Nachrichten:	**ELECT**	**vorläufige Wahl**
	OK	**vorläufige Wahlannahme**
	COORD	**endgültige Wahlannahme**

Wenn Prozess P den Ausfall des Koordinators erkennt, setzt er eine Koordinator-Wahl nach dem folgenden Bully-Algorithmus in Gang:

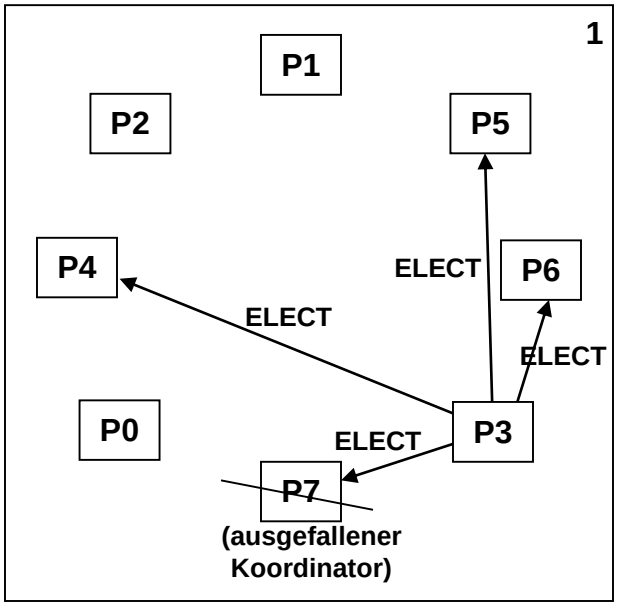
P schickt eine ELECT Message an alle Prozesse mit höherer Prozessnummer als seine eigene und erwartet eine bestätigende OK Message von ihnen, wenn sie aktiv sind.

Wenn P nach einer bestimmten Zeit von keinem dieser Prozesse ein OK erhält, wird er Koordinator und schickt eine COORD Message an alle anderen Prozesse.

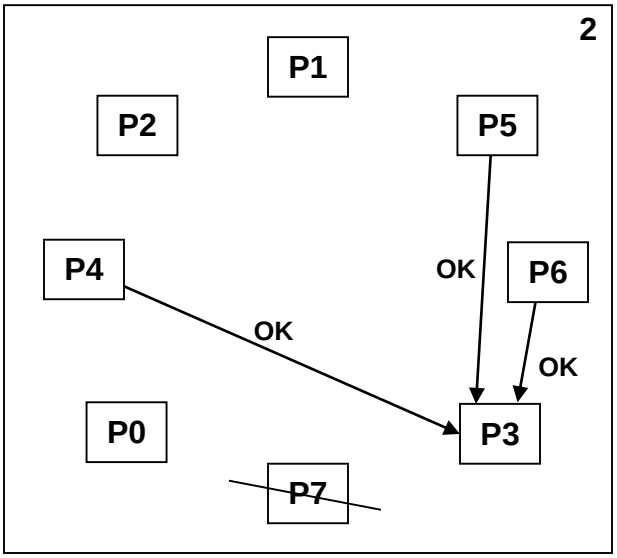
Wenn P von einem Prozess mit höherer Prozessnummer ein OK erhält, scheidet er aus dem Verfahren zur Wahl eines neuen Koordinators aus und wartet nur noch eine gewisse Zeit auf den Empfang eines COORD. Falls er allerdings kein COORD empfängt, startet er die Wahl erneut.

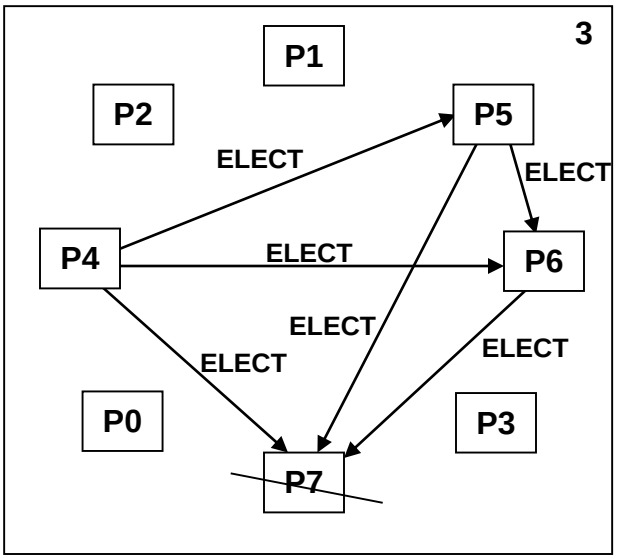
Wenn ein Prozess ein ELECT von einem Prozess mit niedrigerer Prozessnummer erhält, schickt er ein OK zurück und startet eine eigene Wahl nach dem Bully-Algorithmus, es sei denn, er hat seine eigene Wahl schon gestartet.

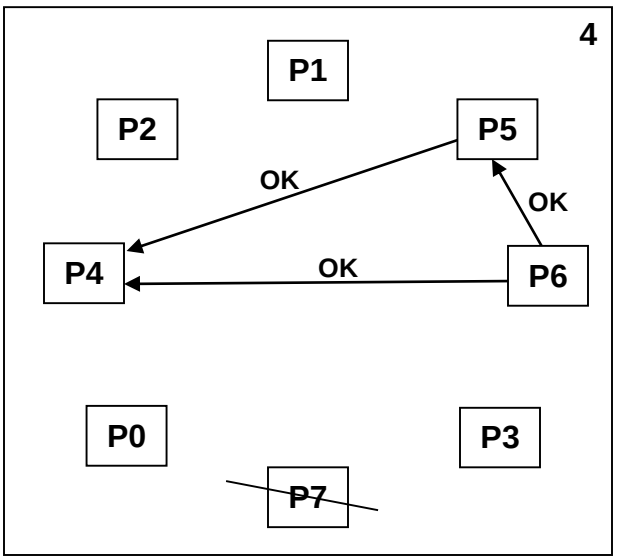
Bully-Algorithmus an Prozess P

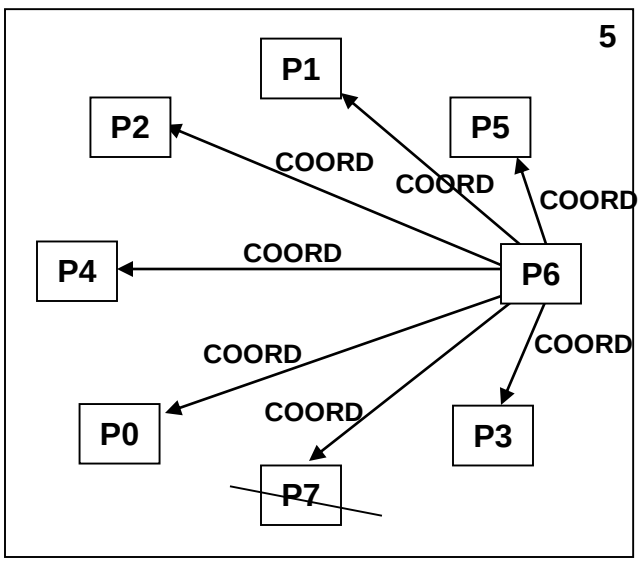


P3 erkennt Koordinator-Ausfall.









P6 erhält keine Antwort auf sein ELECT, ist also neuer Koordinator.

Bully-Algorithmus Beispiel

Nachrichten:	ELECT	Koordinator-Suche
	OK	Bestätigung von ELECT
	COORD	Benachrichtigung über Koordinator

Wahlphase (mit ELECT- und OK-Nachrichten):

Initiierung:

Derjenige Prozess, der das Fehlen eines Koordinators bemerkt, schickt eine ELECT-Nachricht mit seiner Prozessnummer an seinen Nachfolger im logischen Ring.

***ELECT-Empfang*:**

Wenn ein Prozess ein ELECT empfängt, vergleicht er die empfangene Prozessnummer m mit seiner eigenen Prozessnummer n und reagiert folgendermaßen:

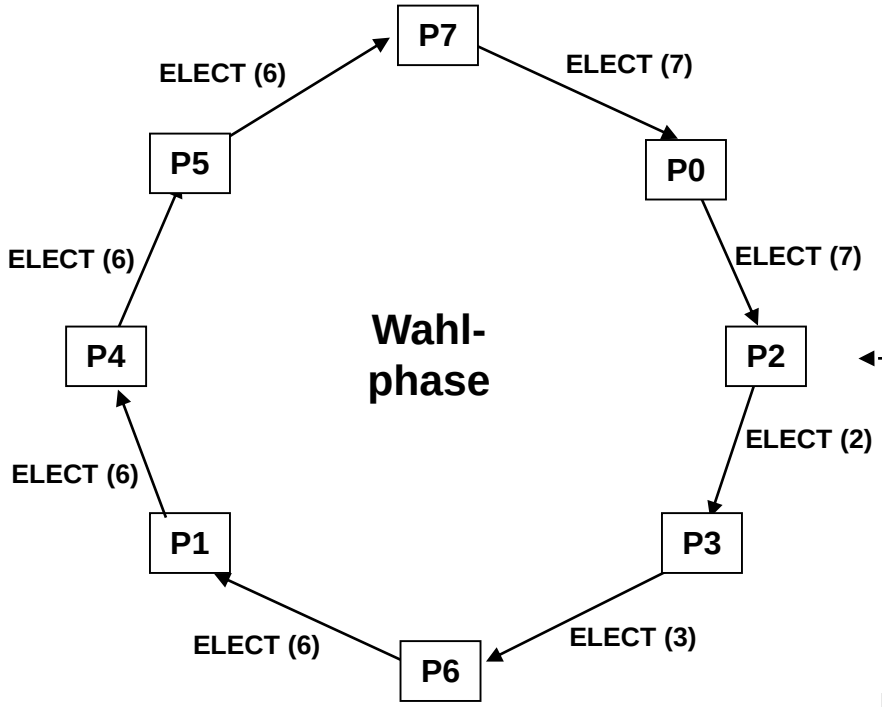
- **Falls m>n, schickt er das ELECT unverändert an seinen Nachfolger weiter.**
- **Falls m<n, schickt er das ELECT mit seiner eigenen Prozessnummer an seinen Nachfolger.**
- **Falls m=n, ist er Koordinator und leitet die folgende Koordinationsphase ein.**

Eine ELECT-Nachricht muss durch OK bestätigt werden. Trifft keine Bestätigung ein, muss sie an den nächsten Nachfolger wiederholt werden (dafür muss ihm dieser allerdings bekannt sein).

Koordinationsphase (mit COORD-Nachrichten):

Der neue Koordinator schickt eine COORD-Nachricht mit seiner Prozessnummer durch den Ring.

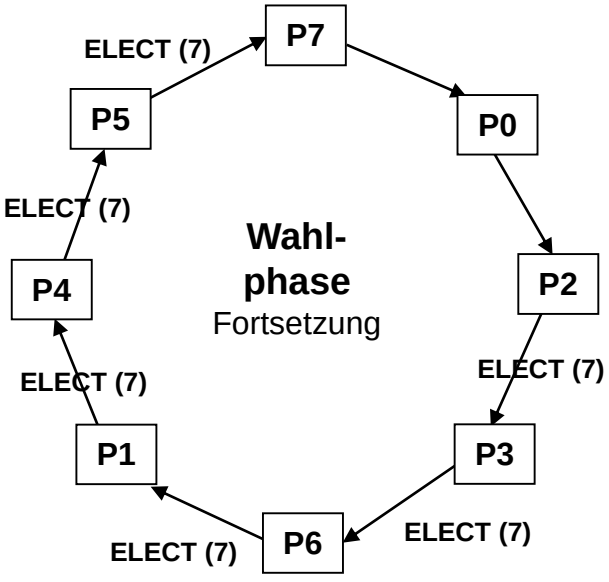
Ring-Algorithmus

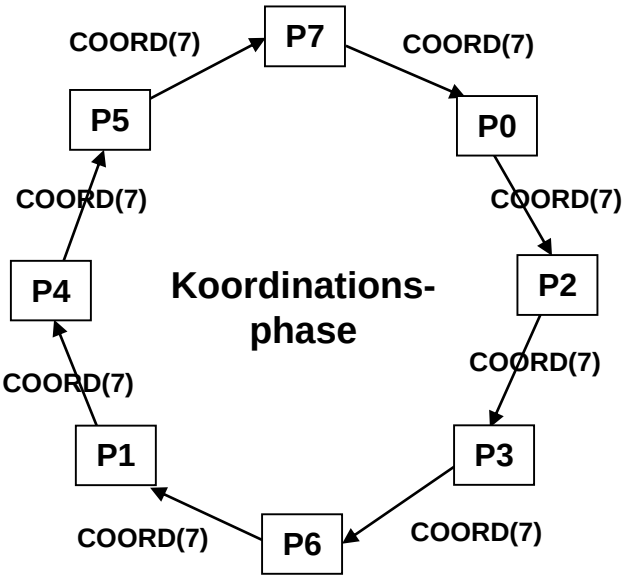


P2 bemerkt Koordinator-Ausfall und initiiert Koordinator-Wahl durch Aussenden von ELECT(2)

ELECT-Nachrichten müssen bestätigt werden (OK-Nachricht).

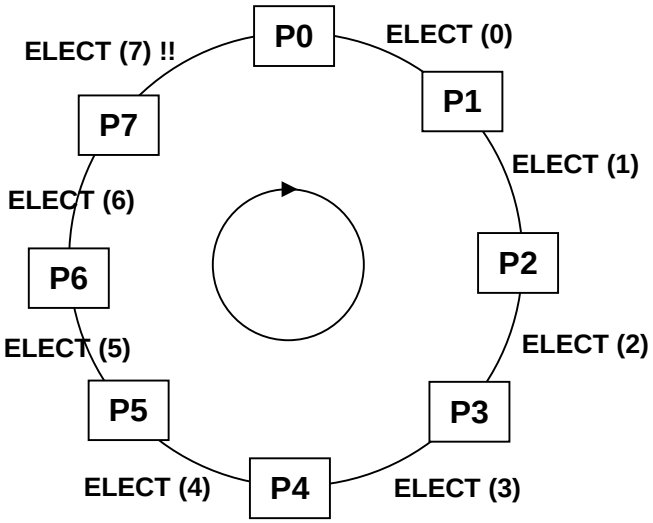
P7 erkennt eigene Nummer wieder: er ist neuer Koordinator



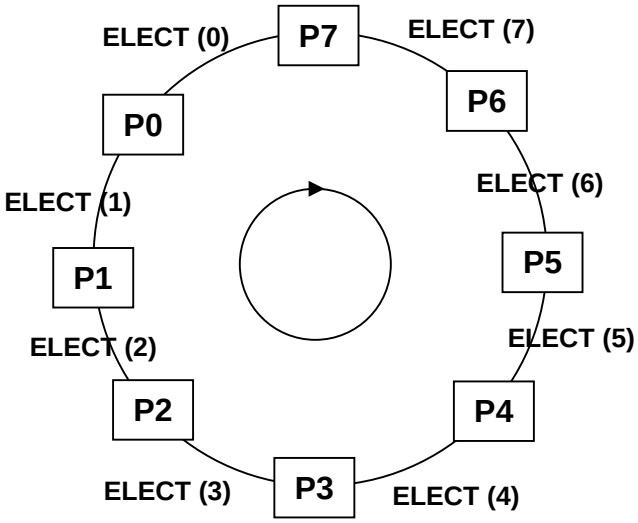


Ring-Algorithmus Beispiel

Alle Prozesse starten den Wahlgorithmus im Ring mit N Prozessen.
ELECT-Empfang:
m>n: Schicke das ELECT unverändert an seinen Nachfolger weiter.
m<n: Schicke das ELECT nicht weiter.
m=n: Werde Koordinator und leite die folgende Koordinationsphase ein.



Best Case:
ELECT(0), ... ELECT(6):
jeweils nur eine Übertragung
⇒ N-1 Übertragungen
ELECT(7): N Übertragungen
(einmal durch den gesamten Ring),
insgesamt 2N-1 Übertragungen
⇒ Komplexität *O*(N)



Worst Case:
ELECT(0): 1 Übertragung
ELECT(1): 2 Übertragungen
ELECT(2): 3 Übertragungen
...
ELECT(7): 8 Übertragungen
(jeweils bis Prozess 7)
Σ: 1+2+...+8 = 36 Übertragungen
Allgemein: $\sum_{i=1}^{N} i$ = N(N+1)/2 Übertragungen
⇒ Komplexität *O*(N²)

Average Case:
längere Rechnung [Chang, Roberts, CACM May 1979]: Komplexität *O*(N logN)

Ring-Algorithmus
Komplexität Nachrichtenanzahl

Betriebsmittel (Ressource):

Element (Objekt, Ding), das von einem Prozess zu seiner korrekten Ausführung benötigt wird.

Betriebsmittel können physikalischen oder logischen Charakter haben.

Beispiele: Server, Verbindungen, Daten, Variablen, programmiersprachliche Objekte

Kritischer Bereich (kritischer Abschnitt):

Programmabschnitt, in dem durch einen Prozess auf Betriebsmittel zugegriffen wird und der nicht parallel von anderen Prozessen ausgeführt werden darf.

Exklusivität

Zu jedem Zeitpunkt kann sich höchstens ein Prozess in einem kritischen Bereich befinden.

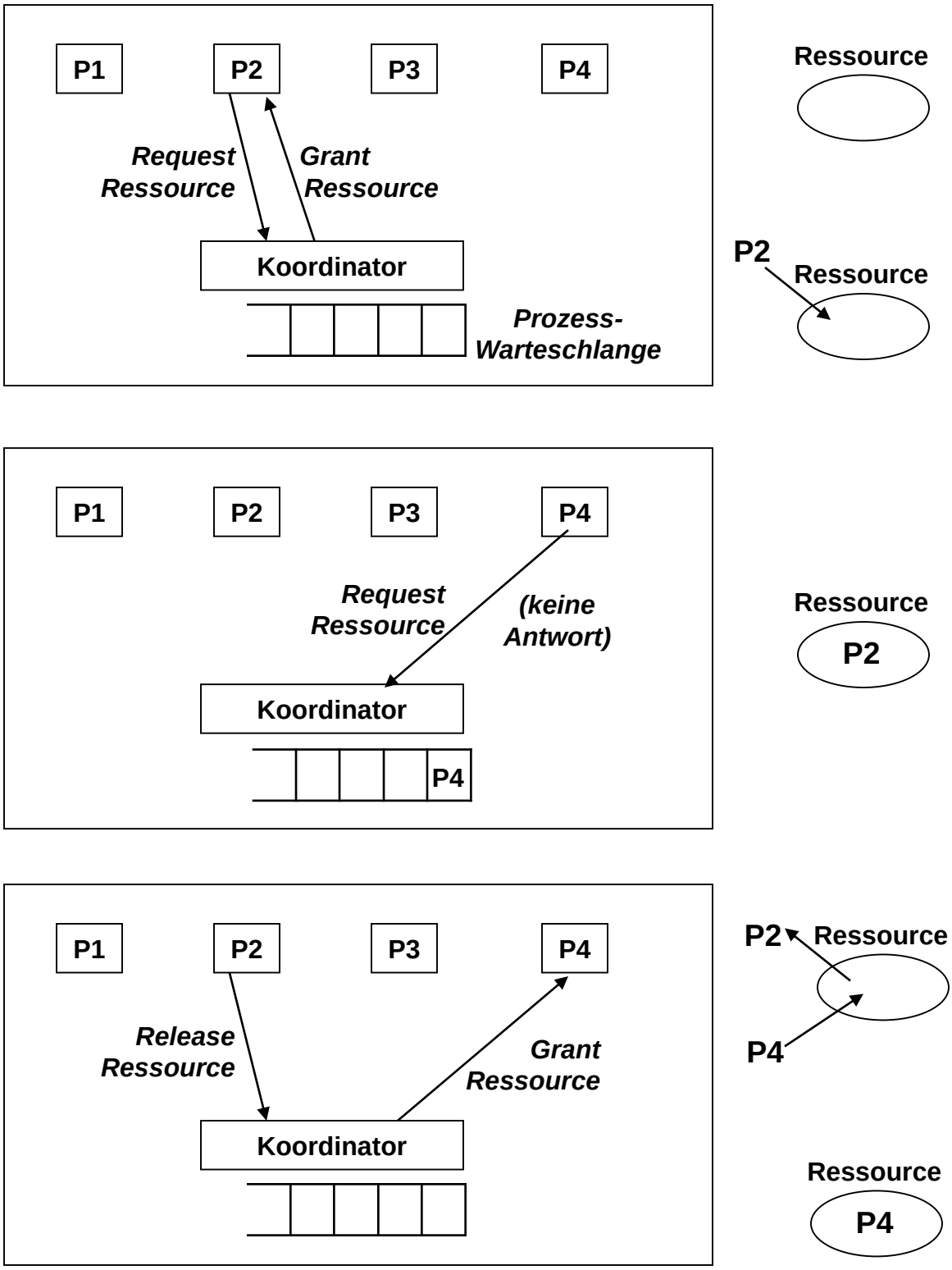
Liveness:

Eine Anforderung eines Prozesses zum Eintritt in einen kritischen Bereich muss nach endlicher Zeit erfolgreich sein.

Reihenfolge-Einhaltung:

Wenn ein Prozess zeitlich vor einem anderen Prozess den Eintritt in einen kritischen Bereich anfordert, muss dies ihm auch zeitlich vor dem anderen Prozess gewährt werden.

Definitionen und Anforderungen für Algorithmen zum wechselseitigen Ausschluss



wechselseitiger Ausschluss: zentraler Algorithmus

Alle beteiligten Prozesse arbeiten mit einer Prozesswarteschlange. Ein Prozess stellt in seine Prozesswarteschlange diejenigen Prozesse ein, die den kritischen Bereich angefordert haben, den er selbst gerade besetzt hält.

Wenn ein Prozess in einen kritischen Bereich eintreten möchte, schickt er eine Anforderungsnachricht REQUEST mit Prozessnummer und Zeitstempel an *alle* Prozesse des verteilten Systems, also auch an sich selbst. Der Empfang eines solchen REQUESTs kann folgende Reaktionen durch den Empfänger auslösen:

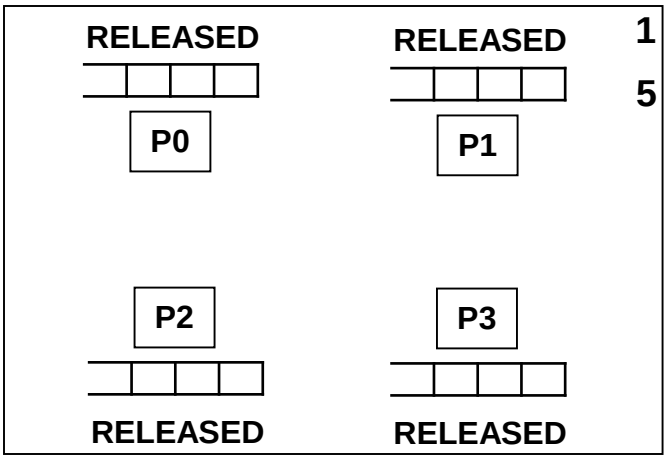
- **befindet er sich im kritischen Bereich, antwortet er nicht und stellt den anfordernden Prozess in seine Warteschlange,**
- **befindet er sich *nicht* im kritischen Bereich und will auch *nicht* ihn in eintreten, antwortet er mit einer OK-Nachricht,**
- **befindet er sich *nicht* im kritischen Bereich, will aber in ihn eintreten, so vergleicht er den Zeitstempel aus der eingehenden Nachricht mit dem aus der REQUEST-Nachricht, die er selbst an die übrigen Komponenten des Systems und sich selbst ausgesendet hat. Je nach Ausgang des Zeitstempel-Vergleichs sendet er eine OK-Nachricht zurück (eigene Anforderung wird zurückgestellt), oder er sendet nichts und stellt den anfordernden Prozess in seine Prozesswarteschlange (eigene Anforderung wird durchgesetzt).**

Sobald ein Prozess auf seine REQUEST-Nachricht hin die OK-Nachrichten *aller* anderen Komponenten erhalten hat, betritt er den kritischen Bereich. Mit Verlassen des kritischen Bereichs schickt er OK-Nachrichten an alle Prozesse in seiner Warteschlange (die also auf diesen jetzt von ihm verlassenen kritischen Bereich gewartet haben) und löscht diese Prozesse aus ihr.

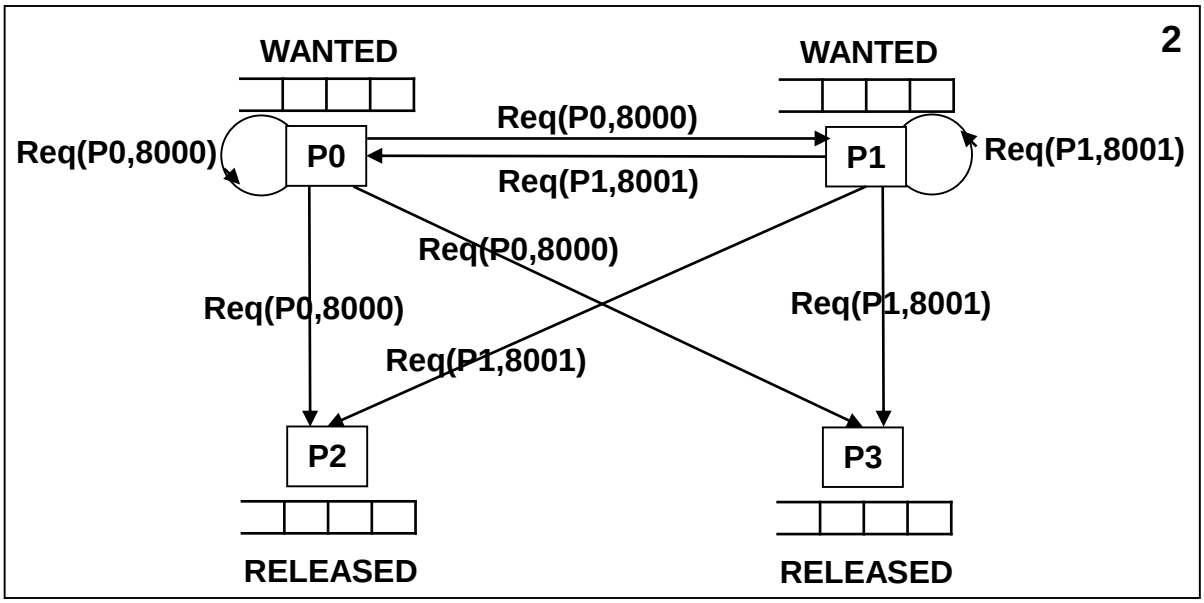
wechselseitiger Ausschluss: Arbeitsweise des verteilten Algorithmus

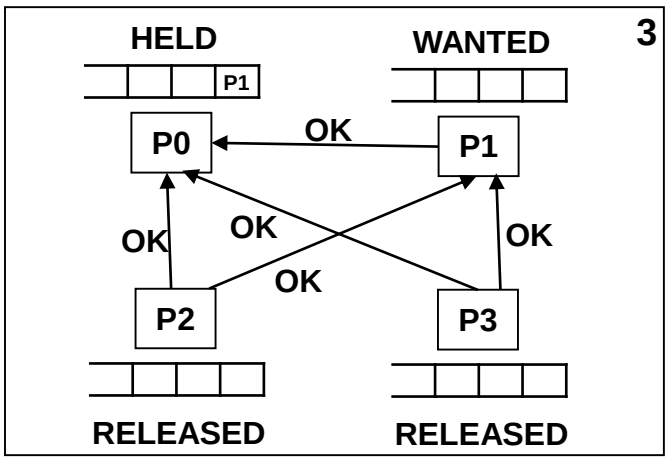
```
// Verteilter Algorithmus für Prozess Pj  im verteilten System
// mit den Prozessen P1, ..., PN


// Zustände von Pj:
//      RELEASED (nicht im kritischen Bereich, Eintritt nicht angefordert),
//      WANTED (nicht im kritischen Bereich, Eintritt angefordert),
//      HELD (im kritischen Bereich befindlich)

// Nachrichten:
//      REQUEST (Anfrage auf einen kritischen Bereich)
//      OK (positive Bestätigung auf einen kritischen Bereich)

// Zeitstempel der Prozesse: T1,...,TN

Initialisierung:
        Zustand = RELEASED;

Eintritt in einen kritischen Bereich:
        Zustand = WANTED;
        sende REQUEST (Pj,Tj) an alle Prozesse;
        Warte, bis (Anzahl der empfangenen OKs == (N-1));
        Zustand = HELD;

Empfang eines REQUEST (Pi, Ti):
        if (Zustand == HELD or (Zustand == WANTED and Tj < Ti))          (*)
            then füge REQUEST(Pi, Ti) ohne Antwort in die Warteschlange ein
            else antworte mit OK an Pi;

Verlassen eines kritschen Bereichs:
        Zustand = RELEASED;
        antworte OK auf alle REQUESTs in der Warteschlange;
        lösche alle REQUESTs aus der Warteschlange;

(*) bei Gleichheit der Zeitstempel entscheidet die Prozessnummer
```

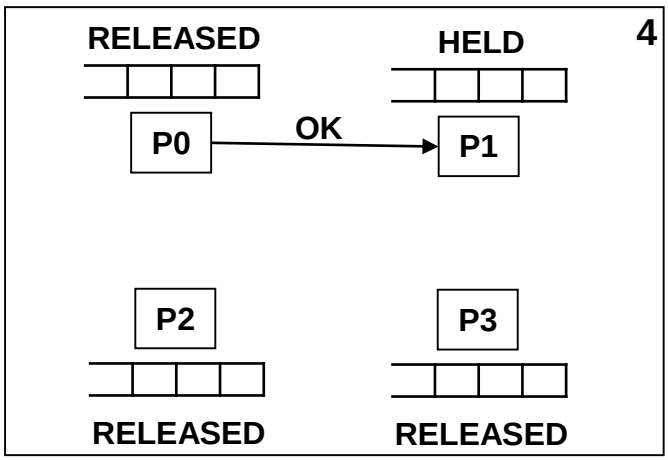
wechselseitiger Ausschluss: verteilter Algorithmus



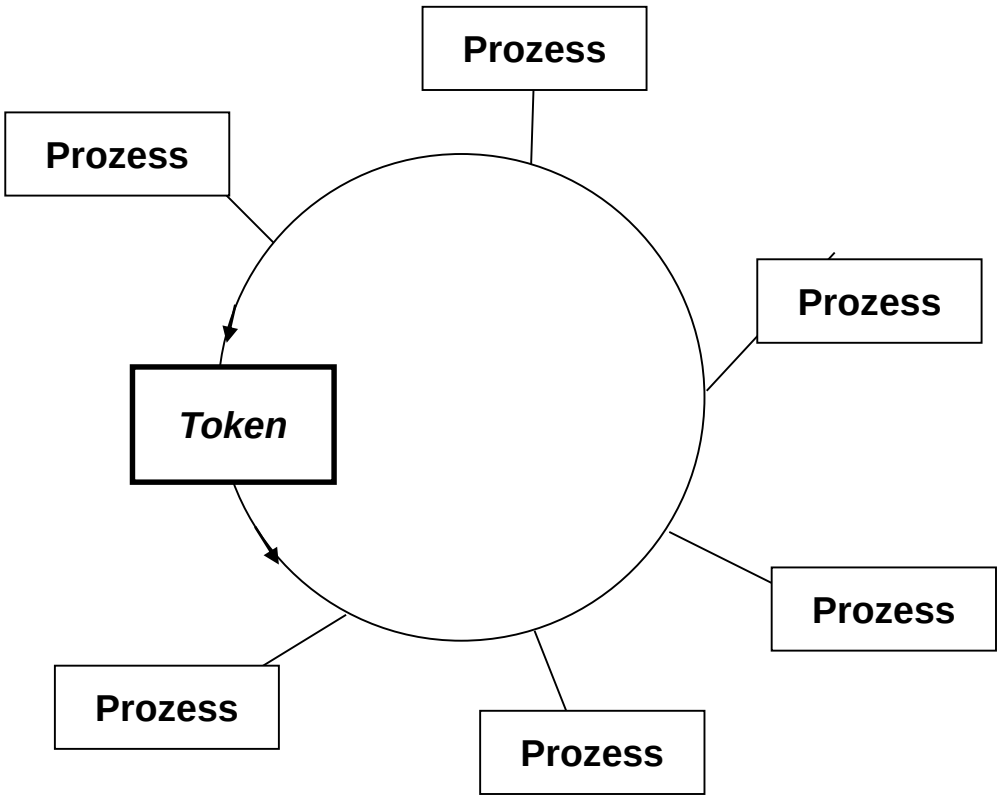
Request-Format:
Req(Prozess#, Zeit)







wechselseitiger Ausschluss: Beispiel zum verteilten Algorithmus



Token = Exklusivrecht auf das Betreten eines kritischen Bereichs

wechselseitiger Ausschluss: Ring-Algorithmus

Deadlock (Verklemmung):

Zustand eines Systems nebenläufiger Prozesse, in dem die Prozesse derart aufeinander warten, dass keiner von ihnen mehr arbeiten kann.

Beispiel:

Dining Philosophers Problem: Alle Philosophen um einen runden Tisch ergreifen gleichzeitig die linke Gabel, benötigen aber zum Essen auch die rechte (die der Nachbar aber schon ergriffen hat).

Livelock (aktive Verklemmung):

Zustand eines Systems nebenläufiger Prozesse, in dem sich die Prozesse derart behindern, dass sie zwar zwischen ihren Zuständen wechseln können, sich aber dennoch gegenseitig blockieren.

Beispiel:

Zwei Personen, die sich auf einem engen Gang entgegenkommen und bis in die Unendlichkeit versuchen, einander auszuweichen, und sich dabei immer gegenseitig blockieren.

Starvation (Verhungern):

Zustand eines Systems nebenläufiger Prozesse, in dem einem oder mehreren Prozessen dauerhaft die Betriebsmittel zum Weiterarbeiten verweigert werden, andere Prozesse im System aber normal arbeiten können.

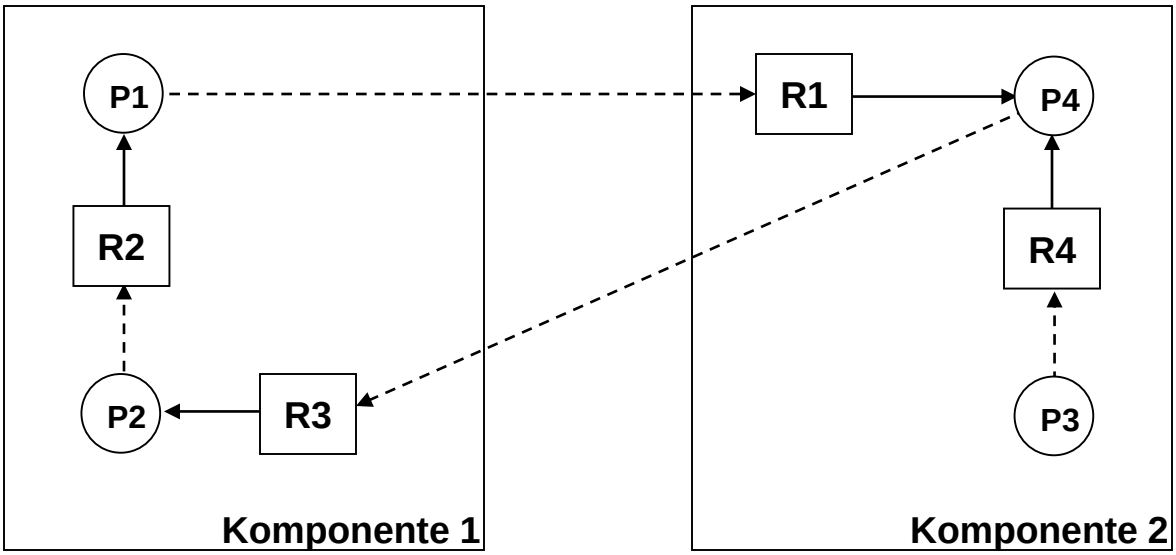
Beispiel:

Eine Station in einem Rechnernetz mit gemeinsam benutztem Übertragungsmedium möchte eine Nachricht niedriger Priorität versenden, wird daran aber dauerhaft dadurch gehindert, dass andere Stationen immer Nachrichten höherer Priorität versenden.

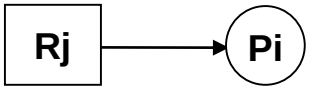
problematische Systemzustände

Exklusivität von Ressourcen-Anforderungen und -Belegungen	**Die Ressourcen werden von den Prozessen für ausschließlichen Gebrauch angefordert und belegt (keine anderen Prozesse dürfen die Ressourcen gleichzeitig belegen).**
Nicht-Unterbrechbarkeit von Ressourcen-Belegungen	**Hat ein Prozess eine Ressource belegt, kann er nicht dazu gezwungen werden, sie wieder freizugeben.**
Fordern und Halten von Ressourcen	**Prozesse können weitere Ressourcen anfordern, auch wenn sie schon Ressourcen halten.**
Zyklisches Warten auf Ressourcen	**Es existiert ein Zyklus aus mindestens zwei Prozessen, von denen jeder eine Ressource angefordert hat, die der nächste schon besetzt hat.**

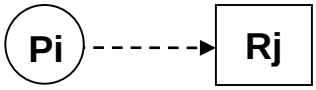
notwendige Bedingungen für Deadlock



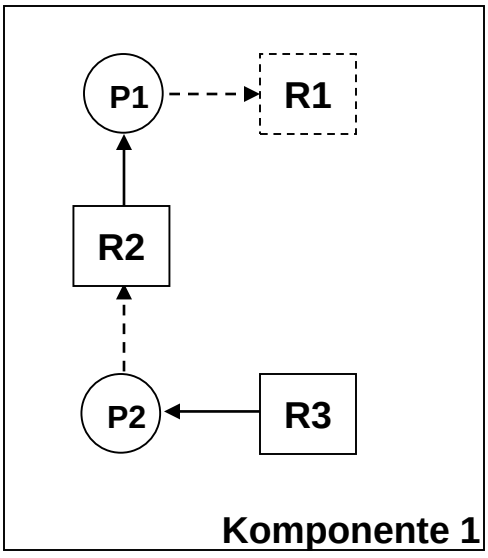
Belegungs-Anforderungs-Graph



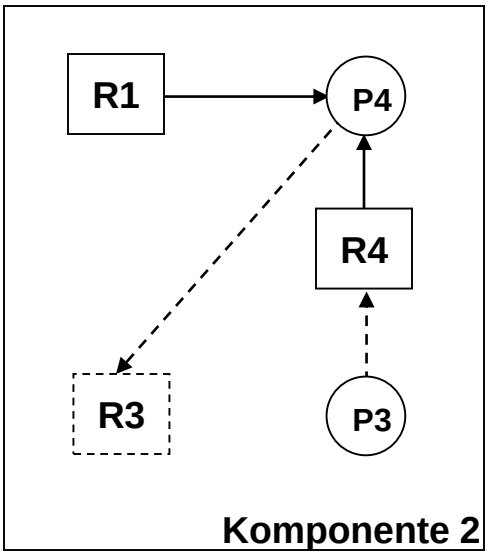
Belegung:
Ressource Rj wurde von Prozess Pi belegt.



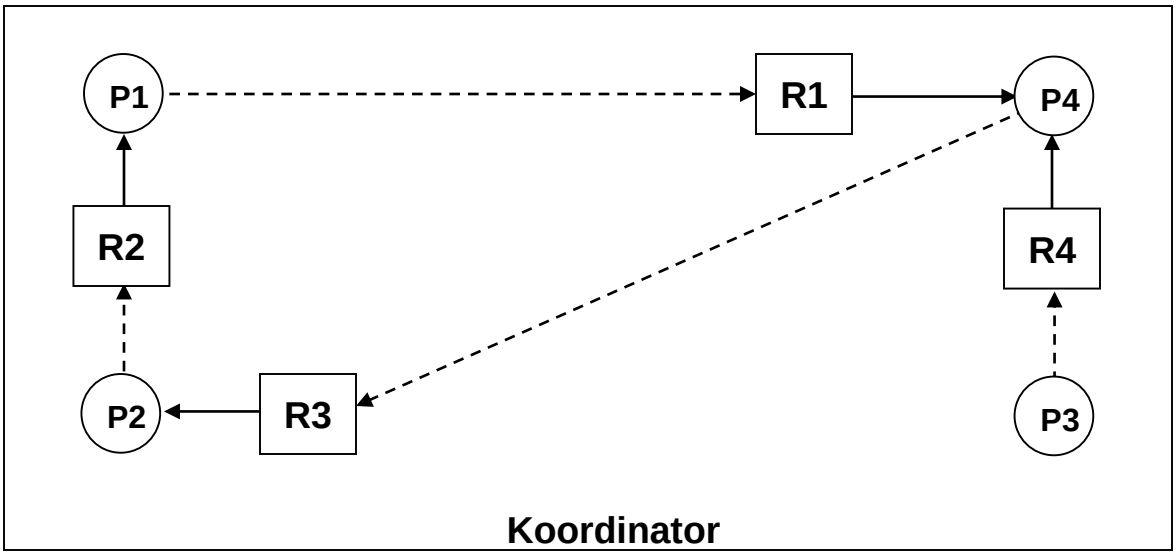
Anforderung:
Prozess Pi hat Ressource Rj angefordert.



lokaler Belegungs-
Anforderungs-Graph



lokaler Belegungs-
Anforderungs-Graph



globaler Belegungs-Anforderungs-Graph

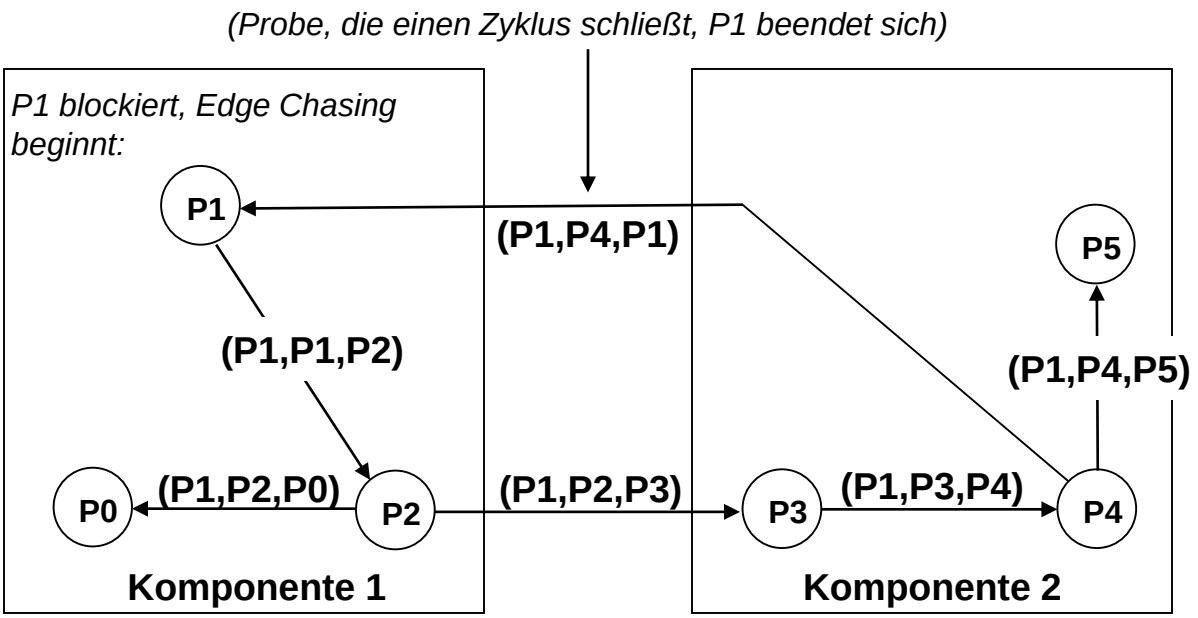
Deadlock-Auflösung:
Der Koordinator entscheidet, welcher Prozess abgebrochen wird.

zentrale Deadlockerkennung und –Auflösung durch Koordinator

Ein Prozess, der wegen der Anforderung einer Ressource blockiert, sendet eine *Probe* an den diese Ressource belegenden Prozess, mit dem Format: (blockierter Prozess, Sender, Empfänger).

Der Empfänger einer Probe sendet, *falls er ebenfalls wegen Anforderung von Ressourcen (egal welche) blockiert ist,* Probes an die diese Ressourcen haltenden Prozesse, wobei er nur Sender und Empfänger, aber nicht den blockierten Prozess ändert. Dieser Schritt setzt sich fort.

Erkennt ein Prozess in einer empfangenen Probe den blockierten Prozess wieder, den er in einer vorher gesendeten Probe eingetragen hat, stellt er einen Zyklus im Belegungs-Anforderungs-Graphen und damit einen Deadlock fest. Dieser Prozess beendet sich (Deadlock-Auflösung).



Probe: Management-Nachricht zur Deadlockerkennung

Probe-Format: (blockierter Prozess, Sender, Empfänger)

verteilte Deadlock-Erkennung und –Auflösung durch Edge Chasing

Deadlock-Vermeidung:

Aufgabe einer der notwendigen Bedingungen für Deadlocks (Exklusivität, Nicht-Unterbrechbarkeit, Fordern und Halten, zyklisches Warten:

bei allen Bedingungen nur für sehr spezielle Fälle realistisch

Beispiel: Aufgabe der Bedingung der Nicht-Unterbrechbarkeit:

// wait-die-Schema

if (TS(P_i) < TS(P_j))

then P_i waits ***// wait***

else P_i aborts and restarts with same TS later; ***// die***

oder

//wound-wait-Schema

if (TS(P_i) < TS(P_j))

then P_j aborts and restarts with same TS later ***// wound***

else P_i waits; ***// wait***

// TS(P_i) = Timestamp des Prozesses P_i (vergeben beim Start)
// TS(P_i) < TS(P_j) : P_i ist „älter“ als P_j, d.h. hat einen früheren Startzeitpunkt
// Ausgangssituation:
// Prozess P_j hat Ressource X bei Start des Schemas schon gesperrt,
// Prozess P_i versucht, X zu sperren.

Prozesse P1, ..., P5, Ressourcen R1,...,R4

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	0	0
P3	1	1	1	0
P4	1	1	0	1
P5	0	0	0	0

schon belegte Ressourcen

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	1	2
P3	3	1	0	0
P4	0	0	1	0
P5	2	1	1	0

noch benötigte Ressourcen

	R1	R2	R3	R4
Vorhandene Ressourcen	6	3	4	2
Belegte Ressourcen	5	3	2	2
Verfügbare Ressourcen	1	0	2	0

(Diese Ausgangssituation wurde aus Tanenbaum: Moderne Betriebssysteme adaptiert.)

Algorithmus:

Repeat (until alle Prozesse als beendet markiert oder ein unsicherer Systemzustand auftritt) {

Suche eine Zeile z in der Tabelle der noch benötigten Ressourcen mit z ≤ verfügbare Ressourcen (Vergleich zweier Zeilen (Arrays)).
Wenn eine solche Zeile nicht vorhanden ist: unsicherer Systemzustand ⇒ sicherer Deadlock in der Zukunft.
Teile dem Prozess, dem die Zeile z entspricht, die noch benötigten Ressourcen zu, lassen ihn laufen, beende ihn unter Rückgabe aller seiner belegten Ressourcen und markiere ihn als beendet.

}

Deadlock-Verhinderung durch Banker's Algorithmus

belegt

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	0	0
P3	1	1	1	0
P4	1	1	0	1
P5	0	0	0	0

benötgt

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	1	2
P3	3	1	0	0
P4	0	0	1	0
P5	2	1	1	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	5	3	2	2
Verfügbar	1	0	2	0

Frage: Kann P2 eine Ressource R3 belegen? Dann wäre das System im Zustand "Z":

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	**1!**	0
P3	1	1	1	0
P4	1	1	0	1
P5	0	0	0	0

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	**0!**	2
P3	3	1	0	0
P4	0	0	1	0
P5	2	1	1	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	5	3	**3!**	2
Verfügbar	1	0	**1!**	0

Ist dieser Zustand Z also sicher? Das wird im Folgenden überprüft ...
P4 belege 1mal R3, arbeite und gebe dann alle seine belegten Ressourcen zurück.

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	1	0
P3	1	1	1	0
P4	**1**	**1**	**1**	**1**
P5	0	0	0	0

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	0	2
P3	3	1	0	0
P4	**0**	**0**	**0**	**0**
P5	2	1	1	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	5	3	**4!**	2
Verfügbar	1	0	**0!**	0

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	1	0
P3	1	1	1	0
~~**P4**~~	**0**	**0**	**0**	**0**
P5	0	0	0	0

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	0	2
P3	3	1	0	0
~~**P4**~~	**0**	**0**	**0**	**0**
P5	2	1	1	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	4	2	**3!**	1
Verfügbar	2	1	**1!**	1

Banker's Algorithmus: Beispiel (Teil 1)

VS 6.19

P5 belege 2mal R1, je 1mal R2,R3, arbeite und gebe dann alle seine belegten Ressourcen zurück.

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	1	0
P3	1	1	1	0
~~P4~~	0	0	0	0
P5	2	1	1	0

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	0	2
P3	3	1	0	0
~~P4~~	0	0	0	0
P5	0	0	0	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	6	3	4	1
Verfügbar	0	0	0	1

	R1	R2	R3	R4
P1	3	0	1	1
P2	0	1	1	0
P3	1	1	1	0
~~P4~~	0	0	0	0
~~P5~~	0	0	0	0

	R1	R2	R3	R4
P1	1	1	0	0
P2	0	1	0	2
P3	3	1	0	0
~~P4~~	0	0	0	0
~~P5~~	0	0	0	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	4	2	3	1
Verfügbar	2	1	1	1

P1 belege 1mal R1, 1mal R2, arbeite und gebe dann alle seine belegten Ressourcen zurück.

	R1	R2	R3	R4
P1	4	1	1	1
P2	0	1	1	0
P3	1	1	1	0
~~P4~~	0	0	0	0
~~P5~~	0	0	0	0

	R1	R2	R3	R4
P1	0	0	0	0
P2	0	1	0	2
P3	3	1	0	0
~~P4~~	0	0	0	0
~~P5~~	0	0	0	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	5	3	3	1
Verfügbar	1	0	1	1

	R1	R2	R3	R4
~~P1~~	0	0	0	0
P2	0	1	1	0
P3	1	1	1	0
~~P4~~	0	0	0	0
~~P5~~	0	0	0	0

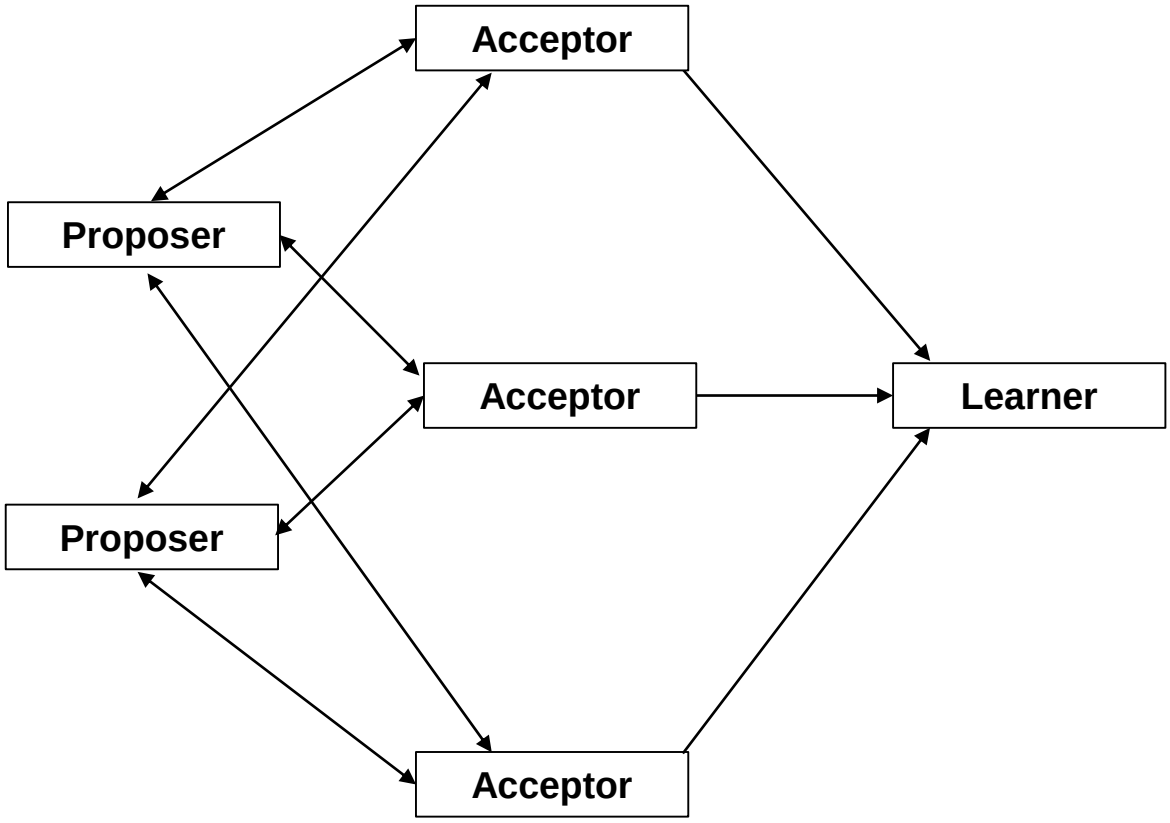
	R1	R2	R3	R4
~~P1~~	0	0	0	0
P2	0	1	0	2
P3	3	1	0	0
~~P4~~	0	0	0	0
~~P5~~	0	0	0	0

	R1	R2	R3	R4
Vorhanden	6	3	4	2
Belegt	1	2	2	0
Verfügbar	5	1	2	2

Jetzt kann P3 belegen: (3,1,0,0), arbeiten und dann alle Ressourcen zurückgeben. Dann kann P2 belegen: (0,1,0,2), arbeiten und dann alle Ressourcen zurückgeben. Dann sind alle Prozesse als beendet markiert, damit war Zustand Z sicher, und P2 konnte R3 belegen.

Banker's Algorithmus: Beispiel (Teil 2)

VS 6.20

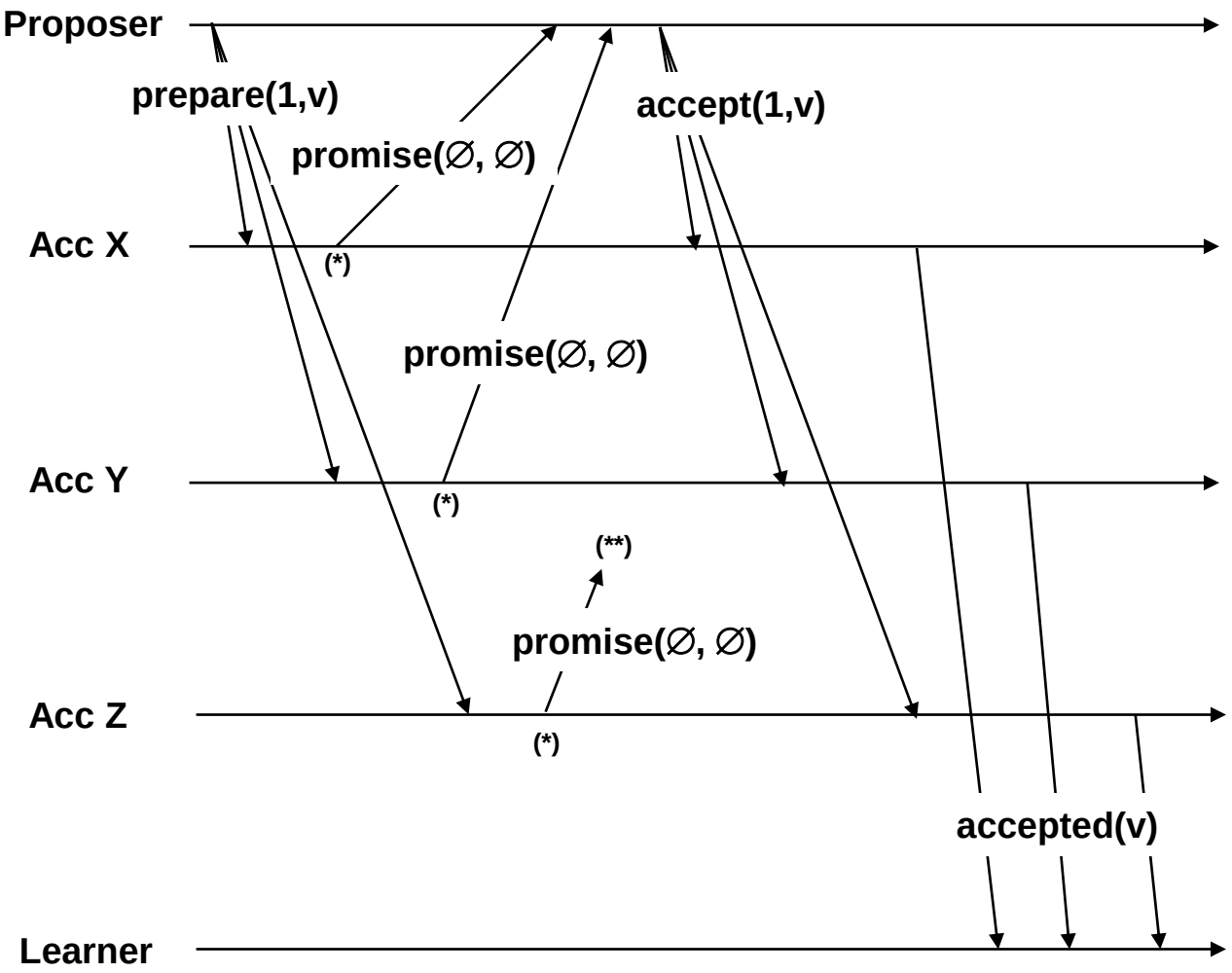


Nachricht	Bedeutung
prepare (n,v)	Proposer schlägt Acceptor mit Proposal Nummer n den Wert v vor.
promise (n,v)	Acceptor verspricht Proposer, niemals in der Zukunft einen Proposal mit niedrigerer Nummer als n zu akzeptieren.
accept (n,v)	Proposer fordert Akzeptor auf, den Wert v aus Proposal Nummer n als Wert zu akzeptieren.
accepted (v)	Akeptor gibt v als akzeptierten Wert an Learner weiter.

Paxos Architektur und Nachrichten

1 Proposer, mehrere Acceptors, 1 Learner

Proposal mit Nummer 1, Wert v



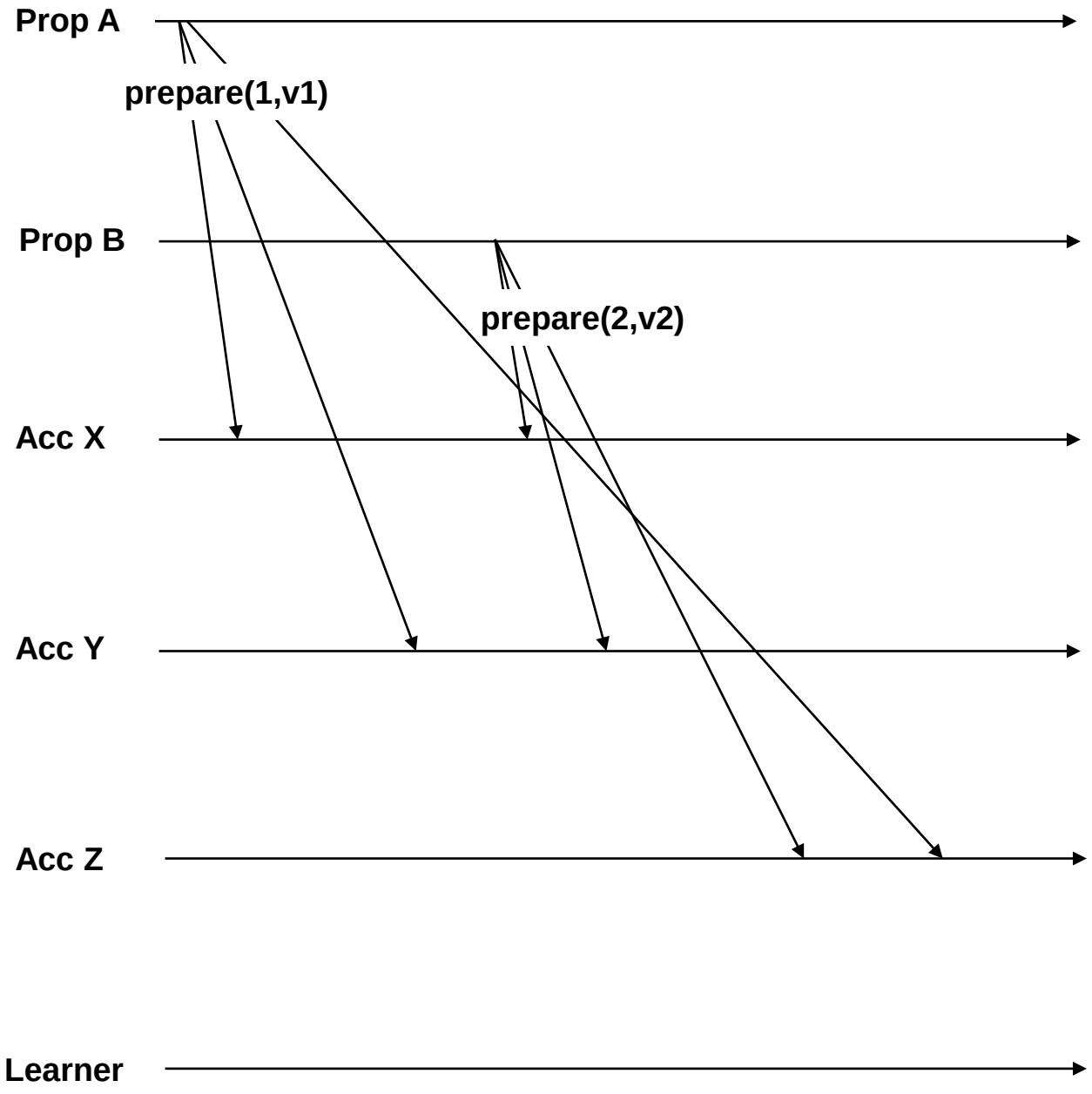
(*) Versprechen: keine Zustimmung zu Vorschlägen mit Nr. < 1

(**) nicht mehr relevant, da Mehrheit den Proposer erreicht hat

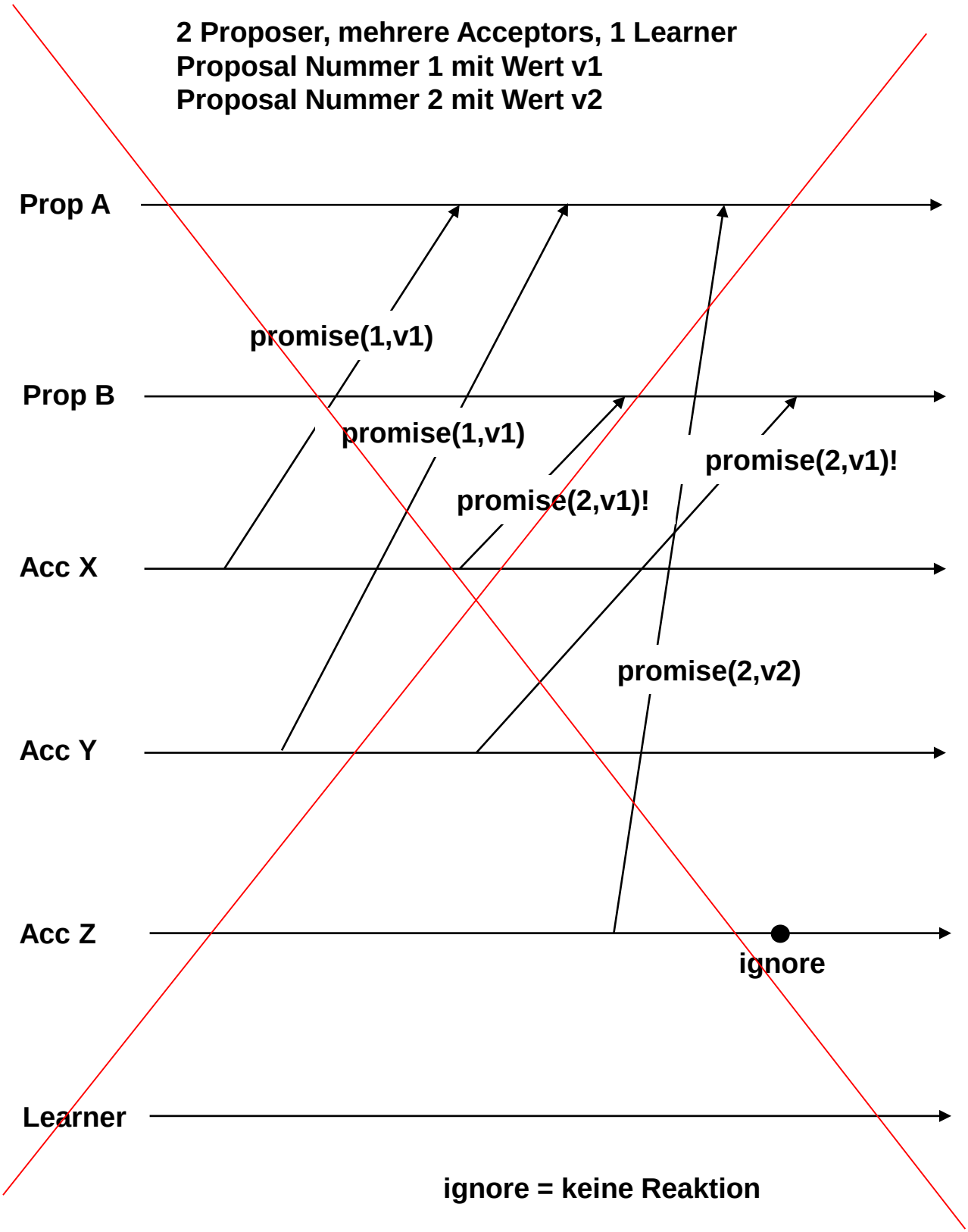
Zeit

Paxos Beispiel I

2 Proposer, mehrere Acceptors, 1 Learner
Proposal Nummer 1 mit Wert v1
Proposal Nummer 2 mit Wert v2

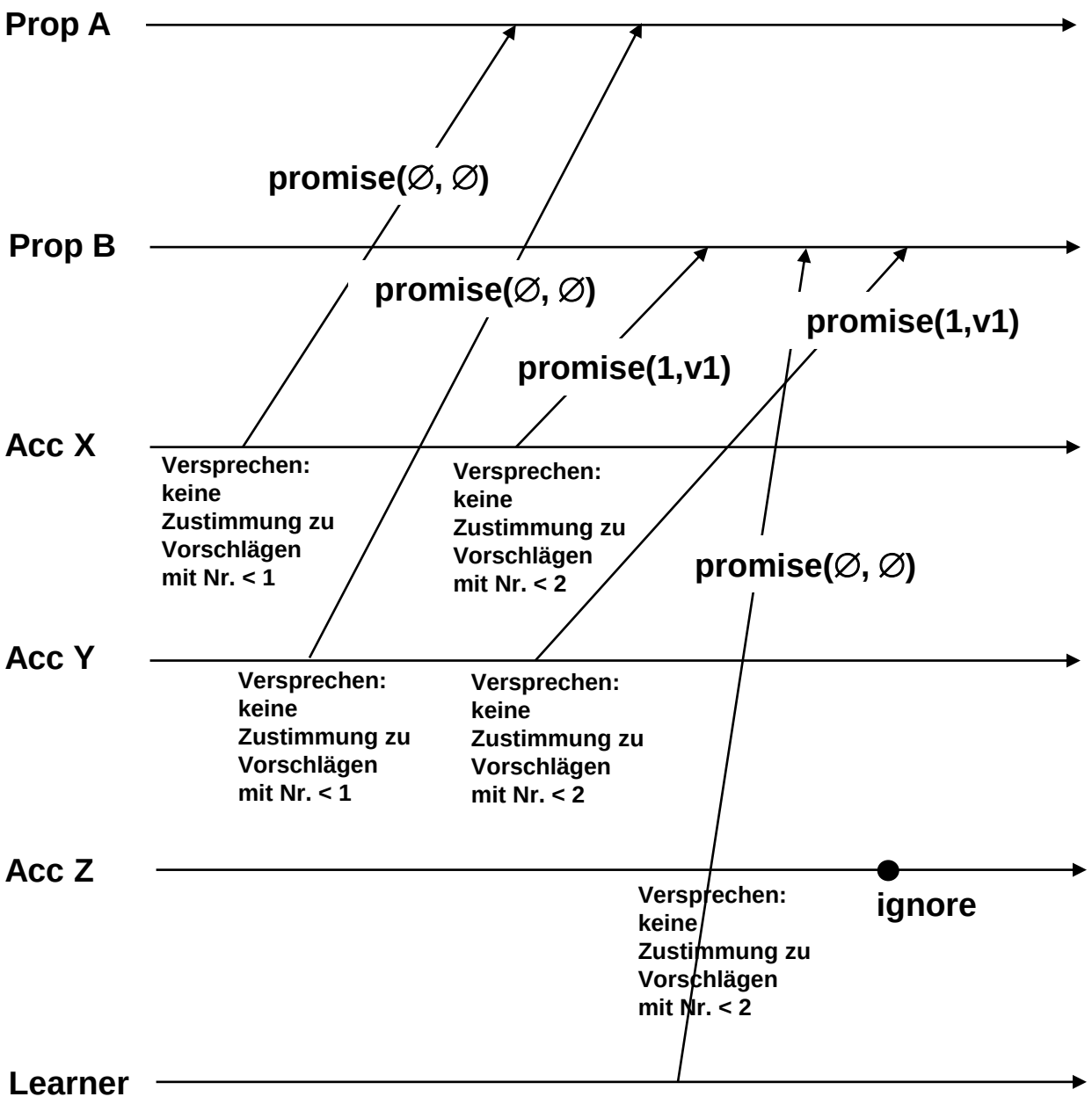


Paxos Beispiel II, Prepares



Paxos Beispiel II, Promises

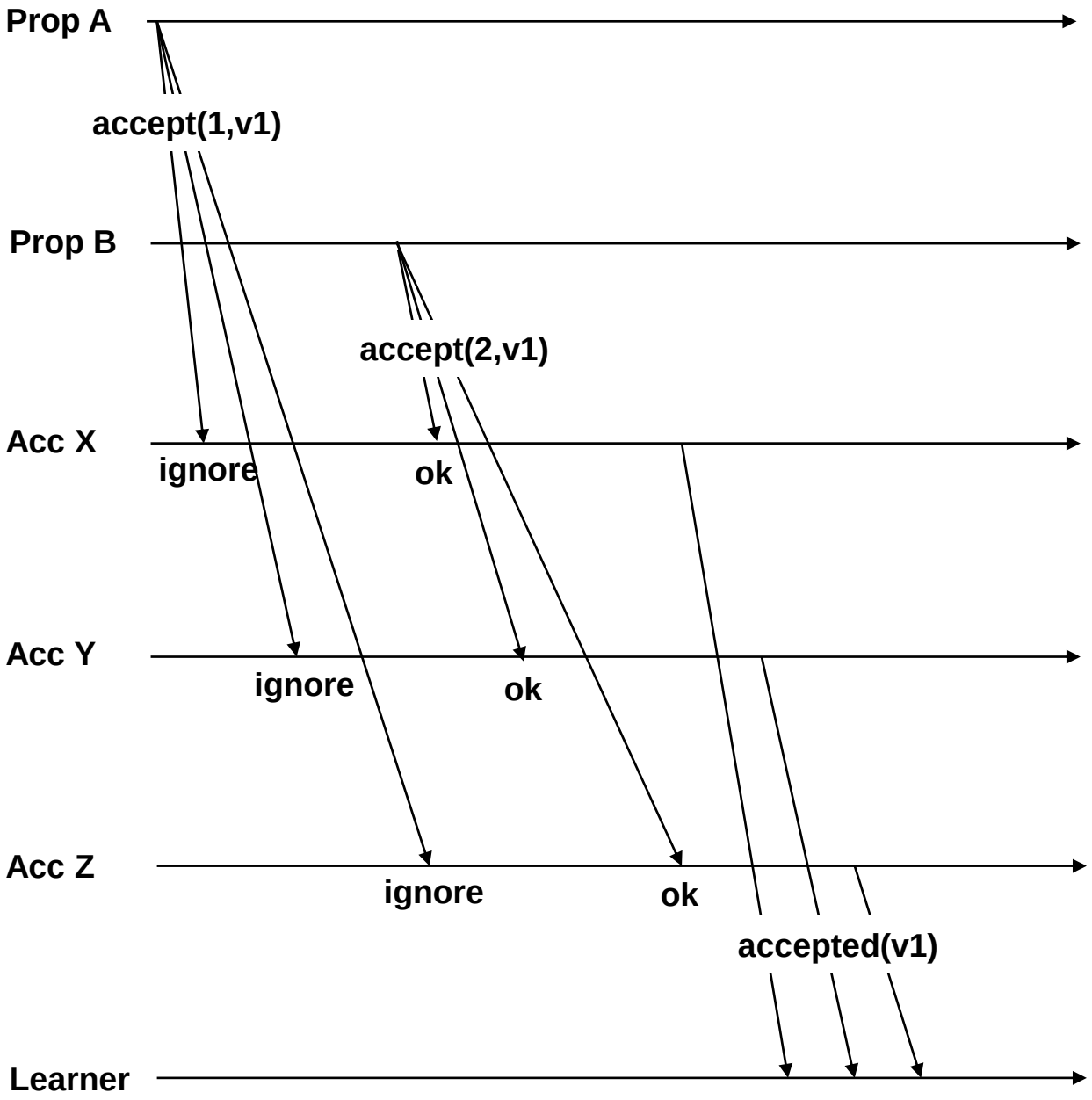
2 Proposer, mehrere Acceptors, 1 Learner
Proposal Nummer 1 mit Wert v1
Proposal Nummer 2 mit Wert v2



ignore = keine Reaktion, da schon Vorschlag mit höherer Nummer gesehen

Paxos Beispiel II, Promises

2 Proposer, mehrere Acceptors, 1 Learner
Proposal Nummer 1 mit Wert v1
Proposal Nummer 2 mit Wert v2



ignore: es wurde versprochen,
keine Proposals mehr mit Nummer < 2 zu akzeptieren.

Paxos Beispiel II, Accepts, Accepted's