Optimistic Validation of Electronic Tickets

Fernando Pedone
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
pedone@hpl.hp.com

Abstract

Electronic tickets, or e-tickets, give evidence that their holders have permission to enter a place of entertainment, use a means of transportation, or have access to some 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 discusses e-ticket validation in contexts in which users cannot be trusted and validation servers may fail by crashing. The paper considers formal definitions for the e-ticket problem and proposes an optimistic protocol for validation of e-tickets. The protocol is optimistic in the sense that its best performance is achieved when e-tickets are validated only once.

1 Introduction

Widespread use of the Internet has recently led to the emergence of a variety of electronic services, also known as "e-services." Electronic tickets, or *e-tickets*, is an example of such a class of e-services. Generally speaking, e-tickets are the Internet counterpart of real-world tickets, and give evidence that the holder has paid or is entitled to some service (e.g., entering a place of entertainment, upgrading a software from the Internet). Users can acquire e-tickets by purchasing them from a web server, or simply receiving them from a vendor, as part of a promotion, or from another user who previously acquired them. E-tickets can be stored in a desktop computer or in a personal digital assistant (PDA) for future use.

To use an e-ticket, a user first relays it to a server for validation (e.g., if the e-ticket is stored in a PDA, the user could actually "beam" the e-ticket to the server). The validation process, hereafter called *e-ticket 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 [16]. 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 holders.

This paper studies the e-ticket problem in contexts in which users cannot be trusted (i.e., users may try to use the same e-ticket several times) and servers may fail by crashing. The paper discusses formal specifications of the e-ticket problem, shows that some intuitive guarantees cannot be implemented when users are not trusted and servers may fail, and discusses two specifications of the e-ticket problem, the *at-most-once* and the *at-least-once* e-ticket problems. In executions without failures, both specifications require e-tickets to be accepted exactly once. In executions with failures, 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.

At a first glance, the e-ticket problem is somewhat similar to the mutual exclusion problem, and one may think of solving it using mutexes. The paper discusses the relationships and points out differences between the e-ticket problem, the mutual exclusion problem, and some of its variations.

A simple highly-available protocol that solves the at-most-once e-ticket problem is presented. High availability stems from the fact that the failure of some server does not prevent the remaining ones from validating e-tickets. The paper also presents a highly-available optimistic protocol for validation of at-most-once e-tickets, and compares its cost to the simple protocol. The protocol is optimistic in the sense that its best performance is achieved when e-tickets are validated only once.

The rest of 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 the more efficient optimistic protocol for the at-most-once e-ticket validation problem. Section 4 compares the efficiency of the protocols analytically and by simulation. Section 5 discusses related work, and Section 6 concludes the paper.

2 System Model and Problem

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 [9]. Server processes may fail by crashing, but otherwise they respect their protocols (i.e., no Byzantine behavior). User processes may behave maliciously and cannot be trusted by server processes. We make no assumptions about process speeds and message transmission times. Communication between processes is reliable, and defined by the primitives *send* and *receive*. 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', they do so in the same order *(total order)*. Atomic Broadcast is implemented using point-to-point messages and some additional assumptions about the model (e.g., failure detectors [5]).

2.2 The E-ticket Problem

In this paper, we are interested in the validation of e-tickets, and thus, we do not address the issue of how users acquire e-tickets—we simply assume that they use some means to do so. To use an e-ticket, a user first sends it to some server for validation, which will result 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]

The e-ticket problem is defined as follows:

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

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

If no server in $\sigma(\tau)$ crashes, properties E-1 and E-2 ensure that e-ticket τ is accepted *exactly-once*. If some server in $\sigma(\tau)$ crashes, however, there is no guarantee that τ is accepted. Therefore, in the presence of crashes, properties E-1 and E-2 ensure that τ is accepted *at-most-once*.

[1]Authentication and integrity of e-tickets are also of major importance (e.g., preventing users from forging e-tickets, 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 [16].

In an attempt to enforce exactly-once semantics even in the presence of crashes, property E-2 may be rephrased as "...if not all servers in $\sigma(\tau)$ crash, then there is some server in $\sigma(\tau)$ that accepts τ" (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.

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 τ. In Figure 1, server s_1 crashes before accepting τ, and to satisfy property E-2', server s_2 accepts τ. In Figure 2, server s_1 crashes after accepting τ. From s_2's viewpoint, these executions are indistinguishable, and since s_2 accepts τ in the former execution, it also accepts τ 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 any messages it sends will be received.

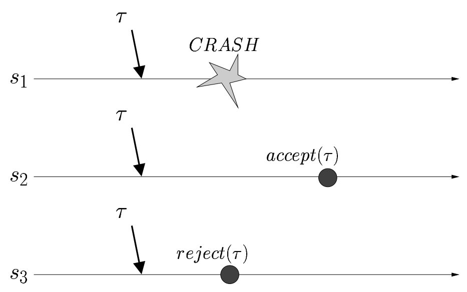

Figure 1: Execution satisfying E-1 and E-2'

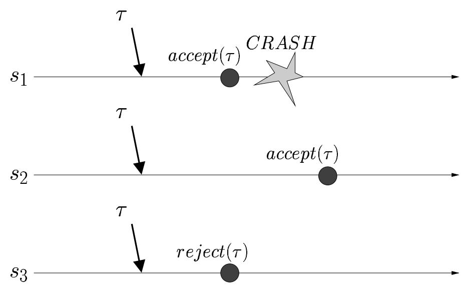

Figure 2: Execution violating E-1

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

(E-1') If a server accepts an e-ticket τ and does not crash, then no other server that does not crash accepts τ, 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 τ. If not all servers in $\sigma(\tau)$ crash, then there is a server in $\sigma(\tau)$ that eventually accepts τ.

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 τ, properties E-1' and E-2' guarantee that τ 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 τ is accepted *at-least-once*.

In the rest of the paper, we focus the discussion on the at-most-once e-ticket problem—in [11] we show that to solve the at-least-once e-ticket problem, additional assumptions have to be made about our system model.

3 Solving the E-ticket Problem

3.1 Quorum-Based E-ticket Protocol

We initially try to solve the at-most-once e-ticket problem with a quorum-based protocol: to accept an e-ticket τ, a server s_i has to gather a quorum Q of servers that agree with the acceptance of τ. Thus, to validate τ, s_i sends a message with τ to all servers and waits for the replies from a quorum of servers. A server replies with an ACK if it has not agreed with τ before, and replies with a NACK otherwise. Server s_i accepts τ if it gathers a quorum of servers that reply with ACK's. Of course, if $|Q| > n/2$, then property E-1 is guaranteed, since two servers cannot both gather a quorum for τ. However, this simple protocol can lead to situations where τ is not accepted at all, even if no server in $\sigma(\tau)$ crashes, violating property E-2. For example, consider the case where two servers obtain each $n/2$ replies with ACK: neither server can accept τ, even though no server crashes.

The problem with the quorum-based e-ticket protocol is similar to the deadlock problem in replicated databases that use a locking-based mechanism to synchronize transactions (e.g., two-phase locking) [4], and one could think of detecting it using some deadlock-detection mechanism and solve it by having servers cancel the first attempt and try to acquire a quorum again if they fail the first time. This approach, however, offers no guarantee that servers will not find themselves again in a similar situation, where no one can accept the e-ticket. Besides, it has been shown, in the context of replicated databases, that deadlocks rise as the third power of the number of database replicas [7], and so, we could also expect such a behavior from the quorum-based e-ticket protocol. In the next section we present a protocol that solves this problem.

3.2 A Simple E-ticket Protocol

The e-ticket validation problem can be solved with a simple protocol based on Atomic Broadcast (hereafter, SE protocol): when s_i receives an e-ticket τ from some user, s_i broadcasts τ and waits for the delivery of a message with τ. If the first message delivered by s_i is the message s_i broadcast, s_i accepts τ; otherwise s_i rejects τ. 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, the SE protocol does not solve the at-most-once e-ticket problem efficiently: 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 hopefully only occurs in rare occasions. Thus, since ordering messages has a cost, 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.

3.3 The Optimistic E-ticket Protocol

The optimistic e-ticket protocol (hereafter, OPT protocol) is divided into two phases: *Phase 1* and *Phase 2*. The protocol is optimistic 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: the validation in Phase 1 of the OPT protocol is more efficient than the validation using the SE protocol, but the validation in Phase 2 of the OPT protocol is less efficient than the validation using the SE protocol.

Overview of the OPT Protocol. Phase 1 of the OPT protocol is similar to the quorum-based e-ticket protocol. Once a server s_i receives an e-ticket τ from some user, s_i sends τ to all servers to find out whether some server has already accepted τ. When a server s_j receives τ from s_i, if s_j has not received τ before, s_j sends an ACK message to s_i; otherwise, 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 be up (i.e., $f < n/2$). If s_i receives a majority of ACK messages, s_i accepts τ in Phase 1; otherwise, s_i proceeds to Phase 2.

Phase 2 handles cases where more than one user tries to use the same e-ticket. Phase 2 has to take into account two constraints: (a) if every server that validates τ proceeds to Phase 2—that is, no server accepts τ in Phase 1 and does not crash, then some server should accept τ in Phase 2; and (b) if some server accepts τ in Phase 1, then no server should accept τ in Phase 2.

Servers in Phase 2 initially broadcast a message with τ, and then execute a deterministic procedure whose parameters are the messages they deliver. Since all servers deliver the same messages in the same order, and they use a deterministic procedure, they all reach the same decision on which server should accept τ. The deterministic procedure is designed in such a way as to fulfill constraints (a) and (b), described above.

Figures 3 and 4 depict executions of the OPT protocol. In Figure 3, server s_1 receives e-ticket τ 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 τ. Server s_5 receives the same e-ticket τ from user u_2, sends τ to all servers, and receives a NACK message from s_3. Thus, server s_5 executes Phase 2, and rejects τ. In Figure 4, neither s_2 nor s_5 gathers 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.

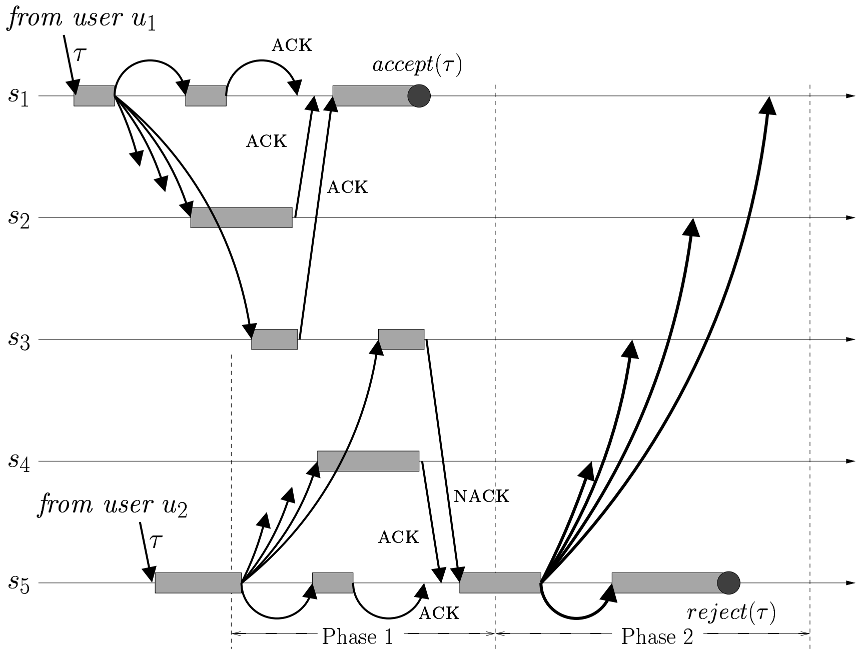

Figure 3: E-ticket accepted in Phase 1

OPT Protocol in Detail. Algorithm 1 presents a detailed description of the OPT e-ticket protocol. To validate an e-ticket τ 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, τ, NACK) to s_j (line 15); otherwise, s_i sends (s_j, τ, 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 $Replies_i^\tau$ (line 20), which stores the identifiers of every server s_k contained in each reply message received by s_i for e-ticket τ. Once s_i receives $\lceil (n+1)/2 \rceil$ reply messages and all messages received by s_i are of the type (s_i, τ, ACK) (lines 21–22), s_i accepts τ(line 23). If there is a message (s_j, τ, 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 $(s_i, \tau, Replies_i^\tau)$ (line 27), and waits for the delivery of any message of the type $(*, \tau, Replies_*^\tau)$ (line 29). Server s_i stores in $Srvs_i^\tau$ 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 $(s_k, \tau, Replies_k^\tau)$ that allows some server s_k to accept τ, that is, $Replies_k^\tau \subseteq Srvs_i^\tau$ (line 31). The condition for s_i to accept an e-ticket is deliver the message $(s_i, \tau, Replies_i^\tau)$ it broadcast such that $Replies_i^\tau \subseteq Srvs_i^\tau$ (line 32). Validated e-tickets are stored in $vTkts$ so that they are not accepted again (line 36).

For a proof of correctness of the Optimistic E-ticket protocol see [10].

4 Evaluating the OPT Protocol

4.1 Analytical Evaluation

In the following, we evaluate the SE and the OPT protocols analytically. For the SE protocol we consider an implementation of Atomic Broadcast and an implementation of Generic Broadcast [12], which can be used instead of Atomic Broadcast to improve the performance of the SE protocol.[2] Our analytical evaluation assumes ex-

[2]The Atomic Broadcast and Generic Broadcast implementations we evaluate assume that the system model presented in Section 2 is augmented with failure detectors of class $\diamond\mathcal{S}$ [5].

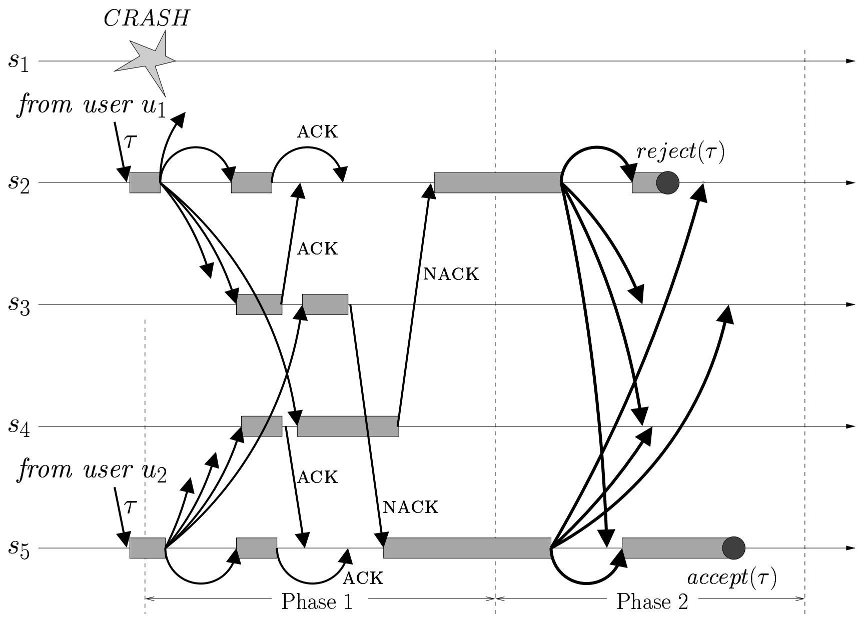

Figure 4: E-ticket accepted in Phase 2

ecutions without failures and the most common case where the same e-ticket is only used once by the users—multiple use of the same e-ticket is evaluated in the next section by simulation. 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 [5] (hereafter, CT-broadcast). Briefly, in the CT-broadcast protocol, broadcast messages are first sent to the servers, which decide on a common delivery order for the messages using Consensus [5]. The performance of the SE protocol can be improved by replacing Atomic Broadcast by Generic Broadcast [12]. Generic Broadcast takes application semantics into account to order messages only when really necessary, 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. Before ordering some message m, a server using Generic Broadcast checks with the other servers if there are messages with which m has to be ordered. If not, m can be delivered without the cost of a Consensus execution.

In the OPT protocol, e-tickets are accepted in Phase 1 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 δ is the maximum message delay, and $2(n-1)$ messages. To accept or reject an e-ticket, the OPT protocol requires that a majority of servers do not crash (i.e., $f < n/2$).

Summing up (see Table 1), the OPT protocol can tolerate as many failures as the SE protocol with CT-broadcast [5] but its Phase 1 is more efficient in terms of latency and number of messages necessary to validate an e-ticket. Compared to the SE protocol with Generic Broadcast [12], the OPT protocol has the same latency but better resilience and needs fewer messages to validate e-tickets in Phase 1.

Protocol	**Resilience**	**Latency**	**Messages**
OPT (Phase 1)	$f < n/2$	$2\,\delta$	$2(n-1)$
SE with [5]	$f < n/2$	$4\,\delta$	$4(n-1)$
SE with [12]	$f < n/3$	$2\,\delta$	$(n+1)(n-1)$

Table 1: OPT *vs.* SE

Algorithm 1 OPT protocol (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 already received $\tau$...}
8:     $reject(\tau)$                                        {...reject it,}
9:   else                                          {else start validation:}
10:    $rTkts_i \leftarrow rTkts_i \cup \{\tau\}$                    {keep $\tau$,}
11:    $Replies_i^\tau \leftarrow \emptyset$   {get ready to count replies, and}
12:    send $(s_i, \tau, \text{NEWTKT})$ to all          {contact others}

13: when receive $(s_j, \tau, \text{NEWTKT})$ from $s_j$     {Task 2}
14:   if $[\exists\, s_k \text{ s.t.} (s_k, \tau) \in aTkts_i]$ then   {if know $\tau$...}
15:     send $(s_k, \tau, \text{NACK})$ to $s_j$       {...send nack 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 $\tau$, and}
18:     send $(s_j, \tau, \text{ACK})$ to $s_j$              {send ack 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:   $Replies_i^\tau \leftarrow Replies_i^\tau \cup \{s_j\}$     {get 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 acks}
28:       repeat       {repeat until can accept/reject $\tau$}
29:         wait until deliver($s_j, \tau, Replies_j^\tau$)
30:         $Srvs_i^\tau \leftarrow Srvs_i^\tau \cup \{s_j\}$
31:       until $(j = i$ or $\exists s_k \in Srvs_i^\tau$ s.t.
                                 $Replies_k^\tau \subseteq Srvs_i^\tau)$
32:       if (delivered $(s_i, \tau, Replies_i^\tau)$) and
                          $(Replies_i^\tau \subseteq Srvs_i^\tau)$ then
33:         $accept(\tau)$
34:       else
35:         $reject(\tau)$
36:     $vTkts_i \leftarrow vTkts_i \cup \{\tau\}$
```

4.2 Simulation-Based Evaluation

To evaluate the performance of the OPT and the SE protocols under different system loads, we developed a simple simulation model in C++ using the simulation package CSIM [6]. The simulation model consists of a group of servers connected by a local-area network. Each server has two threads, one to create e-tickets and one to validate them. The former thread only creates e-tickets, and the latter thread runs the actual validation protocol.

There are two parameters to control the creation of e-tickets: the *think time* and the *conflict rate* (CR). The think time and the number of servers determine the load of the system—that is, the number of e-tickets submitted in one execution, as described next. The conflict rate determines the percentage of e-tickets that are submitted more than once in an execution—that is, e-tickets that are rejected. We have arbitrarily considered 6 servers in our experiments and a message transmission latency between 3 and 5 milliseconds per message: whenever a message is sent, the actual latency is randomly taken within this range. CT-broadcast is used to broadcast messages, and we assume that during the executions servers do not crash nor are suspected to have crashed.

An execution proceeds as follows. In each server, the thread responsible for the creation of e-tickets initially (a) generates an e-ticket taking the conflict rate into account and forwards it to all servers (using a "broadcast" in the case of the SE protocol, and a "send to all" in the case of the OPT protocol); (b) waits the time determined by the think time; and (c) starts again with item (a) until the simulation finishes.

Figures 5, 6, 7, and 8 depict the results found in the experiments. In each case, we present the latency to validate an e-ticket (i.e., the time elapsed between an e-ticket is forwarded by the thread that creates it and the e-ticket is accepted or rejected in the same server by the thread that validates them). For the OPT protocol, we present individual results for e-tickets validated in Phase 1, Phase 2, and their mean value.

Figure 5 shows executions where the think time is constant (i.e., 200 milliseconds) and the conflict rate varies. The latency for the SE protocol does not change with the conflict rate because the protocol always uses the same mechanism to validate e-tickets. With the OPT protocol, the higher the conflict rate, the greater the network contention because more messages have to be exchanged to handle rejected e-tickets. Thus, the latency for validating e-tickets in Phases 1 and 2 increases with the conflict rate. From Figure 5, with a conflict rate below 30%, the latency of the OPT is in the average smaller than the latency of the SE protocol; after this point, the OPT protocol in the average performs worse than the SE protocol.

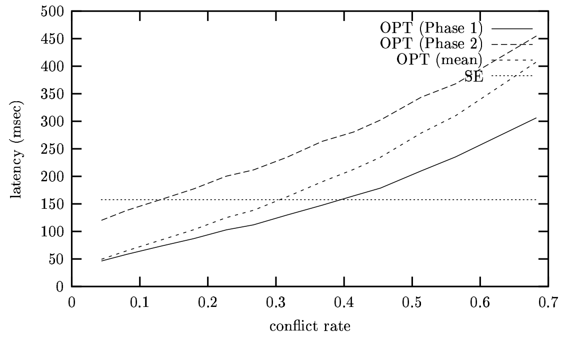

Figure 5: Conflict rate *vs.* latency

Figures 6, 7, and 8 depict the effect of the think time on the latency, for three different conflict rates. In all graphs, as the think time increases, the network contention decreases, and both protocols tend to have similar latencies. For smaller think times, the latency depends on the conflict rate: the OPT protocol performs better than the SE protocol for a conflict rate of about 5% , both protocols have similar performance at about 30%, and the OPT protocol performs worse than the SE protocol for a conflict rate of about 40%.

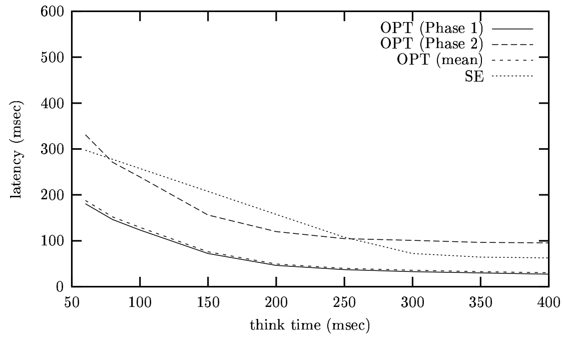

Figure 6: Think time *vs.* latency (CR ≈ 0.05)

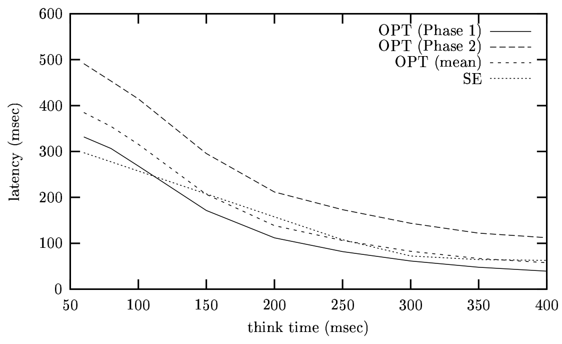

Figure 7: Think time *vs.* latency (CR ≈ 0.30)

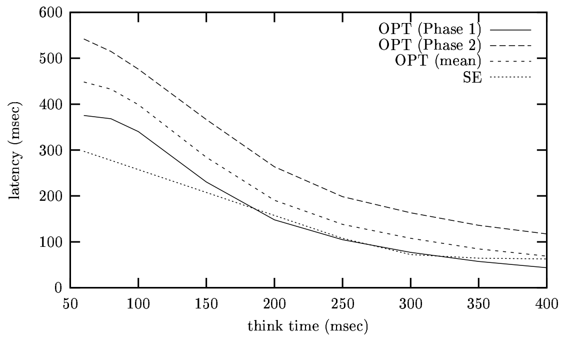

Figure 8: Think time *vs.* latency (CR ≈ 0.40)

4.3 Garbage Collecting E-Tickets

In all protocols, servers have to keep accepted e-tickets indefinitely to guarantee that they will not be accepted again. As the execution evolves, however, garbage collecting e-tickets may become an important issue. One way to tackle this problem is allow e-tickets to "expire," and embed some expiration information in the e-ticket before it is provided to the user. This approach reduces flexibility but makes sense in many practical scenarios. From the system's perspective, it requires servers to be able to determine when an expiration event has happened or when a deadline has passed. Besides, users should be prevented from modifying the expiration time of e-tickets, which can be done with use of some security technique (e.g., public-key cryptography).

5 Related Work

5.1 Transaction-Based Systems

Solutions to problems similar to the e-ticket validation problem (e.g., double spending problem in electronic payment systems, digital cash, micro-payments systems) can be divided into online and offline protocols [1]. Offline validation systems trust users not to use the same e-ticket more than once (e.g., using a "tamper-resistant" hardware that prevents multiple use or copy of an e-ticket), 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 to some e-ticket will accept it, and all the others will reject the e-ticket. Relying on a centralized resource (such as a database) provides less availability than the SE and the OPT protocols.

Availability can be improved by replacing the centralized database by a highly-available database. Database systems supporting asynchronous data replication, however, such as Tandem Remote Data Facility (RDF) and Microsoft SQL Server, are immediately ruled out since 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, accepting 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. However, failover requires log-based recovery: if one process takes over for a failed process, it must reconcile its state with the log of the failed process, and would hardly provide a faster response time than the OPT protocol. Moreover, these solutions require special hardware, such as high-availability clusters.

5.2 E-tickets and Mutexes

The e-ticket problem is somewhat similar to the mutual exclusion and to the k-exclusion problems. Most previous works on mutual exclusion assumed that processes have access to a shared memory [8, 13] and that processes do not crash in the critical section [15]. In the mutual exclusion problem, a group of processes compete for access to a *critical section.* Access to the critical section is granted in such a way that (a) in each configuration of every execution of a mutual execution algorithm, at most one process is in the critical section, and (b) in every execution, if some process requests to enter a critical section, then eventually this same process is in the critical section [2]. Clearly, if processes may crash, the problem cannot be solved—for example, a process that crashes in the critical section never leaves it, preventing other processes from entering the critical section.

Failures in the critical section have been studied in the context of the k-exclusion problem [2, 3]. In the *k-exclusion problem,* (a) no more than k processes are concurrently in the critical section (*k-Exclusion*), and (b) if at most $f < k$ processes are faulty, then a correct process that requests to enter a critical section eventually does so (*k-Lockout Avoidance*). Similarly to the at-least-once e-ticket problem, which allows multiple processes to accept an e-ticket, the k-exclusion problem allows multiple processes to be simultaneously in the critical section. However, the at-least-once e-ticket problem only allows multiple processes to accept the same e-ticket in runs where processes crash; in runs where processes do not crash, only one process can accept a given e-ticket.

5.3 The Resource Allocation Problem

Finally, one could think of using a resource allocation system to solve the e-ticket validation problem. Apparently, few works on resource allocation address high-availability issues. Rhee [14] 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, and the failure of such process renders the resource unavailable (although other resources can still be accessed). Considering e-tickets as resources, the crash of a process renders all e-tickets associated with it unavailable.

6 Conclusion

This paper studied the e-ticket 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 [18].

The effects of failures on electronic-commerce services were first pointed out in [17]. Nevertheless, it seems that little has been done since then to understand their implications. This paper discussed some insights on the subject, presented formal specifications for the e-ticket 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 problem and compared them analytically and by simulation.

Acknowledgments

I wish to thank Svend Frølund for the interesting discussions we had about e-ticket validation and Rui Oliveira for providing me with some feedback on a previous version of this paper.

References

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

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill International, 1998.

[3] A. Bar-Noy, D. Dolev, D. Koller, and D. Peleg. Fault-tolerant critical section management in asynchronous environments. *Information and Computation*, 95(1):1–20, November 1991.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

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

[6] *CSIM 18 simulation engine (C++ version).* Mesquite Software, Inc. 3925 W. Braker Lane, Austin, TX 78755-0306.

[7] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), June 1996.

[8] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

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

[10] F. Pedone. A two-phase highly-available protocol for online validation of e-tickets. Technical Report HPL-2000-116, Hewlett-Packard Laboratories, 2000.

[11] F. Pedone. Online fault-tolerant validation of electronic tickets. Technical report, Hewlett-Packard Laboratories, 2001.

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

[13] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, 1986.

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

[15] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, 1993.

[16] W. Stallings. *Cryptography and network security: principles and practice.* Prentice-Hall, Inc., second edition, 1999.

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

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