The PlusCal Code for Byzantizing Paxos by Refinement

Leslie Lamport

28 June 2011

Contents

This document contains the PlusCal code of the four specifications described in the paper *Byzantizing Paxos by Refinement*, which is available on the Web [1]. Comments are indicated by a gray background. The code comes from the TLA$^+$ modules, which are available on the Web site. The algorithms use some constants that are declared in the TLA$^+$ modules, but the meanings of those constants should be clear. The code was formatted by hand, so errors could have been introduced.

1 Algorithm Consensus

We specify the safety property of consensus as a trivial algorithm that describes the allowed behaviors of a consensus algorithm. It uses the variable *chosen* to represent the set of all chosen values. The algorithm allows only behaviors that contain a single state-change in which the variable *chosen* is changed from its initial value $\{\}$ to the value $\{v\}$ for an arbitrary value v in Value. The algorithm itself does not specify any fairness properties, so it also allows a behavior in which *chosen* is not changed. We could use a translator option to have the translation include a fairness requirement, but we don't bother because it is easy enough to add it by hand to the safety specification that the translator produces.

A real specification of consensus would also include additional variables and actions. In particular, it would have *Propose* actions in which clients propose values and *Learn* actions in which clients learn what value has been chosen. It would allow only a proposed value to be chosen. However, the interesting part of a consensus algorithm is the choosing of a single value. We therefore restrict our attention to that aspect of consensus algorithms. In practice, given the algorithm for choosing a value, it is obvious how to implement the *Propose* and *Learn* actions.

For convenience, we define the macro *Choose*() that describes the action of changing the value of *chosen* from $\{\}$ to $\{v\}$, for a nondeterministically chosen v in the set Value. (There is little reason to encapsulate such a simple action in a macro; however our other algorithms are easier to read when written with such macros, so we start using them now.) The **when** statement can be executed only when its condition, $chosen = \{\}$, is true. Hence, at most one *Choose*() action can be performed in any execution. The **with** statement executes its body for a nondeterministically chosen v in *Value*. Execution of this statement is enabled only if *Value* is non-empty–something we do not assume at this point because it is not required for the safety part of consensus, which is satisfied if no value is chosen.

We put the *Choose*() action inside a **while** statement that loops forever. Of course, only a single *Choose*() action can be executed. The algorithm stops after executing a *Choose*() action. Technically, the algorithm deadlocks after executing a *Choose*() action because control is at a statement whose execution is never enabled. Formally, termination is simply deadlock that we want to happen. We could just as well have omitted the **while** and let the algorithm terminate. However, adding the **while** loop makes the TLA$^+$ representation of the algorithm a tiny bit simpler.

```
--algorithm Consensus {
  variable chosen = {};
  macro Choose() { when chosen = {};
                   with (v ∈ Value) { chosen := {v} }
                 }
  { lbl: while (TRUE){ Choose() }
  }
}
```

2 Algorithm Voting

In the algorithm, each acceptor can cast one or more votes, where each vote cast by an acceptor has the form $\langle b, v\rangle$ indicating that the acceptor has voted for value v in ballot b. A value is chosen if a quorum of acceptors have voted for it in the same ballot.

The algorithm uses two variables, *votes* and *maxBal*, both arrays indexed by acceptor. Their meanings are:

$votes[a]$	The set of votes cast by acceptor a.
$maxBal[a]$	The number of the highest-numbered ballot in which a has cast a vote, or -1 if it has not yet voted.

The algorithm does not let acceptor a vote in any ballot less than $maxBal[a]$.

We specify our algorithm by the following PlusCal code. The specification *Spec* defined by this algorithm describes only the safety properties of the algorithm. In other words, it specifies what steps the algorithm may take. It does not require that any (non-stuttering) steps be taken. Liveness is discussed in the TLA$^+$ specification.

```
--algorithm Voting {
   variables votes = [a ∈ Acceptor ↦ {}],
             maxBal = [a ∈ Acceptor ↦ −1];
   define {
```

The **define** section adds TLA$^+$ definitions of operators that can use the algorithm's variables and can be used within the algorithm.

We now define the operator *SafeAt* so $SafeAt(b, v)$ is a function of the state that equals TRUE if no value other than v has been chosen or can ever be chosen in the future (because the values of the variables *votes* and *maxBal* are such that the algorithm does not allow enough acceptors to vote for it). We say that value v is safe at ballot number b iff $Safe(b, v)$ is true. We define *Safe* in terms of the following two operators.

Note: This definition is weaker than would be necessary to allow a refinement of ordinary Paxos consensus, since it allows different quorums to "cooperate" in determining safety at b. This is used in algorithms like Vertical Paxos that are designed to allow reconfiguration within a single consensus instance, but not in ordinary Paxos. See [2].

We define *SafeAt* in terms of the following two operators.

$$VotedFor(a, b, v) \triangleq \langle b, v\rangle \in votes[a]$$

True iff acceptor a has voted for v in ballot b.

$$DidNotVoteIn(a, b) \triangleq \forall v \in Value : \neg VotedFor(a, b, v)$$

We now define *SafeAt*. We define it recursively. The nicest definition is:

```
RECURSIVE $SafeAt(\_, \_)$
$SafeAt(b, v) \triangleq$
  $\lor b = 0$
  $\lor \exists Q \in Quorum :$
       $\land \forall a \in Q : maxBal[a] \geq b$
       $\land \exists c \in -1 \,..\, (b-1) :$
            $\land (c \neq -1) \Rightarrow \land SafeAt(c, v)$
                                  $\land \forall a \in Q :$
                                       $\forall w \in Value :$
                                          $VotedFor(a, c, w) \Rightarrow (w = v)$
            $\land \forall d \in (c+1) \,..\, (b-1), a \in Q : DidNotVoteIn(a, d)$
```

However, TLAPS does not currently support recursive operator definitions. We therefore define it as follows using a recursive function definition.

```
$SafeAt(b, v) \triangleq$
  LET $SA[bb \in Ballot] \triangleq$
```

This recursively defines $SA[bb]$ to equal $SafeAt(bb, v)$.

```
          $\lor bb = 0$
          $\lor \exists Q \in Quorum :$
               $\land \forall a \in Q : maxBal[a] \geq bb$
               $\land \exists c \in -1 \,..\, (bb-1) :$
                    $\land (c \neq -1) \Rightarrow \land SA[c]$
                                          $\land \forall a \in Q :$
                                               $\forall w \in Value :$
                                                  $VotedFor(a, c, w) \Rightarrow (w = v)$
                    $\land \forall d \in (c+1) \,..\, (bb-1), a \in Q : DidNotVoteIn(a, d)$
  IN   $SA[b]$
 }
```

There are two possible actions that an acceptor can perform, each defined by a macro. In these macros, *self* is the acceptor that is to perform the action. The first action, *IncreaseMaxBal*(b) allows acceptor *self* to set *maxBal*[*self*] to b if b is greater than the current value of *maxBal*[*self*].

```
  macro $IncreaseMaxBal(b)$ {
    when $b > maxBal[self]$ ;
    $maxBal[self] := b$
}
```

Action $VoteFor(b, v)$ allows acceptor *self* to vote for value v in ballot b if its **when** condition is satisfied.

```
macro  $VoteFor(b, v)$ {
  when  $\land\ maxBal[self] \leq b$
        $\land\ DidNotVoteIn(self, b)$
        $\land\ \forall p \in Acceptor \setminus \{self\} :$
              $\forall w \in Value : VotedFor(p, b, w) \Rightarrow (w = v)$
        $\land\ SafeAt(b, v)$ ;
  $votes[self] := votes[self] \cup \{\langle b, v \rangle\}$ ;
  $maxBal[self] := b$
  }
```

The following process declaration asserts that every process *self* in the set *Acceptor* executes its body, which loops forever nondeterministically choosing a Ballot b and executing either an $IncreaseMaxBal(b)$ action or nondeterministically choosing a value v and executing a $VoteFor(b, v)$ action. The single label indicates that an entire execution of the body of the **while** loop is performed as a single atomic action.

From this intuitive description of the process declaration, one might think that a process could be deadlocked by choosing a ballot b in which neither an $IncreaseMaxBal(b)$ action nor any $VoteFor(b, v)$ action is enabled. An examination of the TLA$^+$ translation (and an elementary knowledge of the meaning of existential quantification) shows that this is not the case. You can think of all possible choices of b and of v being examined simultaneously, and one of the choices for which a step is possible being made.

```
  process ($acceptor \in Acceptor$) {
    $acc$ : while (TRUE) {
              with ($b \in Ballot$) {
                either  $IncreaseMaxBal(b)$
                or      with ($v \in Value$) { $VoteFor(b, v)$ }
              }
           }
  }
}
```

3 Algorithm PCon

The algorithm is easiest to understand in terms of the set *msgs* of all messages that have ever been sent. A more accurate model would use one or more variables to represent the messages actually in transit, and it would include actions representing message loss and duplication as well as message receipt.

For our purposes, there is no need to model message loss explicitly. The safety part of the spec says only what messages may be received and does not assert that any message actually is received. Thus, there is no difference between a lost message and one that is never received. The liveness property of the spec will make it clear what messages must be received (and hence either not lost or successfully retransmitted if lost) to guarantee progress.

Another advantage of maintaining the set of all messages that have ever been sent is that it allows us to define the state function *votes* that implements the variable of the same name in the voting algorithm without having to introduce a history variable.

In addition to the variable *msgs*, the algorithm uses four variables whose values are arrays indexed by acceptor, where for any acceptor a:

$maxBal[a]$	The largest ballot number in which a has participated
$maxVBal[a]$	The largest ballot number in which a has voted, or -1 if it has never voted.
$maxVVal[a]$	If a has voted, then this is the value it voted for in ballot $maxVBal$; otherwise it equals *None*.

As in the voting algorithm, an execution of the algorithm consists of an execution of zero or more ballots. Different ballots may be in progress concurrently, and ballots may not complete (and need not even start). A ballot b consists of the following actions (which need not all occur in the indicated order).

Phase1a	The leader sends a $1a$ message for ballot b.
Phase1b	If $maxBal[a] < b$, an acceptor a responds to the $1a$ message by setting $maxBal[a]$ to b and sending a $1b$ message to the leader containing the values of $maxVBal[a]$ and $maxVVal[a]$.
Phase1c	When the leader has received ballot-b $1b$ messages from a quorum, it determines some set of values that are safe at b and sends $1c$ messages for them.
Phase2a	The leader sends a $2a$ message for some value for which it has already sent a ballot-b $1c$ message.
Phase2b	Upon receipt of the $2a$ message, if $maxBal[a] \leq b$, an acceptor a sets $maxBal[a]$ and $maxVBal[a]$ to b, sets $maxVVal[a]$ to the value in the $2a$ message, and votes for that value in ballot b by sending the appropriate $2b$ message.

```
--algorithm PCon {
  variables $maxBal\ = [a \in Acceptor \mapsto -1]$ ,
            $maxVBal = [a \in Acceptor \mapsto -1]$ ,
            $maxVVal = [a \in Acceptor \mapsto None]$ ,
            $msgs = \{\}$
  define {

    $sentMsgs(t,\ b) \triangleq \{m \in msgs : (m.type = t) \land (m.bal = b)\}$
```

We define *ShowsSafeAt* so that $ShowsSafeAt(Q, b, v)$ is true for a quorum Q iff *msgs* contain ballot-b $1b$ messages from the acceptors in Q showing that v is safe at b.

```
    $ShowsSafeAt(Q,\ b,\ v) \triangleq$
      LET $Q1b \triangleq \{m \in sentMsgs(\text{"1b"},\ b) : m.acc \in Q\}$
      IN  $\land\ \forall\, a \in Q : \exists\, m \in Q1b : m.acc = a$
          $\land\ \lor\ \forall\, m \in Q1b : m.mbal = -1$
            $\lor\ \exists\, m1c \in msgs :$
                $\land\ m1c = [type \mapsto \text{"1c"},\ bal \mapsto m1c.bal,\ val \mapsto v]$
                $\land\ \forall\, m \in Q1b :\ \land\ m1c.bal \geq m.mbal$
                                  $\land\ (m1c.bal = m.mbal) \Rightarrow (m.mval = v)$

  }
```

The Actions

As before, we describe each action as a macro. The leader for process *self* can execute a $Phase1a()$ action, which sends the ballot *self* $1a$ message.

```
   macro $Phase1a()$ { $msgs\ := msgs \cup \{[type \mapsto \text{"1a"},\ bal \mapsto self]\}$ }
```

Acceptor *self* can perform a $Phase1b(b)$ action, which is enabled iff $b > maxBal[self]$. The action sets $maxBal[self]$ to b and sends a phase $1b$ message to the leader containing the values of $maxVBal[self]$ and $maxVVal[self]$.

```
  macro $Phase1b(b)$ {
    when $(b > maxBal[self]) \land (sentMsgs(\text{"1a"}, b) \neq \{\})$ ;
    $maxBal[self]\ := b$ ;
    $msgs\ := msgs \cup \{[type \mapsto \text{"1b"},\ acc \mapsto self,\ bal \mapsto b,$
                  $mbal \mapsto maxVBal[self],\ mval \mapsto maxVVal[self]]\}$ ;
  }
```

The ballot *self* leader can perform a $Phase1c(S)$ action, which sends a set S of $1c$ messages indicating that the value in the *val* field of each of them is safe at ballot b. In practice, S will either contain a single message, or else will have a message for each possible value, indicating that all values are safe. In the first case, the leader will immediately send a $2a$ message with the value contained in that single message. (Both logical messages will be sent in the same physical message.) In the latter case, the leader is informing the acceptors that all values are safe. (All those logical messages will, of course, be encoded in a single physical message.)

```
macro $Phase1c(S)$ {
   when $\forall v \in S : \exists Q \in Quorum : ShowsSafeAt(Q, self, v)$ ;
   $msgs := msgs \cup \{ [type \mapsto \text{"1c"}, bal \mapsto self, val \mapsto v] : v \in S \}$
}
```

The ballot *self* leader can perform a $Phase2a(v)$ action, sending a $2a$ message for value v, if it has not already sent a $2a$ message (for this ballot) and it has sent a ballot *self* $1c$ message with *val* field v.

```
macro $Phase2a(v)$ {
   when $\land\ sentMsgs(\text{"2a"}, self) = \{\}$
        $\land\ [type \mapsto \text{"1c"}, bal \mapsto self, val \mapsto v] \in msgs$ ;
   $msgs := msgs \cup \{ [type \mapsto \text{"2a"}, bal \mapsto self, val \mapsto v] \}$
}
```

The $Phase2b(b)$ action is executed by acceptor *self* in response to a ballot-b $2a$ message. Note this action can be executed multiple times by the acceptor, but after the first one, all subsequent executions are stuttering steps that do not change the value of any variable.

```
macro $Phase2b(b)$ {
   when $b \geq maxBal[self]$ ;
   with ($m \in sentMsgs(\text{"2a"}, b)$) {
      $maxBal[self]\ \ := b$ ;
      $maxVBal[self] := b$ ;
      $maxVVal[self] := m.val$ ;
      $msgs := msgs \cup \{ [type \mapsto \text{"2b"}, acc \mapsto self, bal \mapsto b, val \mapsto m.val] \}$
   }
}
```

An acceptor performs the body of its **while** loop as a single atomic action by nondeterministically choosing a ballot in which its *Phase*1*b* or *Phase*2*b* action is enabled and executing that enabled action. If no such action is enabled, the acceptor does nothing.

```
 process ($acceptor \in Acceptor$) {
   acc: while (TRUE) {
           with ($b \in Ballot$) { either $Phase1b(b)$ or $Phase2b(b)$ }
        }
 }
```

The leader of a ballot nondeterministically chooses one of its actions that is enabled (and the argument for which it is enabled) and performs it atomically. It does nothing if none of its actions is enabled.

```
 process ($leader \in Ballot$) {
   ldr: while (TRUE) {
           either $Phase1a()$
           or     with ($S \in$ SUBSET $Value$) { $Phase1c(S)$ }
           or     with ($v \in Value$) { $Phase2a(v)$ }
        }
 }
}
```

4 Algorithm BPCon

In the abstract algorithm BPCon, we do not specify how acceptors learn what $1b$ messages have been sent. We simply introduce a variable *knowsSent* such that *knowsSent*$[a]$ represents the set of $1b$ messages that (good) acceptor a knows have been sent, and have an action that nondeterministically adds sent $1b$ messages to this set.

--algorithm *BPCon* **{**

The variables:

maxBal$[a]$	The highest ballot in which acceptor a has participated.
maxVBal$[a]$	The highest ballot in which acceptor a has cast a vote (sent a $2b$ message), or -1 if it hasn't cast a vote.
maxVVal$[a]$	The value acceptor a has voted for in ballot *maxVBal*$[a]$, or *None* if *maxVBal*$[a] = -1$.
2*avSent*$[a]$	A set of records in $[val : Value,\ bal : Ballot]$ describing the $2av$ messages that a has sent. A record is added to this set, and any element with the same val field (and lower bal field) removed when a sends a $2av$ message.
knownSent$[a]$	The set of $1b$ messages that acceptor a knows have been sent.
bmsgs	The set of all messages that have been sent. See the discussion of the msgs variable in module PConProof to understand our modeling of message passing.

```
variables $maxBal = [a \in Acceptor \mapsto -1]$ ,
          $maxVBal = [a \in Acceptor \mapsto -1]$ ,
          $maxVVal = [a \in Acceptor \mapsto None]$ ,
          $2avSent = [a \in Acceptor \mapsto \{\}]$ ,
          $knowsSent = [a \in Acceptor \mapsto \{\}]$ ,
          $bmsgs = \{\}$
define {
  $sentMsgs(type, bal) \triangleq \{m \in bmsgs : m.type = type \land m.bal = bal\}$

  $KnowsSafeAt(ac, b, v) \triangleq$
```

True for an acceptor ac, ballot b, and value v iff the set of $1b$ messages in *knowsSent*$[ac]$ implies that value v is safe at ballot b in the Paxos consensus algorithm being emulated by the good acceptors. To understand the definition, see the definition of ShowsSafeAt in module PConProof and recall (a) the meaning of the *mCBal* and *mCVal* fields of a $1b$ message and (b) that the set of real acceptors in a *ByzQuorum* forms a quorum of the PCon algorithm.

```
LET $S \triangleq \{m \in knowsSent[ac] : m.bal = b\}$
IN    $\lor \exists BQ \in ByzQuorum :$
          $\forall a \in BQ : \exists m \in S : \land m.acc = a$
                                    $\land m.mbal = -1$
      $\lor \exists c \in 0 \,..\, (b-1) :$
          $\land \exists BQ \in ByzQuorum :$
              $\forall a \in BQ : \exists m \in S : \land m.acc = a$
                                        $\land m.mbal \leq c$
                                        $\land (m.mbal = c) \Rightarrow (m.mval = v)$
          $\land \exists WQ \in WeakQuorum :$
              $\forall a \in WQ :$
                $\exists m \in S : \land m.acc = a$
                            $\land \exists r \in m.m2av : \land r.bal \geq c$
                                                  $\land r.val = v$
```

{

We now describe the processes' actions as macros.

As in the Paxos consensus algorithm, a ballot *self* leader (good or malicious) can execute a Phase1a action at any time.

```
macro $Phase1a()$ { $bmsgs := bmsgs \cup \{[type \mapsto \text{"1a"},\ bal \mapsto self]\}$ }
```

The acceptor's Phase1b ation is similar to that of the PCon algorithm.

```
macro $Phase1b(b)$ {
  when $(b > maxBal[self]) \land (sentMsgs(\text{"1a"}, b) \neq \{\})$ ;
  $maxBal[self] := b$ ;
  $bmsgs := bmsgs \cup \{[type \mapsto \text{"1b"},\ bal \mapsto b,\ acc \mapsto self,\ m2av \mapsto 2avSent[self],$
                        $mbal \mapsto maxVBal[self],\ mval \mapsto maxVVal[self]]\}$
  }
```

A good ballot *self* leader can send a phase $1c$ message for value v if it knows that the messages in $knowsSent[a]$ for a Quorum of (good) acceptors imply that they know that v is safe at ballot *self*, and that they can convince any other acceptor that the appropriate $1b$ messages have been sent to that it will also know that v is safe at ballot *self*.

A malicious ballot *self* leader can send any phase $1c$ messages it wants (including one that a good leader could send). We prove safety with a $Phase1c$ action that allows a leader to be malicious. To prove liveness, we will have to assume a good leader that sends only correct $1c$ messages.

As in the *PCon* algorithm, we allow a $Phase1c$ action to send a set of $Phase1c$ messages. (This is not done in the Castro-Liskov algorithm, but seems natural in light of the *PCon* algorithm.)

```
macro $Phase1c()$ {
  with $(S \in \text{SUBSET}\ [type : \{\text{“1c”}\},\ bal : self,\ val : Value])$ {
    $bmsgs := bmsgs \cup S$
  }
}
```

If acceptor *self* receives a ballot b phase $1c$ message with value v, it relays v in a phase $2av$ message if

- it has not already sent a $2av$ message in this or a later ballot and
- the messages in $knowsSent[self]$ show it that v is safe at b in the non-Byzantine Paxos consensus algo- rithm being emulated.

```
macro $Phase2av(b)$ {
  when $\land\ maxBal[self] =< b$
       $\land\ \forall r \in 2avSent[self] : r.bal < b$ ;
```

We could just as well have used $r.bal \neq b$ in this condition.

```
  with $(m \in \{ms \in sentMsgs(\text{“1c”}, b) : KnowsSafeAt(self, b, ms.val)\})$ {
    $bmsgs := bmsgs\ \cup\ \{[type \mapsto \text{“2av”},\ bal \mapsto b,\ val \mapsto m.val,\ acc \mapsto self]\}$;
    $2avSent[self] := \{r \in 2avSent[self] : r.val \neq m.val\}\ \cup\ \{[val \mapsto m.val,\ bal \mapsto b]\}$
  } ;
  $maxBal[self] := b$
}
```

Acceptor *self* can send a phase $2b$ message with value v if it has received phase $2av$ messages from a Byzantine quorum, which implies that a quorum of good acceptors assert that this is the first $1c$ message sent by the leader and that the leader was allowed to send that message. It sets $maxBal[self]$, $maxVBal[self]$, and $maxVVal[self]$ as in the non-Byzantine algorithm.

```
macro $Phase2b(b)$ {
  when $maxBal[self] =< b$ ;
  with $(v \in \{vv \in Value :$
              $\exists Q \in ByzQuorum :$
                $\forall aa \in Q :$
                  $\exists m \in sentMsgs(\text{“2av”}, b) : \land\ m.val = vv$
                                                  $\land\ m.acc = aa\})$ {
    $bmsgs := bmsgs\ \cup\ \{[type \mapsto \text{“2b”},\ acc \mapsto self,\ bal \mapsto b,\ val \mapsto v]\}$ ;
    $maxVVal[self] := v$
  } ;
  $maxBal[self] := b$ ;
```

```
    $maxVBal[self] := b$
  }
```

At any time, an acceptor can learn that some set of $1b$ messages were sent (but only if they atually were sent).

```
  macro $LearnsSent(b)$ {
    with $(S \in \text{SUBSET } sentMsgs(\text{"1b"}, b))$ { $knowsSent[self] := knowsSent[self] \cup S$ }
  }
```

A malicious acceptor *self* can send any acceptor message indicating that it is from itself. Since a malicious acceptor could allow other malicious processes to forge its messages, this action could represent the sending of the message by any malicious process.

```
  macro $FakingAcceptor()$ {
    with $(m \in \{mm \in 1bMessage \cup 2avMessage \cup 2bMessage : mm.acc = self\})$ {
      $bmsgs := bmsgs \cup \{m\}$
    }
  }
```

We combine these individual actions into a complete algorithm in the usual way, with separate process declarations for the acceptor, leader, and fake acceptor processes.

```
  process $(acceptor \in Acceptor)${
    $acc$: while (TRUE) {
          with $(b \in Ballot)$ {
            either $Phase1b(b)$ or $Phase2av(b)$ or $Phase2b(b)$ or $LearnsSent(b)$
          }
        }
  }

  process $(leader \in Ballot)$ {
    $ldr$: while (TRUE) { either $Phase1a()$ or $Phase1c()$ }
  }

  process $(facceptor \in FakeAcceptor)$ {
    $facc$: while (TRUE) {$FakingAcceptor()$ }
  }
}
```

References

[1] Leslie Lamport. Mechanically checked safety proof of a byzantine paxos algorithm. URL `http://research.microsoft.com/users/lamport/tla/byzpaxos.html`. The page can also be found by searching the Web for the 23-letter string obtained by removing the "`-`" from `uid-lamportbyzpaxosproof`.

[2] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In Srikanta Tirthapura and Lorenzo Alvisi, editors, *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009*, pages 312–313. ACM, 2009.