broadcast to improve the performance of the SE protocol.[3] Our evaluation assumes executions without failures, and the most common case where the same e-ticket is only used once by the users. We compare the protocols based on (a) their resilience, and (b) the latency and (c) the number of messages exchanged to validate an e-ticket.

We consider the Atomic Broadcast protocol presented in [CT96] (hereafter, CT-broadcast), and the Optimistic Atomic Broadcast protocol [Ped99] (hereafter, OPT-broadcast). Briefly, with the CT-broadcast protocol, broadcast messages are first sent to the servers, and then the servers decide on a common delivery order for the messages using Consensus [CT96]. The OPT-broadcast protocol makes some optimistic assumptions about the system (e.g., by taking into account the hardware characteristics of the network) to deliver messages fast. The key observation is that in some cases, there is a good probability that messages arrive at their destinations in a total order, and so, servers do not have to decide on a common delivery order. Servers have to check whether the order is the same, and if this is the case, OPT-broadcast is more efficient than CT-broadcast. Otherwise, OPT-broadcast is less efficient than CT-broadcast.

The performance of the SE protocol can be improved by replacing Atomic Broadcast by Generic Broadcast. Generic Broadcast was introduced in [PS99] as a group communication protocol that takes application semantics into account to order messages. The motivation behind Generic Broadcast is that in many cases, Atomic Broadcast, which orders all messages, is stronger than necessary to guarantee application correctness. Generic Broadcast only orders messages that have to be ordered, according to the application. Since ordering messages has a cost, if not all messages are ordered, Generic Broadcast is more efficient than Atomic Broadcast. Considering SE, only messages concerning the same e-tickets have to be ordered with respect to one another, and so, in this case Generic Broadcast performs better than Atomic Broadcast, although, as it will be shown, not better than the 2PE protocol. Briefly, before ordering some message $m$, a server using Generic Broadcast checks with the other servers if there are messages

[3]The Atomic Broadcast and Generic Broadcast implementations we evaluate consider that the system model presented in Section 2 is augmented with failure detectors of class $\diamond\mathcal{S}$ [CT96].

with which $m$ has to be ordered. If not, $m$ can be delivered without the cost of a Consensus execution.

E-tickets are accepted in Phase 1 of the 2PE protocol after two communication steps: the initial $(-, -, \text{NEWTKT})$ message sent to all servers by the server that receives the e-ticket, and the reply message sent by each server. This amounts to a latency of $2\,\delta$, where $\delta$ is the maximum message delay, and $2(n-1)$ messages. To terminate (i.e., accept or reject an e-ticket), the 2PE protocol requires that a majority of servers do not crash (i.e., $f < n/2$).

Table 1 compares the cost of the 2PE protocol with the cost of the SE protocol, considering Atomic Broadcast and Generic Broadcast. 2PE can tolerate as many failures as SE with CT-broadcast but is more efficient in terms of latency and number of messages necessary to validate an e-ticket. 2PE has the same latency as SE with OPT-broadcast but better resilience and exchanges less messages. Notice that $f_{opt} = 0$ is not the resilience of OPT-broadcast, but an optimistic condition necessary to obtain the latency and number of messages values shown in Table 1. If a server crashes, the OPT-broadcast has a behavior similar to CT-broadcast. Finally, 2PE has the same latency as SE with Generic Broadcast but better resilience and needs less messages to validate e-tickets.

| **E-ticket Protocols** | **Resilience** | **Latency** | **Messages** |
|---|---|---|---|
| 2PE protocol | $f < n/2$ | $2\,\delta$ | $2(n-1)$ |
| SE protocol with... | | | |
| ...CT-broadcast [CT96] | $f < n/2$ | $4\,\delta$ | $4(n-1)$ |
| ...OPT-broadcast [Ped99] | $f_{opt} = 0$ | $2\,\delta$ | $(n+1)(n-1)$ |
| ...Generic Broadcast [PS99] | $f < n/3$ | $2\,\delta$ | $(n+1)(n-1)$ |

Table 1: 2PE *vs.* SE

## 5 Related Work

Solutions to problems similar to the e-ticket validation problem (e.g., double spending problem in electronic payment systems, digital cash, and micro-payments systems) can be divided into online and offline protocols [AJSW97]. Offline validation systems trust users not to use the same e-ticket more than once (e.g., using a "tamper-resistant" hardware), and thus, do not comply with the requirement of malicious users. Online validation systems largely rely on transactional databases to prevent users from using the same e-ticket several times. The key idea is to synchronize transactions (e.g., by means of locking) at some central validation server that only allows one transaction to be active per e-ticket at a time. In such a scheme, the first transaction to lock the database record related some e-ticket will accept it, and all the others will reject the e-ticket. Relying on a centralized resource (such as a database) may block the system in the event of single crashes, and so, is less available than the SE and the 2PE protocols.

Availability can be improved by replacing the centralized database by a highly available database. However, database systems supporting asynchronous data replication, such as Tandem Remote Data Facility (RDF) and Microsoft SQL Server, are immediately ruled out. The reason being that such systems provide weak consistency, and may allow the same e-ticket to be accepted more than once. For example, Microsoft SQL Server ships data operations to remote sites for committed transactions, and so, it can happen that two transactions access different copies of the record related to the same e-ticket at the same time and both are granted access to the records, and accept the same e-ticket. Synchronous data replication systems, such as Oracle Parallel Server (OPS), and Informix Extended Parallel Server (XPS) use clusters with or without shared disks, and can prevent multiple acceptance of the same e-ticket. Compared to traditional database systems, such solutions provide faster recovery (clients can failover to another process). However, a failover requires log based recovery: if one process takes over for a failed process, it must reconcile its own state with the log of the failed process, and would hardly provide a faster response time than specific protocols such as 2PE. Moreover, to use a parallel database system as a highly available transaction processing system, database processes executing on different machines have to access the same disks, which requires special hardware/software, such as high availability clusters.

Quorum systems have since long been prescribed as a distributed, fault-tolerant synchronization mechanism for replicated databases and objects in general [Woo98]. Although several variants exist, they all boil down to detecting conflicting requests by means of quorum intersections. Briefly, in order to treat a request $r$, a server has to gather the approval of a quorum, say $Q(r)$, of servers. If requests $r_1$ and $r_2$ conflict, $Q(r_1)$ and $Q(r_2)$ are such that there is at least one server in any intersection of $Q(r_1)$ and $Q(r_2)$. Such server detects the conflict, and refuses access to either requests $r_1$ or $r_2$. Quorum systems are a safety mechanism (i.e., they avoid multiple use of the same e-ticket), but thus far, no liveness guarantees have been associated with them (i.e., if the same e-ticket is proposed several times, one should be accepted). Moreover, when used with replicated databases, quorum systems may lead to distributed deadlocks, which are expensive to resolve.

Finally, the e-ticket validation problem bears some similarities to the resource allocation problem, and one could think of using a resource allocation system to solve the e-tickets validation problem. Apparently, few works on resource allocation address high-availability issues. Rhee [Rhe95] has proposed a modular algorithm for resource allocation in distributed systems that tolerates the failure of some components of the system. This work assumes one process for each resource, however, and the failure of such process renders the resource unavailable (although other resources can still be accessed). If one considers e-tickets as resources associated with processes, the crash of a process would mean that all e-tickets associated with that process become unavailable.

## 6 Conclusion

This paper studied the e-ticket validation problem in contexts in which users are not trusted and servers may fail. E-ticket-like services are becoming very popular with the increasing dissemination of the Internet. Even though the paper concentrates on the validation of e-tickets, the results presented can be extended to other electronic commerce-like services such as digital checks and digital coupons [Way96].

The effects of failures on electronic-commerce services were first pointed out as of major importance in [Tyg96]. Nevertheless, it seems that little has been done since then to understand its implications. This paper discussed some insights on the subject, presented formal specifications for the e-ticket validation problem, and showed that some intuitive guarantees cannot be implemented when servers are subject to failures. The paper also proposed two protocols to solve the at-most-once e-ticket validation problem.

Several points in the paper deserve further investigation. The results presented concerning the exactly-once e-ticket validation, for example, can be extended to stronger models, broadening their scope. From a solvability perspective, the 2SE protocol could be modified to solve the at-least-once e-ticket problem. Finally, intuitively speaking, the requirements of the 2SE protocol (i.e., resilience, latency, and number of messages) seem to be minimal conditions to solve the at-most-once e-ticket validation problem, but this intuition remains to be confirmed.

## References

[AJSW97] N. Asokan, P. A. Janson, M. Steiner, and M. Waidner. The state of the art in electronic payment systems. *Computer*, 30(9):28–35, September 1997.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Ped99] F. Pedone. *The Database State Machine and Group Communication Issues.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999. Number 2090.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.

[Rhe95] I. Rhee. Optimal fault-tolerant resource allocation in distributed systems. In *Symposium on Parallel and Distributed Processing (SPDP'95)*, pages 460–469. IEEE Computer Society Press, October 1995.

[Sta99] W. Stallings. *Cryptography and network security: principles and practice.* Prentice-Hall, Inc., Upper Saddle River, NJ 07458, USA, second edition, 1999.

[Tyg96] J. D. Tygar. Atomicity in electronic commerce. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 8–26, New York, May 1996. ACM.

[Way96] P. Wayner. *Digital Cash: Commerce on the Net.* Academic Press, 1996.

[Woo98] A. Wool. Quorum systems in replicated databases science or fiction. *Bulletin of the Technical Commitee on Data Engineering*, 21(4):3–11, December 1998.

## Appendix - Proof of Correctness

**Lemma 1** *(Property* E-1*). If a server accepts an e-ticket $\tau$, then no other server accepts $\tau$, and a server does not accept the same e-ticket more than once.*

PROOF (SKETCH): Assume, for a contradiction, that servers $s_i$ and $s_j$ accept $\tau$. There are three cases to consider: (a) both $s_i$ and $s_j$ accept $\tau$ at line 23, (b) $s_i$ accepts $\tau$ at line 23 and $s_j$ accepts $\tau$ at line 33, and (c) both $s_i$ and $s_j$ accept $\tau$ at line 33.

(a) From line 22, $Replies_i^\tau = \{s_i\}$ and $Replies_j^\tau = \{s_j\}$, and from line 20, $s_i$ (respectively $s_j$), did not receive any message $(s_k, \tau, \text{NACK})$. Since $s_i$ and $s_j$ received $\lceil (n+1)/2 \rceil$ messages, there is at least one server that replies to both $s_i$ and $s_j$. From Task 2, a server always sends a NACK message with the identification of the server to which it sent the ACK message before, and so, either $Replies_i^\tau = \{s_i, s_j\}$ or $Replies_j^\tau = \{s_i, s_j\}$, a contradiction.

(b) From lines 19-23, there are $\lceil (n+1)/2 \rceil$ servers that sent a message of the type $(s_i, \tau, \text{ACK})$ to $s_i$, and since before executing Phase 2, $s_j$ receives $\lceil (n+1)/2 \rceil$ messages of the type $(-, \tau, \text{ACK})$, when $s_j$ executes line 26, $s_i \in Replies_j^\tau$. Server $s_i$ does not execute Phase 2, and so, $s_i$ does not broadcast message $(\tau, s_i, -)$. Thus, $s_i \notin Srvs_j^\tau$. But from the hypothesis, $s_j$ accepts $\tau$ at line 33, and therefore, $s_i$ executes line 32 such that $Replies_j^\tau \setminus Srvs_j^\tau = \emptyset$, a contradiction.

(c) Servers $s_i$ and $s_j$ accept $\tau$ at line 33, and thus, both $s_i$ and $s_j$ execute the *then* branch of the *if* statement at line 32. By the *if* test, $s_i$ delivers $(s_i, \tau, Replies_i^\tau)$ and $Replies_i^\tau \setminus Srvs_i^\tau = \emptyset$, and $s_j$ delivers $(s_j, \tau, Replies_j^\tau)$ and $Replies_j^\tau \setminus Srvs_j^\tau = \emptyset$. Without loss of generality, consider that message $(s_i, \tau, Replies_i^\tau)$ is delivered before message $(s_j, \tau, Replies_j^\tau)$. For every $l$-th interaction of lines 28-31, $Srvs_i^{\tau,l} = Srvs_j^{\tau,l}$. So, when $s_j$ evaluates line 31, after delivering $(s_i, \tau, Replies_i^\tau)$, $Replies_i^\tau \setminus Srvs_j^\tau = \emptyset$. Thus, $s_j$ exits the *repeat* statement without delivering message $(s_j, \tau, Replies_j^\tau)$, a contradiction.

Whenever a server $s_i$ validates an e-ticket $\tau$, $s_i$ includes $\tau$ in $vTkts_i$ (line 36). Therefore, it follows directly from line 19 of the algorithm that $s_i$ does not accept the same e-ticket more than once. □

**Lemma 2** *(Property* E-2*). Let $\sigma(\tau)$ be the set of servers that receive the same e-ticket $\tau$. If no server in $\sigma(\tau)$ crashes, then there is a server in $\sigma(\tau)$ that eventually accepts $\tau$.*

PROOF (SKETCH): The proof is by contradiction. Assume that no server in $\sigma(\tau)$ crashes, but there is no server in $\sigma(\tau)$ that accepts $\tau$. When some server $s_i$ in $\sigma(\tau)$ receives an e-ticket for the first time, $s_i$ sends message $(s_i, \tau, \text{NEWTKT})$ to all servers (Task 1), and upon receiving such a message, a server sends a reply message, that is, an ACK or a NACK message, to $s_i$ (Task 2). Since there is a majority of correct servers, $s_i$ eventually receives $\lceil (n+1)/2 \rceil$ reply messages , and executes the *then* branch of the *if* statements at line 21. By the (contradiction) hypothesis, $s_i$ does not execute line 23, but executes the *else* branch of the *if* statement at line 21. Thus, $s_i$ executes broadcast$(s_i, \tau, \mathit{Replies}_i^\tau)$, and the *repeat* statement at lines 28-31.

Assume that messages are delivered according to the following order: $(s_{i_1}, \tau, \mathit{Replies}_{i_1}^\tau)$; $(s_{i_2}, \tau, \mathit{Replies}_{i_2}^\tau)$; ... $(s_{i_{|\sigma(\tau)|}}, \tau, \mathit{Replies}_{i_{|\sigma(\tau)|}}^\tau)$. We claim that every server $s_{i_x}$ in $\sigma(\tau)$ exits the *repeat* statement only after delivering message $(s_{i_x}, \tau, \mathit{Replies}_{i_x}^\tau)$. To see why, consider that after the $l$-th iteration of lines 29-30, there is a server $s_{i_y}$ that delivers a message $(s_{i_z}, \tau, \mathit{Replies}_{i_z}^\tau)$, $z < y$, such that $\mathit{Replies}_{i_z}^\tau \setminus \mathit{Srvs}_{s_{i_y}}^\tau = \emptyset$. By the agreement and total order properties of Atomic Broadcast, after $s_{i_z}$ executes the $l$-th iteration of lines 29-30, and delivers $(\tau, s_{i_z}, \mathit{Replies}_{i_z}^\tau)$, $\mathit{Srvs}_{s_{i_y},l}^\tau = \mathit{Srvs}_{s_{i_z},l}^\tau$. Thus, $\mathit{Replies}_{i_z}^\tau \setminus \mathit{Srvs}_{s_{i_z}}^\tau = \emptyset$, and $s_{i_z}$ exits the *repeat* statement and executes line 33, contradicting that no $s_{i_z}$ accepts $\tau$, and concluding the proof of our claim.

Therefore, there is a time after server $s_{i_{|\sigma(\tau)|}}$ delivers message $(s_{i_{|\sigma(\tau)|}}, \tau, \mathit{Replies}_{i_{|\sigma(\tau)|}}^\tau)$ such that $\mathit{Srvs}_{i_{|\sigma(\tau)|}}^\tau = \sigma(\tau)$. From the algorithm, for every server $s_j$, if $s_j \in \mathit{Replies}_j^\tau$ then $s_j$ received $\tau$ from some client, and so, $s_j \in \sigma(\tau)$. Therefore, we conclude that $\mathit{Replies}_{i_{|\sigma(\tau)|}}^\tau \setminus \sigma(\tau) = \emptyset$, but from the (contradiction) hypothesis $s_i$ does not execute line 33, and thus, from the *if* test at line 32, either $s_i$ did not deliver $(s_i, \tau, \mathit{Replies}_i^\tau)$ or $\mathit{Replies}_i^\tau \setminus \mathit{Srvs}_i^\tau \neq \emptyset$, a contradiction. □

**Theorem 1** *Algorithm 1 is a correct implementation of the at-most-once e-ticket problem.*

PROOF. Immediate from Lemmata 1 and 2. □