Chapter 2

Consensus & Classic Paxos

We begin our study of distributed consensus by first considering how to decide upon a single value between a set of participants. This task, whilst seemingly simple, will occupy us for the majority of this thesis. Single-valued agreement is often overlooked in the literature as already solved or trivial and is seldom considered at length, despite being a vital component in distributed systems which is infamously poorly understood.

This chapter is broadly divided into three parts. We begin by defining the requirements for an algorithm to solve distributed consensus (§2.1). Secondly, we outline two existing algorithms which solve single valued consensus: the single acceptor algorithm (§2.1.1), a naïve straw man solution and Classic Paxos (§2.2, §2.3, §2.4), the widely adopted solution which lies at the foundation of a broad range of complex distributed systems. Thirdly and finally, we go on to prove that both algorithms satisfy all of the requirements of distributed consensus as defined in the first part (§2.5,2.6,2.7).

2.1 Preliminaries

Single valued distributed consensus is the problem of deciding a single value $v \in V$ between a finite set of n participants, $U = \{u_1, u_2, \ldots, u_n\}$.

Definition 1. *An algorithm is said to solve distributed consensus only if it satisfies the following three safety requirements:*

Non-triviality *The decided value must have been proposed by a participant.*

Safety *If a value has been decided, no other value will be decided.*

Safe learning *If a participant learns a value, it must learn the decided value.*

In addition to the following two progress requirements[1]*:*

Progress *Under a specified set of liveness conditions, if a value has been proposed by a participant then a value is eventually decided.*

Eventual learning *Under a specified set of liveness conditions, if a value has been decided then a value is eventually learned.*

Together, these five requirements prevent many trivial algorithms from satisfying distributed consensus. Without the safety and safe learning requirement, then a consensus algorithm could decide/learn all values proposed by participants. If non-triviality was not required, then a consensus algorithm could simply decide a fixed value. If progress and eventual learning were not required, then a consensus algorithm could never decide a value, rejecting all proposals it receives or never allowing any participant to learn the value. These trivial approaches are of little interest thus all five of the stated requirements are necessary.

It is important to note that the safety requirements do not rely on any liveness conditions. In other words, failures or asynchrony cannot lead to a violation of safety thus the algorithm cannot depend upon bounded clock drift, message delay, or execution time.

In contrast, progress can rely upon specified liveness conditions such as partial synchrony. The liveness conditions are always sufficient for the algorithm to make progress, regardless of the state of the system. In other words, the algorithm cannot become indefinitely stuck in deadlock (or livelock).

Notice that none of the requirements restrict which proposal value is decided. Specifically, an algorithm for distributed consensus is free to choose from any of the proposed values, regardless of which participant proposed the value, the proposal order, the number of participants proposing the same value or the proposed values themselves. The only restriction is that, under the progress condition, a value must eventually be decided if at least one value has been proposed. Therefore, from the non-triviality condition, we observe that if only one value is proposed then it must eventually be chosen.

In this chapter, we will formulate the problem of consensus in the usual manner adopted in the academic literature[2]. Each participant in the system is assigned one or both of the following two roles.

- Proposer - A participant who wishes to have a particular value chosen.

[1]Note, that this definition of progress is more general than the one commonly found in the literature which is specific to majorities. This generalisation aims to decouple majorities (a common aspect of consensus algorithms) from the problem definition.

[2]This approach is taken to aid readers who are already familiar with the field, although, it can be ambiguous at times.

- Acceptor - A participant who agrees and persists decided values.

In a system U of n participants, we will denote the set acceptors as $A = \{a_1, a_2, \dots\}$ where $A \subseteq U$ and $|A| = n_a$ and the set of proposers as $P = \{p_1, p_2, \dots\}$ where $P \subseteq U$ and $|P| = n_p$. A consensus algorithm defines the process by which a value v is chosen by the acceptors from the proposers. We refer to the point in time when the acceptors have committed to a particular value as the *commit point*. After this point in time, v has been decided and cannot be subsequently altered. The proposers learn which value has been decided, this must always take place after the commit point has been reached.

If we are able to reach agreement over a single value, we are able to reach agreement over an infinite sequence of values $v_1, v_2, v_3, \dots$ [3] by independently reaching consensus over each value in the sequence in turn. This sequence could represent updates to a re-writeable register, operations for a replicated state machine, messages for atomic broadcast, a shared log or state changes in a primary-backup system.

All notation introduced in this section and the remainder of the thesis is summarised for reference in Table 2.1.

2.1.1 Single acceptor algorithm

In the section, we introduce a straw-man algorithm which solves distributed consensus. The algorithm, which we will refer to as the *single acceptor algorithm (SAA)*, requires that exactly one participant be assigned the role of acceptor[4]. The liveness conditions for SAA are that the acceptor and at least one proposer are up and can exchange messages reliably. This algorithm is included here to familiarise the reader with the terminology and methodology before we progress to more advanced algorithms.

The single acceptor algorithm chooses the first value proposed by the proposers. A proposer who has a candidate value γ will propose the value to the acceptor using the message *propose(γ)*. If this is the first proposal the acceptor has received, it will write γ to persistent storage (known as *accepting*) and notify the proposer that the value has been decided using the message *accept(γ)*. Otherwise, if this is not the first proposal to be received, the acceptor will reply to the proposer with the already decided value γ' using *accept(γ')*. In either case, provided that the acceptor is available then the proposer will learn the decided value. See algorithm 1 and algorithm 2 for pseudocode descriptions of this approach[5].

[3] The sequence of values will always be 1-indexed.

[4] We are not the first to use this straw-man solution to explain the problem of consensus [Lam01a, §2.2]

[5] Note that for all pseudocode in this thesis, variables are stored in volatile memory and are initially *nil* unless otherwise stated. Also note that all pseudocode in this thesis favours clarity and consistency over performance.

Notation	Description	First use:
$u_1, u_2, \ldots$	participants	2.1
$a_1, a_2, \ldots / p_1, p_2, \ldots$	specific acceptors/proposers	2.1
$n / n_a / n_p$	number of participants/acceptors/proposers	2.1
$v, w, x, \ldots$	values	2.1
$v_1, v_2, \ldots$	sequence of values	2.1
$a, a', \ldots / p, p', \ldots$	acceptors/proposers	2.1.1
$A, B, C \ldots$	concrete values	2.1.1
γ, γ'	candidate values	2.1.1
v_{acc}	last accepted value	2.1.1
$e, f, g, \ldots$	epochs	2.2
(e, v)	proposal with epoch e and value v	2.2
e_{min} / e_{max}	minimum/maximum epoch	2.2
e_{pro} / e_{acc}	last promised/accepted epoch	2.2
v_{dec}	decided value	3.3
$pid / sid / vid$	proposer ID/sequence ID/version ID	3.8
p_{lst}	last proposer	3.9
$U / A / P$	set of participants/acceptors/proposers	2.1
V	set of values	2.1
E	set of epochs	2.2
$\mathcal{E}$	set of unused epochs	2.2
Q_P / Q_A	set of acceptors which have promised/accepted	2.2
Q_V	set of acceptors which have promised with e_{max}	3.2
Γ	set of candidate values	2.2
$Q, Q', \ldots$	quorums (set of acceptors)	3.11
$\mathcal{Q}, \mathcal{Q}', \ldots$	quorum set	3.11
$\mathcal{Q}_i^e$	quorum set for phase i and epoch e	4.1
V_{dec}	set of values which maybe decided	6.1
R	mapping from acceptors to a promise, *no* or (e, v)	6.1
D	mapping from quorums to decisions	6.1
$min(\mathcal{E})$	returns minimum epoch in $\mathcal{E}$	2.2
$succ(e)$	returns successor of epoch e	3.8
$only(V)$	returns the only element in singleton set V	6.1

Table 2.1: Reference table of notation.

Algorithm 1: Proposer algorithm for SAA

state:

- γ: candidate value (configured, persistent)

1 **send** *propose(γ) to acceptor*
2 **case** *accept(v) received from acceptor*
`/* proposer learns that` v `was decided so return` v `*/`
3 **return** v

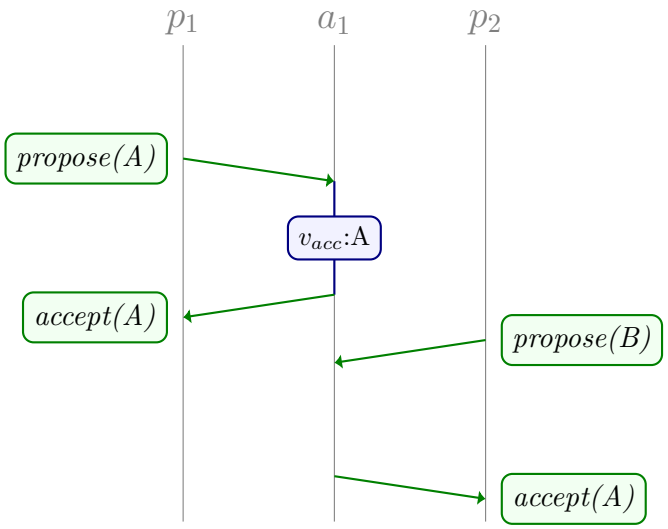

Figure 2.1: Example run of SAA between one acceptor $\{a_1\}$ and two proposers $\{p_1, p_2\}$.

Algorithm 2: Acceptor algorithm for SAA

state:

- v_{acc}: accepted value (persistent)

1 **while** *true* **do**
2 **case** *propose(v) received from proposer*
3 **if** $v_{acc} = nil$ **then**
4 $v_{acc} \leftarrow v$
5 **send** *accept(v_{acc}) to proposer*

Figure 2.1 is a message sequence diagram (MSD) for an example execution of the single acceptor algorithm. We will make extensive use of MSDs to illustrate the messages exchanged and state updates which occur over time. Note that the time axis (the negative y-axis) is not assumed to be linear. In this example, proposer p_1 has candidate value $\gamma = A$ and proposer p_2 has candidate value $\gamma = B$. The acceptor receives *propose(A)* first and therefore the value A is decided.

Safety

Informally, we can see that this simple algorithm satisfies the three safety conditions for distributed consensus. The acceptor chooses the first proposal it receives thus it satisfies non-triviality. After accepting its first proposal, the acceptor accepts no other proposals from proposers thus this algorithm satisfies safety. If a proposer returns a value, then the value must have been received from the acceptor and therefore must be decided, satisfying safe learning. We will now consider these in more detail.

Theorem 1 (Non-triviality of SAA)**.** *If the value v is decided, then v must have been proposed by a proposer*

Proof of theorem 1. Assume the value v is decided. For v to be decided, the acceptor must have accepted the proposal *propose(v)*. Thus since messages cannot be corrupted, v must have been proposed by some proposer □

Theorem 2 (Safety & safe learning of SAA)**.** *For any two proposers $p, p' \in P$, which learn that the decided value v is γ and γ' respectively then $\gamma = \gamma'$.*

Proof of theorem 2. A proposer p learns that the decided value v is γ as a result of receiving *accept(γ)* from the single acceptor. The same is true for any other participant p'.

Since the events of sending *accept(γ)* and sending *accept(γ')* occur on one participant, the single acceptor, the two events cannot have occurred concurrently. Thus one event must happen before the other.

Assume that the event send *accept(γ)* is before send *accept(γ')*.

The acceptor determines the values γ, γ' by reading the accepted value v_{acc}. If $\gamma \neq \gamma'$ then the value v_{acc} would have changed from γ to γ' between sending the two accepted messages. The only mechanism for the accepter to update v_{acc} to γ' is by receiving *accept(γ')*. Updating v_{acc} is conditional on v_{acc} being nil, since v_{acc} is persistent, it cannot be nil as it has previously been set to γ. Therefore v_{acc} cannot have been updated between the two sending events so $\gamma = \gamma'$.

The same applies when send *accept(γ')* is before send *accept(γ)*. □

Progress

Informally, we can see that this simple algorithm also satisfies the two progress conditions for distributed consensus, under the liveness conditions that the acceptor and at least one proposer must be up. Note, that though we use the assumption that messages are eventually delivered, we do not require time bounds on message delivery or operating speed.

Theorem 3 (Progress of SAA)**.** *If a proposer $p \in P$ proposes the value γ and the liveness conditions are satisfied for a sufficient period then a value v is eventually decided.*

Proof of theorem 2. Assume proposer p sends *propose(γ)* to the acceptor. Under the liveness conditions, this message will be eventually received by the acceptor. Under the liveness conditions, this acceptor must be up and handle the message. Either no decision has yet been reached thus the proposal is accepted and $v = \gamma$, otherwise a decision has already been reached and $v = v_{acc}$. □

Summary

This simple algorithm provides consensus in one round trip (two messages) to the acceptor and one synchronous write to persistent storage, provided the acceptor is up. If the acceptor is down, then the system cannot progress until the acceptor is up. This algorithm works as all value proposals intersect at a single point, the acceptor. The result is that proposals are totally ordered so choosing a proposal is trivial. However, this reliance on a single acceptor is also this algorithm's downfall. Should this acceptor fail, the algorithm could not make progress until it recovers.

The acceptor in SAA is a single point of failure, the obvious step to address this is to have multiple acceptors. However, we can no longer guarantee a total ordering over proposals so the single acceptor algorithm is no longer suitable[6]. In the next section, we instead describe Classic Paxos, a consensus algorithm which is able to handle multiple acceptors.

2.2 Classic Paxos

Classic Paxos [Lam98][7] is an algorithm[8] for solving the problem of distributed consensus. In the best case, the unoptimised algorithm is able to reach agreement in two round trips to the majority of acceptors and three synchronous writes to persistent storage, though in some cases more time will be needed. The liveness conditions is that $\lfloor n_a/2 \rfloor + 1$ of n_a acceptors and one proposer must be up and communicating synchronously. These conditions are both necessary and sufficient for progress.

The approach taken by Classic Paxos to deciding a value has two phases. Phase one can be viewed as the reading phase, where the proposer learns about the current state of the system and takes a type of version number to detect changes in the future. Phase two can be viewed as the writing phase, where the proposer tries to get a value accepted. If, after phase one of the algorithm, the proposer is certain that a value has not yet been decided,

[6] Assuming the network does not provide atomic broadcast.

[7] Also known as *Synod* or *Single-degree Paxos*

[8] More correctly, it is a family of algorithms

the proposer can propose the candidate value γ. If the outcome of phase one is that a value might already be decided, then that value must be proposed in phase two instead. Each of these two phases requires a majority of acceptors to agree in order to proceed.

We now define the terms *epoch* and *proposal* and then use these to summarise the Classic Paxos algorithm.

Definition 2. *An epoch e is any member of the set of epochs E. E is any infinite totally ordered set such that the operators $<$, $>$ and $=$ are always defined* [9].

Definition 3. *A proposal (e, v) is any epoch and value pair*[10].

Classic Paxos Phase 1

1. A proposer chooses a unique epoch e and sends *prepare(e)* to the acceptors.

2. Each acceptor stores the last promised epoch and last accepted proposal. When an acceptor receives *prepare(e)*, if e is the first epoch promised or if e is equal to or greater than the last epoch promised, then e is written to storage and the acceptor replies with *promise(e,f,v)*. *(f,v)* is the last accepted proposal (if present) where f is the epoch and v is the corresponding proposed value.

3. Once the proposer receives *promise(e,_,_)* from the majority of acceptors, it proceeds to phase two. Promises may include a last accepted proposal which will be used by the next phase.

4. Otherwise if the proposer times out, it will retry with a greater epoch.

Classic Paxos Phase 2

1. The proposer must now select a value v using the following *value selection rules*:

   i If no proposals were returned with promises in phase one, then the proposer will choose its candidate value γ.

   ii If one proposal was returned, then its value is chosen.

   iii If more than one proposal was returned then the proposer must choose the value associated with the greatest epoch.

   The proposer then sends *propose(e,v)* to the acceptors.

[9] Epochs are also referred to as terms [OO14, §5.1], view numbers [LC12, §3], round numbers [MPSP10, §3], instance values/epoch [HKJR10, §1] or ballot numbers in the literature

[10] Proposals are also referred to as ballots in the literature

2. Each acceptor receives a *propose(e,v)*. If e is the first epoch promised or if e is equal to or greater than the last promised epoch, then the promised epoch and accepted proposal is updated and the acceptor replies with *accept(e)*.

3. Once the proposer receives *accept(e)* from the majority of acceptors, it learns that the value v is decided.

4. Otherwise if the proposer times out, it will retry phase 1 with a greater epoch.

	Message	**Description**	**Sent by:**	**Received by:**
Phase 1	*prepare(e)*	e: epoch	proposers	acceptors
	promise(e,f,v)	e: epoch f: last accepted epoch* v: last accepted value* *maybe *nil*	acceptors	proposers
Phase 2	*propose(e,v)*	e: epoch v: proposal value	proposers	acceptors
	accept(e)	e: epoch	acceptors	proposers

Table 2.2: Messages exchanged in Classic Paxos

For reference, Table 2.2 gives an overview of the four messages used in Classic Paxos[11].

We will now look at this process in more detail.

2.2.1 Proposer algorithm

Algorithm 3 describes the Classic Paxos algorithm for participants with the role of a proposer. The key input to this algorithm is a candidate value γ to propose and the output is the decided value v. The decided value may or may not be the same as the candidate value, depending upon the state of acceptors when the algorithm is executed. The proposer will only propose its candidate value γ if it is sure that another value has not already been chosen. Once the proposer learns that a value has been decided, no proposer will learn that a different value has been decided.

After initialising its variables (Algorithm 3, lines 1-2), the algorithm begins by selecting a epoch e to use (Algorithm 3, line 3). To remain general, we do not specify how the set of available epochs, $\mathcal{E} \subseteq E$, should be generated. However, the algorithm does require that each proposer is configured with an infinite disjoint set of epochs. The algorithm ensures that each epoch is used only once, by removing the current epoch, e, from the set

[11] These message are often referred to as 1a, 1b, 2a and 2b respectively. Confusingly, the propose message is called prepare in VRR [LC12, §4.1]

Algorithm 3: Proposer algorithm for Classic Paxos

state:

- n_a: total number of acceptors (configured, persistent)
- e: current epoch
- v: current proposal value
- e_{max}: maximum epoch received in phase 1
- $\mathcal{E}$: set of unused epochs (configured, persistent)
- Q_P: set of acceptors who have promised
- Q_A: set of acceptors who have accepted

`/* (Re)set variables */`

1 $v, e_{max} \leftarrow nil$
2 $Q_P, Q_A \leftarrow \emptyset$
`/* Select and set the epoch` e `*/`
3 $e \leftarrow min(\mathcal{E})$
4 $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
`/* Start Phase 1 for epoch` e `*/`
5 **send** *prepare(e) to acceptors*
6 **while** $|Q_P| < \lfloor n_a/2 \rfloor + 1$ **do**
7 **switch do**
8 **case** *promise(e,f,w) received from acceptor a*
9 $Q_P \leftarrow Q_P \cup \{a\}$
10 **if** $f \neq nil \wedge (e_{max} = nil \vee f > e_{max})$ **then**
 `/*` (e_{max}, v) `is the greatest proposal received */`
11 $e_{max} \leftarrow f$, $v \leftarrow w$
12 **case** *timeout*
13 **goto** *line 1*
14 **if** $v = nil$ **then**
 `/* no proposals were received thus propose` γ `*/`
15 $v \leftarrow \gamma$
`/* Start Phase 2 for proposal (`e`,`v`) */`
16 **send** *propose(e,v) to acceptors*
17 **while** $|Q_A| < \lfloor n_a/2 \rfloor + 1$ **do**
18 **switch do**
19 **case** *accept(e) received from acceptor a*
20 $Q_A \leftarrow Q_A \cup \{a\}$
21 **case** *timeout*
22 **goto** *line 1*
23 **return** v

of available epochs (Algorithm 3, line 4). For simplicity, we have the proposers try epochs in-order, though it is safe for the proposer to use any unused epoch.

The message *prepare(e)* is sent to all acceptors (Algorithm 3, line 5) and the proposer waits for responses. As promises are received, the proposer tracks the maximum epoch, e_{max}, received in a proposal and its associated value, v (Algorithm 3, lines 8-11). If a promise does not include a proposal then the maximum epoch, e_{max}, and its associated value, v, are not updated (Algorithm 3, line 10). The set $\mathcal{Q}_P$ tracks which acceptors have promised thus far. If promises are not received from a majority of acceptors before a timeout then the algorithm retries (Algorithm 3, lines 6, 12-13). If no proposals were received with promises, the proposal value v is set to the candidate value γ (Algorithm 3, lines 14-15).

The proposer then sends *propose(e,v)* to the acceptors (Algorithm 3, line 16). The proposer will return value v (Algorithm 3, line 23) after the majority of acceptors accept the proposal (e, v) (Algorithm 3, lines 17-20) and retry otherwise (Algorithm 3, lines 21-22).

Note that all other messages which are received by the proposer but do not match a switch statement (such as messages from previous epochs or promises during phase two) can be safely ignored.

2.2.2 Acceptor algorithm

Algorithm 4: Acceptor algorithm for Classic Paxos

state:
- e_{pro}: last promised epoch (persistent)
- e_{acc}: last accepted epoch (persistent)

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if e_pro = nil ∨ e ≥ e_pro then
5                  e_pro ← e
6                  send promise(e,e_acc,v_acc) to proposer
7          case propose(e,v) received from proposer
8              if e_pro = nil ∨ e ≥ e_pro then
9                  e_pro ← e
10                 v_acc ← v, e_acc ← e
11                 send accept(e) to proposer
```

The acceptors in Classic Paxos are responsible for handling incoming *prepare* and *propose* messages. The logic for this is described in Algorithm 4[12]. All messages, whether *prepare(e)*

[12] Algorithm 4 uses the variable v_{acc}, however it is not included in the state list. Due to space limitations,

or *propose(e,v)*, must have an epoch e greater than or equal to e_{pro} to be processed by the acceptor (Algorithm 4, lines 4,8). If this is the first message the acceptor has received then e_{pro} is *nil* and this test is always successful. If the test is successful then e_{pro} is updated to e (Algorithm 4, lines 5,9).

If the message was *prepare(e)*, then the acceptor replies with *promise(e,e_{acc},v_{acc})* (Algorithm 4, line 6). If the acceptor has not yet accepted a proposal then e_{acc} and v_{acc} will be *nil*. When an acceptor sends a promise message, we say that the acceptor has *promised* epoch e[13].

If the message was *propose(e,v)* then the acceptor will set e_{acc} and v_{acc} to the proposal (e, v) (Algorithm 4, line 10) and reply with *accept(e)* (Algorithm 4, lines 11). In this case, we say that the acceptor has *accepted* the proposal (e, v).

Definition 4. *In Classic Paxos, a proposal* (e, v) *is decided if the proposal* (e, v) *has been accepted by the majority of acceptors.*

Note that this definition does not require that the proposal is still the last accepted proposal on a majority of acceptors. A value $v \in V$ is said to be decided if there exists an epoch $e \in E$ such the proposal (e, v) has been decided. This is also described as value v is *decided* in e. The *commit point* is the first time a proposal is decided.

2.3 Examples

In this section, we will consider example message sequence diagrams (MSDs) for a sample of possible executions of Classic Paxos. For simplicity, messages are omitted if their receipt will have no effect. Each example system is comprised of three acceptors $A = \{a_1, a_2, a_3\}$ and two proposers $P = \{p_1, p_2\}$, thus $\lfloor n_a/2 \rfloor + 1 = 2$. Initially, $\gamma = A$ for proposer p_1 and $\gamma = B$ for proposer p_2.

In our examples, epochs are natural numbers $E = \mathbb{N}^0$, which have been divided round robin between the proposers. Therefore initially $\mathcal{E} = \{0, 2, 4, \dots\}$ on p_1 and $\mathcal{E} = \{1, 3, 5, \dots\}$ on p_2.

Figure 2.2 gives an example of two proposers executing Classic Paxos in serial. Firstly, proposer p_1 executes Classic Paxos and the proposal $(0, A)$ is decided. Then proposer p_2 executes Classic Paxos and the proposal $(1, A)$ is decided. Both proposers are able to complete Classic Paxos in two phases. This represents the best case scenario for Classic Paxos.

the state list for each algorithm only includes new variables. The descriptions of variables such as v_{acc} can be found in Table 2.1.

[13]The term *adopts* is sometimes used in the literature instead of promised, for example in [VRA15]

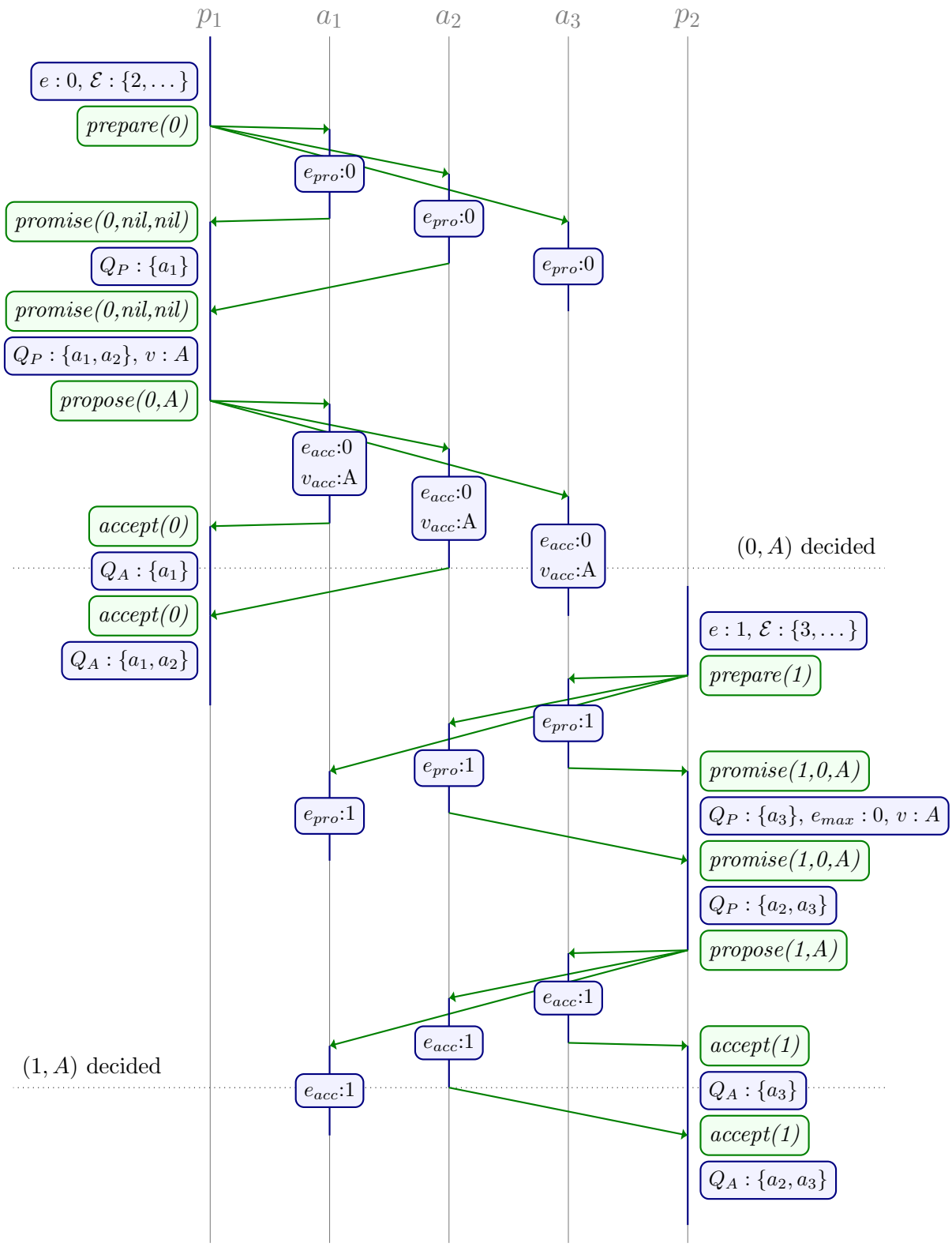

Figure 2.2: Example run of Classic Paxos with two serial proposers. Proposer p_1 executes Classic Paxos followed by the proposer p_2.

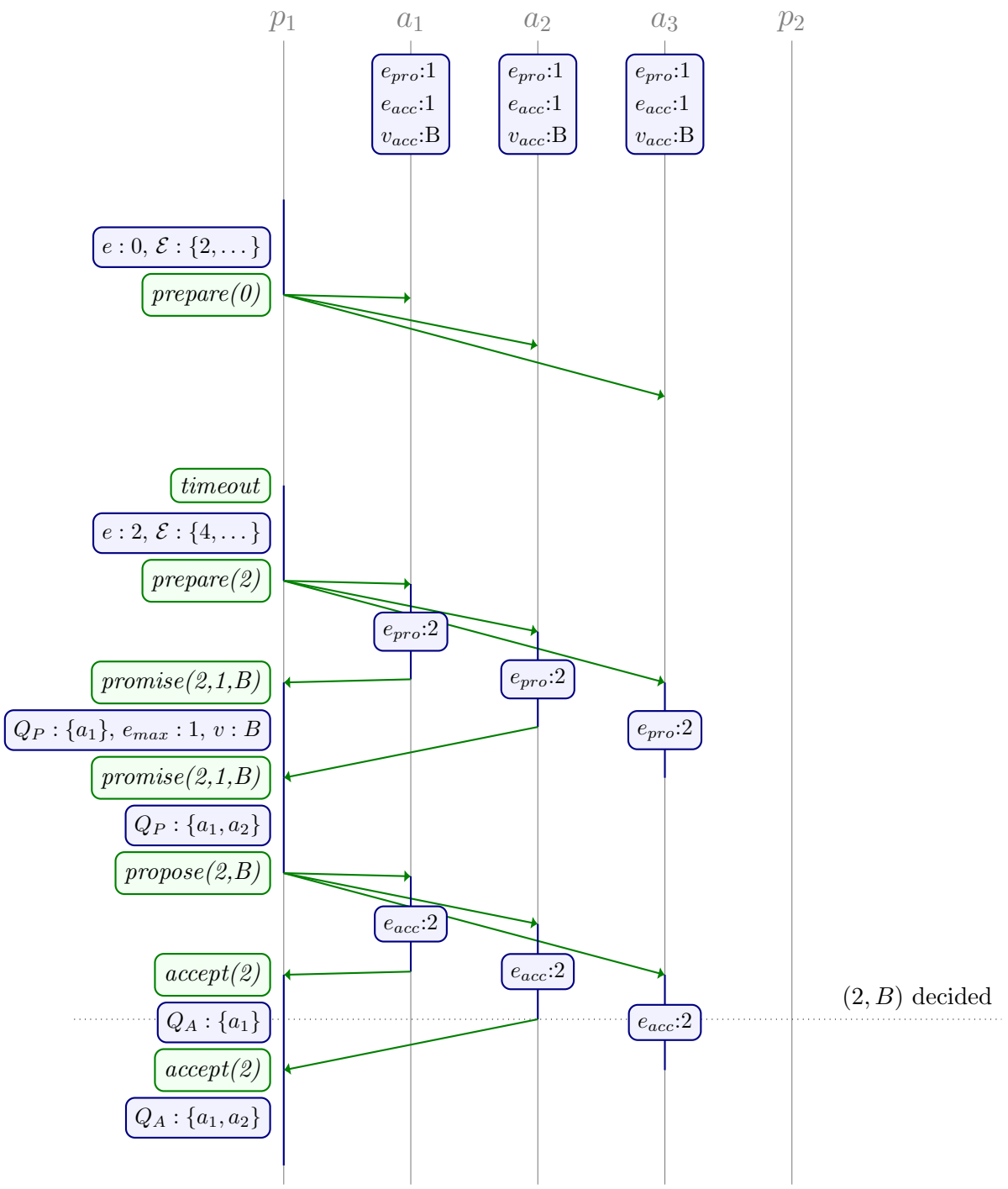

Figure 2.3: Example run of Classic Paxos with two serial proposers. Proposer p_2 has finished executing Classic Paxos before proposer p_1 begins.

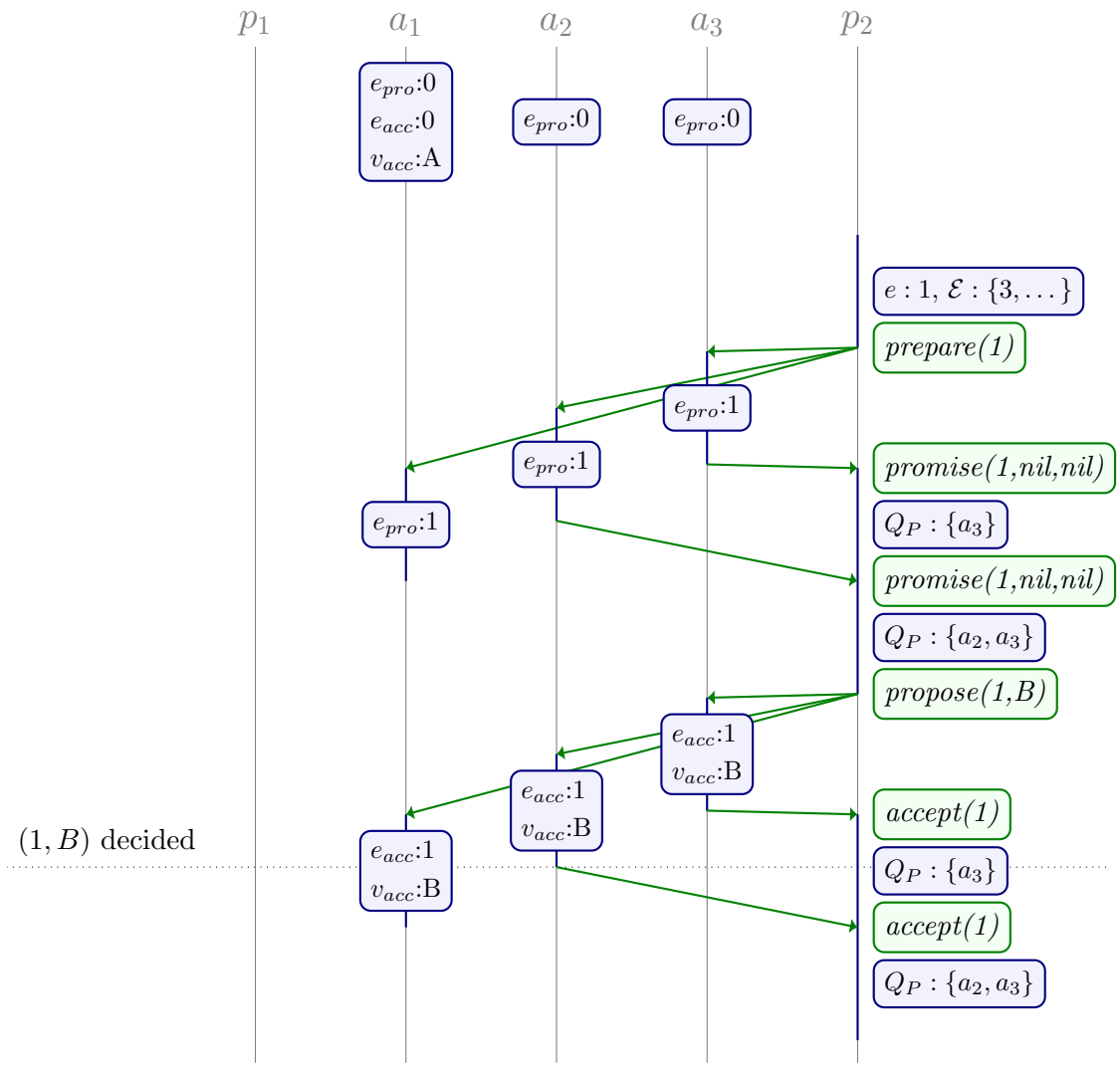

Figure 2.4: Example run of Classic Paxos where proposer p_1 stops during phase two prior to reaching the commit point. Proposer p_2 does not observe the proposal from p_1.

Initially, in Figure 2.3, proposer p_2 has executed Classic Paxos and the proposal $(1, B)$ has been decided and accepted by all acceptors. Subsequently the proposer p_1 executes phase one for epoch 0, however this phase is unsuccessful. The proposer p_1 retries Classic Paxos and the proposal $(2, B)$ is decided. Unlike before, proposer p_1 in this example required three phases to learn the decided value.

Figures 2.4 and 2.5 illustrate two possible outcomes if a proposer (in this case p_1) stops after making a proposal (in this case $(0, A)$) but prior to reaching the commit point. In Figure 2.4, proposer p_2 does not observe the proposal $(0, A)$ during its phase one thus the proposal $(1, B)$ is subsequently decided. In contrast, in Figure 2.5 the proposer p_2 does observe the proposal $(0, A)$ during its phase one thus the proposal $(1, A)$ is subsequently decided.

The examples thus far have demonstrated proposers executing Classic Paxos in serial. In Figure 2.6, we observe the worst case scenario of Classic Paxos when concurrent proposers

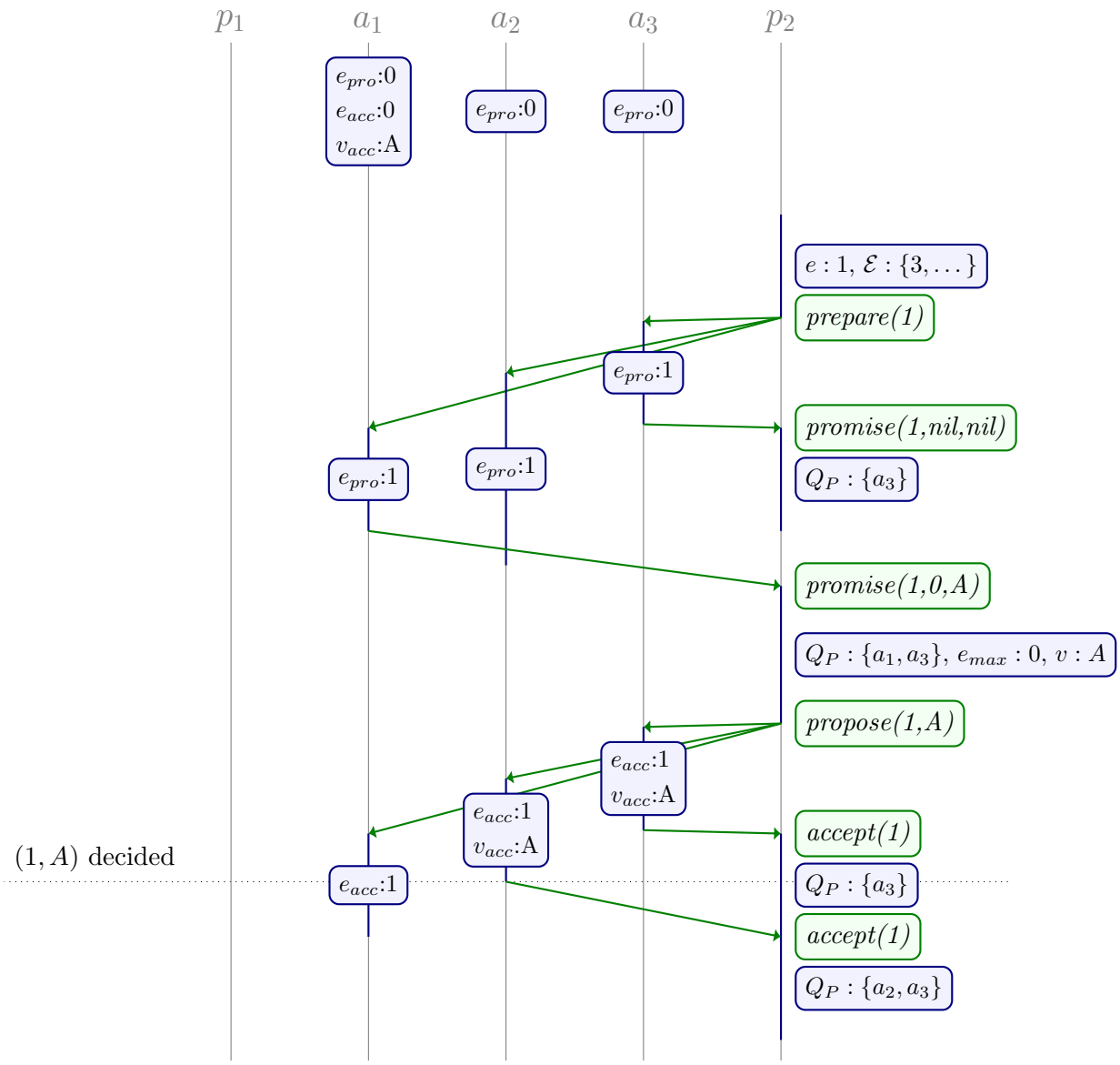

Figure 2.5: Example run of Classic Paxos where proposer p_1 stops during phase two prior to reaching the commit point. Proposer p_2 does observe the proposal from p_1.

duel such that neither proposer is able to make progress. Proposer p_1 executes phase one for epoch 0 then proposer p_2 executes phase one for epoch 1. Proposer p_1 is unsuccessful at phase two for proposal $(0, A)$ thus executes phase one for epoch 2. Proposer p_2 is then unsuccessful at phase two for proposal $(1, B)$. Though unlikely, this situation could continue indefinitely. Note that this situation can still occur when both proposers are proposing the same value or after a decision has been reached.