

Consensus

Κατανεμημένα Συστήματα
2016-2017

<http://www.cslab.ece.ntua.gr/courses/distrib>

Στόχος του consensus

- Επιτρέπει σε ομάδα διεργασιών να συμφωνήσουν σε ένα αποτέλεσμα
 - Όλες οι διεργασίες πρέπει να συμφωνήσουν στην ίδια τιμή
 - Η τιμή πρέπει να έχει υποβληθεί από τουλάχιστον μια διεργασία (ένας αλγόριθμος consensus δεν μπορεί απλώς να επινοήσει μια τιμή)

Έχουμε δει ήδη...

- Mutual exclusion
 - Συμφωνία για το ποιος έχει πρόσβαση σε κάποιον πόρο
- Αλγόριθμοι εκλογών
 - Συμφωνία για το ποιος αναλαμβάνει κάποιον ξεχωριστό ρόλο
- Άλλες χρήσεις consensus:
 - Διαχείριση group membership
 - Συγχρονισμός replicas
 - Commit για κατανεμημένα transaction
- Γενικό πρόβλημα consensus:
 - *Πώς ερχόμαστε σε ομόφωνη απόφαση για μια συγκεκριμένη τιμή;*

Ορισμός του προβλήματος

- N διεργασίες
- Κάθε διεργασία p έχει
 - Μεταβλητή εισόδου x_p : αρχικά 0 ή 1
 - Μεταβλητή εξόδου y_p : αρχικά b ($b=undecided$) – μπορεί να αλλάξει μια φορά μόνο
- Το πρόβλημα consensus: Ορισμός πρωτοκόλλου έτσι ώστε
 - Όλες οι σωστές διεργασίες να θέσουν τη μεταβλητή εξόδου τους 0 ή
 - Όλες οι σωστές διεργασίες να θέσουν τη μεταβλητή εξόδου τους 1
 - Υπάρχει τουλάχιστον μια αρχική κατάσταση που οδηγεί σε καθένα από τα παραπάνω αποτελέσματα

Μοντέλο συστήματος

- Οι διεργασίες αποτυγχάνουν μόνο με *crash-stop*
- Σύγχρονο σύστημα: Όριο σε
 - Καθυστερήσεις μηνυμάτων
 - Μέγιστος χρόνος για κάθε βήμα επεξεργασίας
- Ασύγχρονο σύστημα: δεν υπάρχουν τέτοια όρια
 - Π.χ. το Internet

Σύγχρονο σύστημα

- Κάθε διεργασία ξεκινά με αρχική τιμή εισόδου 0 ή 1
- Κάθε διεργασία κρατά το ιστορικό των τιμών που έχει λάβει
- Το πρωτόκολλο τρέχει σε γύρους
- Σε κάθε γύρο, όλοι στέλνουν με **multicast** το ιστορικό των τιμών
- Αφού τελειώσουν όλοι οι γύροι, διαλέγουμε την ελάχιστη τιμή

Σύγχρονο σύστημα

- Για κάθε σύστημα που αποτυγχάνουν το πολύ f διεργασίες, ο αλγόριθμος συνεχίζει σε $f+1$ γύρους (με timeout), χρησιμοποιώντας το βασικό multicast
- $Values^r_i$: Το σύνολο των προτεινόμενων τιμών που είναι γνωστές στην διεργασία P_i στην αρχή του γύρου r
- Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i = x_p\}$
for round $r = 1$ to $f+1$ do
 multicast ($Values^r_i$)
 $Values^{r+1}_i \leftarrow Values^r_i$
 for each V_j received
 $Values^{r+1}_i = Values^{r+1}_i \cup V_j$
 end
end
 $y_p = \text{minimum}(Values^{f+1}_i)$

Γιατί δουλεύει;

- Ας υποθέσουμε ότι 2 σωστές διεργασίες p_i και p_j διαφέρουν στο τελικό σύνολο τιμών \rightarrow proof by contradiction
- Ας υποθέσουμε ότι η p_i έχει μια τιμή v που δεν έχει η p_j
- Άρα η p_j δεν πρέπει να έχει λάβει την v σε κανένα γύρο
 - \rightarrow Στον τελευταίο γύρο, κάποια τρίτη διεργασία p_k έστειλε v στην p_i και πέθανε πριν τη στείλει στην p_j .
 - \rightarrow Οποιαδήποτε διεργασία που έστειλε v στον προτελευταίο γύρο πρέπει να πέθανε, αλλιώς θα είχαν λάβει και οι δύο p_k και p_j την τιμή v
 - \rightarrow Προχωρώντας έτσι, σε καθέναν από τους προηγούμενους γύρους θα πρέπει να πέθαινε τουλάχιστον μια διεργασία
 - \rightarrow Αλλά υποθέσαμε το πολύ f σφάλματα και $f+1$ γύρους \implies contradiction.

Ασύγχρονο σύστημα

- Τα μηνύματα μπορούν να έχουν αυθαίρετη καθυστέρηση και οι επεξεργασίες να είναι αυθαίρετα αργές
- **Είναι αδύνατον να εγγυηθούμε ότι θα πετύχουμε consensus**
 - Ακόμα και μια αποτυχία είναι ικανή να αποτρέψει συμφωνία
 - Μια αργή διεργασία δε διακρίνεται από μια διεργασία που έχει πεθάνει
- Η αδυναμία ισχύει για οποιοδήποτε πρωτόκολλο consensus
- Αποδείχτηκε από Fischer, Lynch και Patterson το 1983 (FLP)
 - Λύθηκε διαφωνία που υπήρχε για μια δεκαετία

Είμαστε καταδικασμένοι;

- Τα ασύγχρονα συστήματα δεν μπορούν να εγγυηθούν ότι θα έρθουν σε συμφωνία ακόμα και με αποτυχία μιας μόνο διεργασίας
- Κλειδί: “εγγύηση”
 - Δε σημαίνει ότι οι διεργασίες δεν μπορούν να έρθουν σε συμφωνία αν μια από αυτές αποτύχει
 - Επιτρέπει συμφωνία με κάποια πιθανότητα μεγαλύτερη από 0
 - Στην πράξη πολλά συστήματα επιτυγχάνουν consensus
- Πώς το ξεπερνάμε;

Τεχνικές για να ξεπεράσουμε το αδύνατο

- Τεχνική 1: **κρύβουμε τις αποτυχίες** (crash-stop)
 - Χρήση persistent storage και τοπικών checkpoints
 - Μετά από αποτυχία, επανεκκίνηση της διεργασίας και ανάνηψη από το τελευταίο checkpoint
 - Μπορεί να εισάγει αυθαίρετες καθυστερήσεις
- Τεχνική 2: **χρησιμοποιούμε ανιχνευτές λαθών**
 - Π.χ., αν μια διεργασία είναι αργή, θεωρούμε ότι έχει αποτύχει
 - Τότε τη σκοτώνουμε πραγματικά ή αγνοούμε τα μηνύματά της από εκείνο το σημείο και μετά (fail-silent)
 - Αυτό μετατρέπει το ασύγχρονο σε σύγχρονο σύστημα
 - Οι ανιχνευτές λαθών δεν είναι 100% ακριβείς και απαιτούν μεγάλο timeout για να έχουν μια λογική συμπεριφορά

Σκοπός ενός αλγορίθμου consensus

- Ανοχή σε σφάλματα
 - Δεν μπλοκάρει όταν η πλειονότητα των διεργασιών δουλεύουν
- Συμφωνία σε ένα αποτέλεσμα ανάμεσα σε ομάδα διεργασιών ακόμα κι αν:
 - Κάποιες διεργασίες πεθάνουν
 - Κάποια μηνύματα χαθούν ή έρθουν εκτός σειράς
 - Αν παραδοθούν, τα μηνύματα δεν αλλοιώνονται

Απαιτήσεις

- Ισχύς (validity)
 - Μπορούν να επιλεγούν μόνο τιμές που έχουν προταθεί
- Ομοιόμορφη συμφωνία
 - Δε γίνεται δύο κόμβοι να επιλέξουν διαφορετικές τιμές
- Ακεραιότητα (integrity)
 - Κάθε κόμβος μπορεί να επιλέξει μία μοναδική τιμή
- Τερματισμός
 - Κάθε κόμβος θα επιλέξει τελικά (eventually) μια τιμή

Paxos

- Ένας αλγόριθμος consensus
 - Ένας από τους πιο αποδοτικούς και κομψούς αλγορίθμους
 - Πασίγνωστος
- Τι γίνεται με το FLP (impossibility of consensus)?
 - Προφανώς δε λύνει το FLP
 - Βασίζεται σε ανιχνευτές σφαλμάτων για παράκαμψη

Η ιστορία

- Δημιουργήθηκε από τον Leslie Lamport
- Το abstract του paper
 - *“Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxos parliament’s protocol provides a new way of implementing the state-machine approach to the design of distributed systems.”*

Η ιστορία

- Ο κόσμος αρχικά νόμιζε ότι πρόκειται για αστείο
- Ο Lamport δημοσίευσε τελικά το paper 8 χρόνια μετά τη συγγραφή του, το 1998 (“The Part-Time Parliament”)
- Κανείς δεν το καταλάβαινε
- Ο Lamport έγραψε άλλο paper που εξηγεί το Paxos σε απλά Αγγλικά
 - Τίτλος: “Paxos Made Simple”
 - Abstract: “The Paxos algorithm, when presented in plain English, is very simple.”
- Ακόμα κι έτσι, ο αλγόριθμος είναι δυσνόητος

Τελικά όμως...

- Πολλά πραγματικά συστήματα υλοποιούν το Paxos
 - Google Chubby
 - MS Bing cluster management
 - AWS
 - Cassandra
- Amazon CTO Werner Vogels (σε blog post “Job Openings in My Group”)
 - *“What kind of things am I looking for in you?”*
 - *“You know your distributed systems theory: You know about logical time, snapshots, stability, message ordering, but also acid and multi-level transactions. You have heard about the FLP impossibility argument. You know why failure detectors can solve it (but you do not have to remember which one diamond-w was). **You have at least once tried to understand Paxos by reading the original paper.**”*

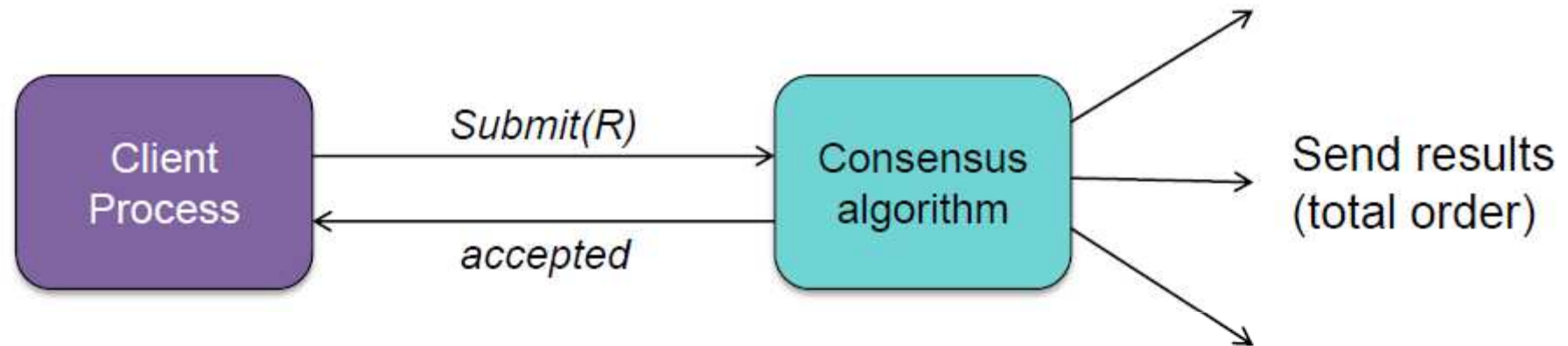
Υποθέσεις

- Το δίκτυο είναι ασύγχρονο με καθυστερήσεις στα μηνύματα
- Το δίκτυο μπορεί να χάσει μηνύματα ή να τα στείλει περισσότερες από μια φορές αλλά δεν μπορεί να τα αλλοιώσει
- Οι διεργασίες μπορεί να κρασάρουν (και να σταματήσουν)
- Οι διεργασίες έχουν *permanent storage* (δίσκο)
- Οι διεργασίες μπορούν να προτείνουν τιμές
- Ο στόχος είναι όλες οι διεργασίες να συμφωνήσουν σε μία από τις προτεινόμενες τιμές

Επιθυμητές ιδιότητες

- Safety
 - Μπορεί να επιλεγεί μόνο τιμή που έχει προταθεί
 - Επιλέγεται μια μόνο τιμή
 - Μια διεργασία μαθαίνει μια τιμή μόνο αν έχει επιλεγεί
- Liveness
 - Κάποια προτεινόμενη τιμή επιλέγεται τελικά (eventually)
 - Αν επιλεγεί μια τιμή, κάθε διεργασία τελικά θα τη μάθει

Η οπτική του χρήστη



- `while (submit_request(R) != ACCEPTED) ;`
- Το R θα μπορούσε να είναι ένα `key:value` ζεύγος μιας βάσης

Οι παίκτες του Paxos

- **Client:**
 - Στέλνει ένα αίτημα
- **Proposers:**
 - Λαμβάνουν αίτημα από τον client και τρέχουν το πρωτόκολλο
 - Leader: Εκλεγμένος coordinator ανάμεσα στους proposers (δεν είναι απαραίτητος, απλώς απλοποιεί τη διάταξη μηνυμάτων και διασφαλίζει ότι δεν υπάρχει διαφωνία)
- **Acceptors:**
 - Πολλαπλές διεργασίες που θυμούνται το state του πρωτοκόλλου
 - Quorum = πλειονότητα από acceptors
- **Learners:**
 - Όταν οι acceptors συμφωνήσουν, ο learner εκτελεί το αίτημα ή/και στέλνει απάντηση στον client

Διαφορετικοί ρόλοι μπορεί να
συνυπάρχουν στον ίδιο κόμβο

Τι κάνει το Paxos

- Κάθε πρόταση (αίτημα) έχει ένα ID.
 - (proposal #, value) == (N, V)
 - Ο αύξων αριθμός του proposal είναι καθολικά μοναδικός
- Τρεις φάσεις
 - Φάση prepare: ένας proposer learns previously-accepted proposals from the acceptors.
 - Φάση Propose: ένας proposer στέλνει proposal.
 - Φάση Learn: οι learners μαθαίνουν το αποτέλεσμα

Τι κάνει το Paxos

- Proposers
 - Πριν να προτείνει κάποιος proposer θα ρωτήσει τους acceptors αν έχει ήδη προταθεί κάποια άλλη τιμή
 - Αν ναι, ο proposer θα προτείνει την ίδια
 - Η συμπεριφορά είναι αλτρουιστική: Ο στόχος του κάθε proposer είναι η ομοφωνία, όχι να επιλεγεί η τιμή που προτείνει
- Acceptors
 - Ο στόχος είναι να επιλέξει το proposal με τον μεγαλύτερο αύξοντα αριθμό από όλους τους proposers
- Learners
 - Οι learners είναι παθητικοί και περιμένουν το αποτέλεσμα

1^η φάση

- Ένας proposer επιλέγει τον αύξοντα αριθμό N του proposal του και στέλνει *prepare request* στους acceptors.
 - “Hey, have you accepted any proposal yet?”
- Ένας acceptor πρέπει να απαντήσει:
 - Αν έχει ήδη κάνει accept σε προτάσεις, την πρόταση με τον μεγαλύτερο αύξοντα αριθμό
 - Μια υπόσχεση να μη δεχτεί πλέον κανένα proposal με αύξοντα αριθμό μικρότερο του N

2^η φάση

- Αν ένας proposer λάβει απάντηση από την πλειοψηφία, τότε στέλνει *accept request* για το proposal (N, V).
 - V: Η τιμή του proposal με τον μεγαλύτερο αύξοντα αριθμό N (από αυτές που επεστράφησαν στην 1^η φάση).
- Μετά τη λήψη του (N, V), οι acceptors είτε:
 - Το δέχονται
 - Το απορρίπτουν αν υπάρχει άλλο prepare request με N' μεγαλύτερο του N και έχει απαντήσει σε αυτό

3^η φάση

- Οι learners πρέπει να μάθουν ποια τιμή επελέγη
- Ένας τρόπος: Κάθε acceptor απαντάει σε όλους τους learners
 - ακριβό
- Άλλος τρόπος: εκλογή “distinguished learner”
 - Οι Acceptors απαντούν σε αυτόν
 - Αυτός ενημερώνει τους υπόλοιπους
 - Επιρρεπές σε σφάλματα
- Μίξη των δύο: σύνολο από distinguished learners

Πρόβλημα

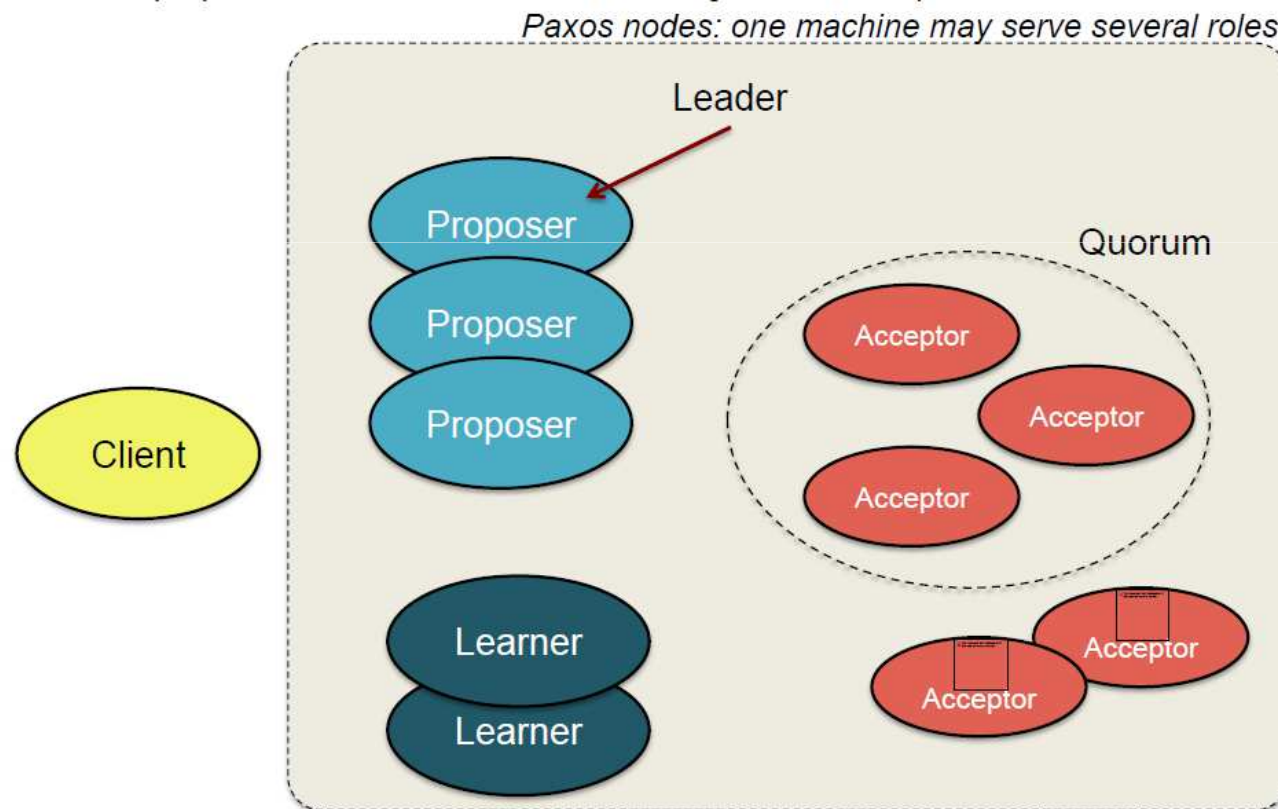
- *race condition για τα proposals.*
- Η P_0 ολοκληρώνει την φάση 1 με proposal number N_0
- Προτού η P_0 ξεκινήσει την φάση 2, η P_1 ξεκινά και ολοκληρώνει την φάση 1 με proposal number $N_1 > N_0$
- Η P_0 ολοκληρώνει την φάση 2, οι acceptors το απορρίπτουν
- Προτού η P_1 ξεκινήσει την φάση 2, η P_0 ξεκινά ξανά και ολοκληρώνει την φάση 1 με proposal number $N_2 > N_1$
- Η P_1 ολοκληρώνει την φάση 2, οι acceptors το απορρίπτουν
- ... (συνεχίζεται επ' άπειρον)

Λύση

- Λύση: εκλογή ενός διακεκριμένου proposer
- Αν ο proposer μπορεί να επικοινωνήσει επιτυχώς με την πλειονότητα των κόμβων, τότε το πρωτόκολλο εξασφαλίζει liveness.
 - δηλαδή, το Paxos έχει ανοχή σε $f < 1/2 * N$ σφάλματα

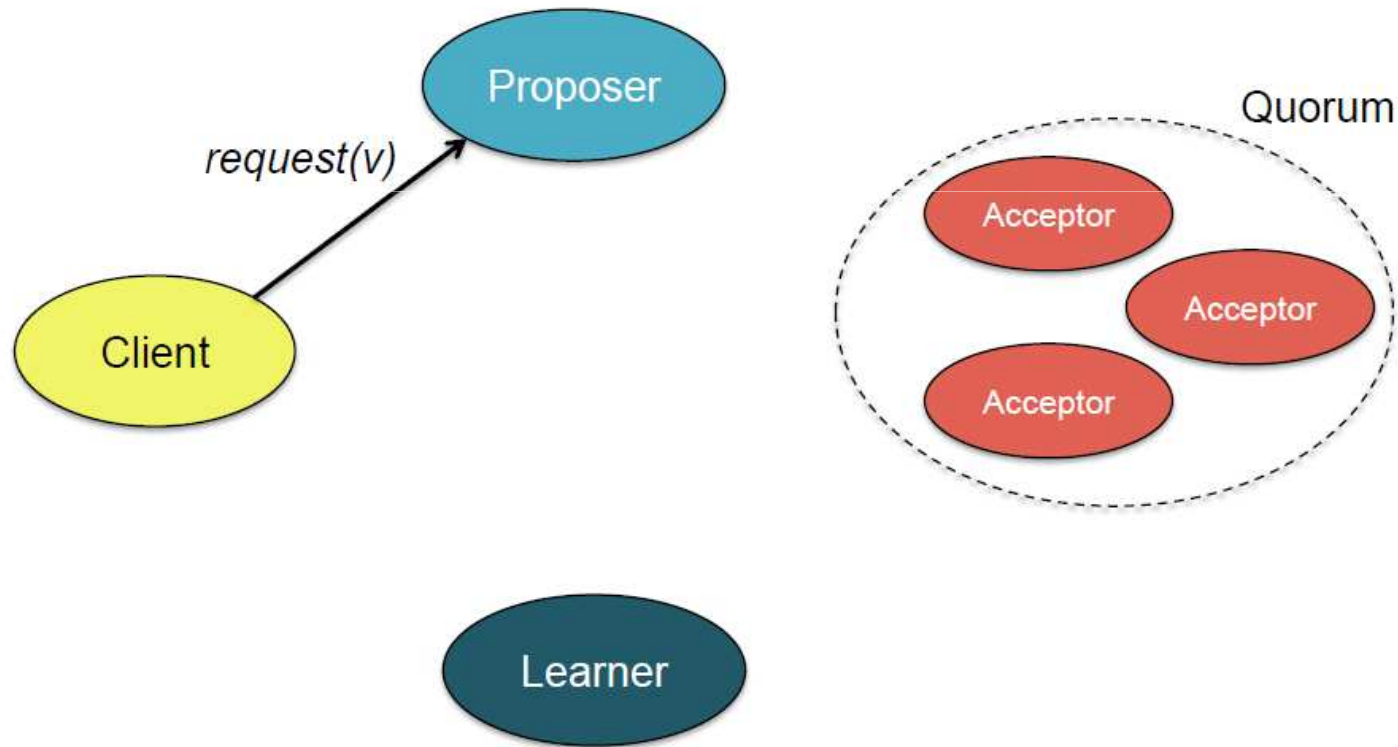
Ραχος εν δράση

- Ένας proposer εκλέγεται ως leader και δέχεται όλα τα requests



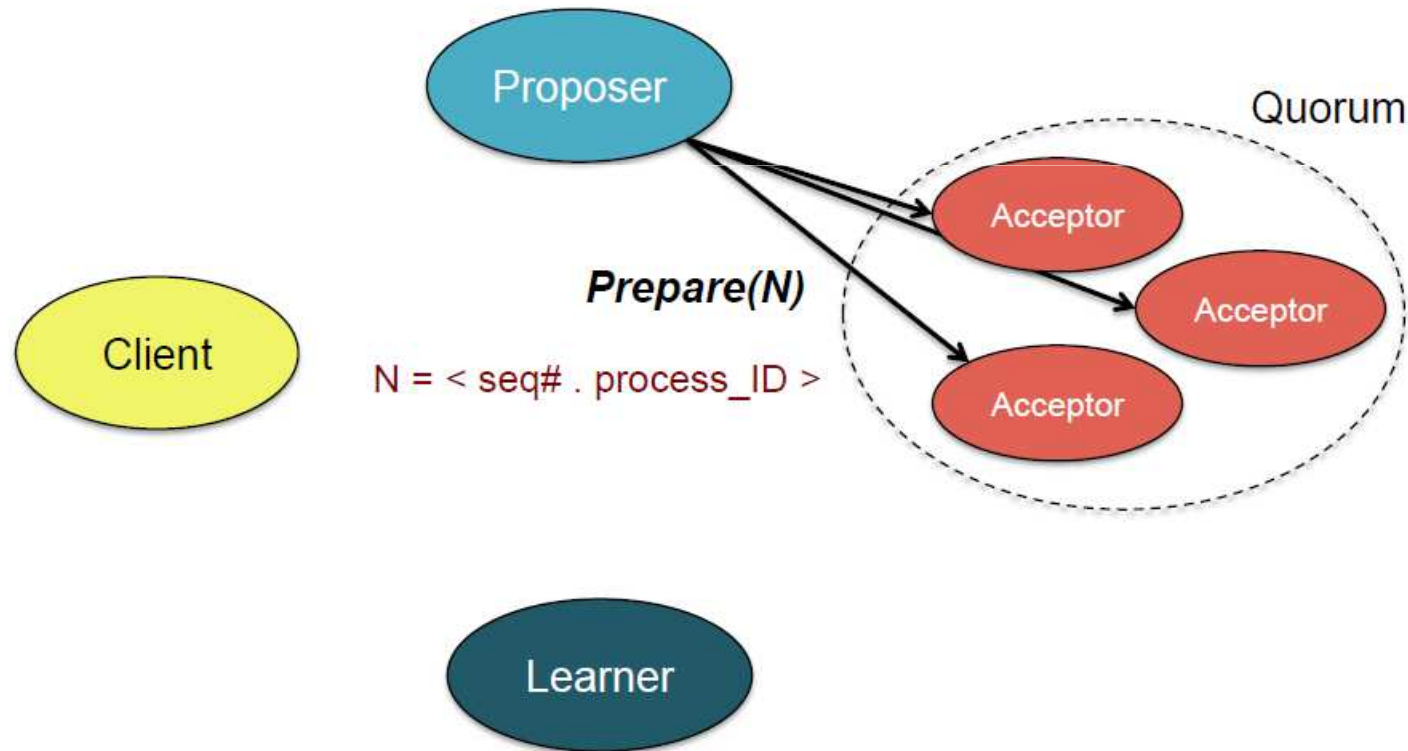
Phase 0

- Ο client στέλνει request στον proposer



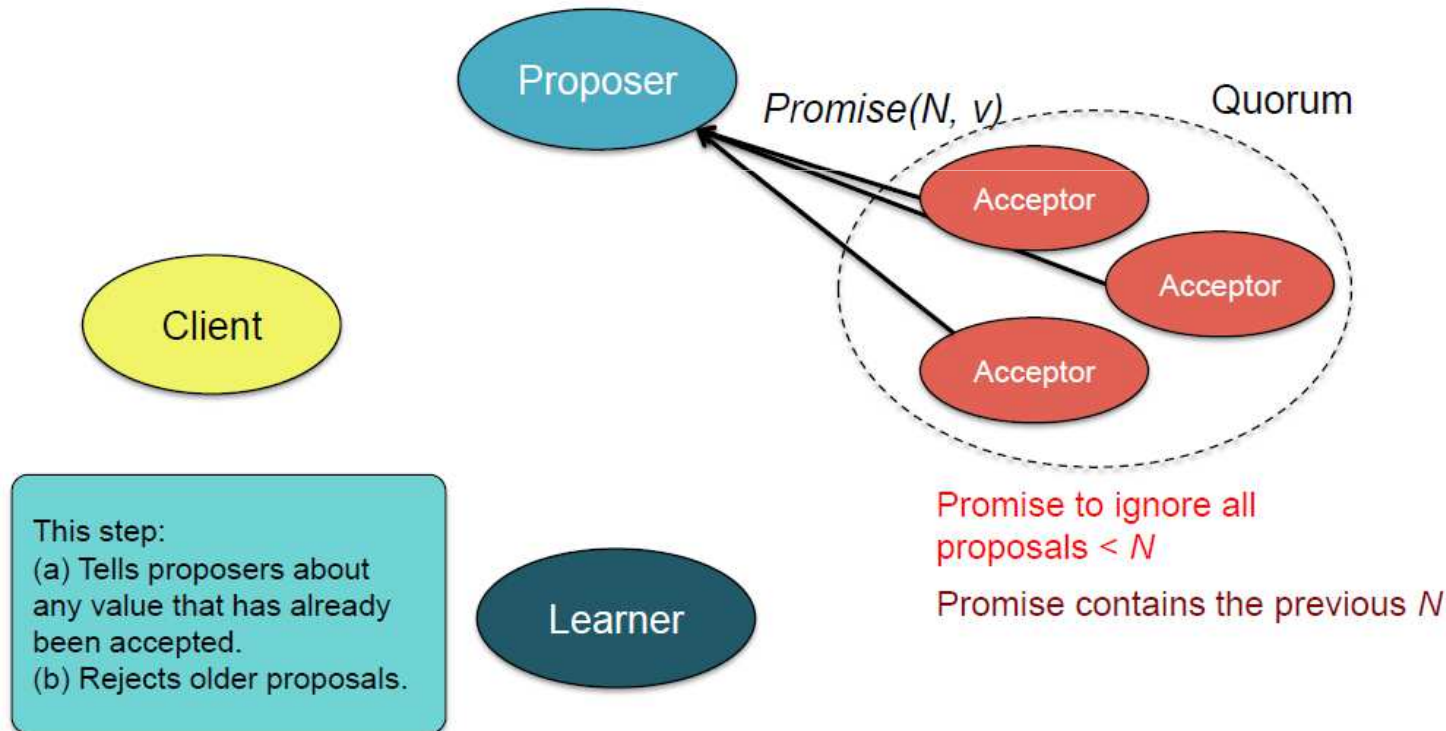
Phase 1a - PREPARE

- **Proposer:** δημιουργεί ένα *proposal #N* (*N λειτουργεί σαν χρονοσφραγίδα Lamport*), όπου το *N* είναι μεγαλύτερο από οποιοδήποτε προηγούμενο proposal number που χρησιμοποιήθηκε από τον proposer
- **Αποστολή σε quorum από Acceptors** (όσους μπορεί να προσπελάσει – σίγουρα πλειονότητα)



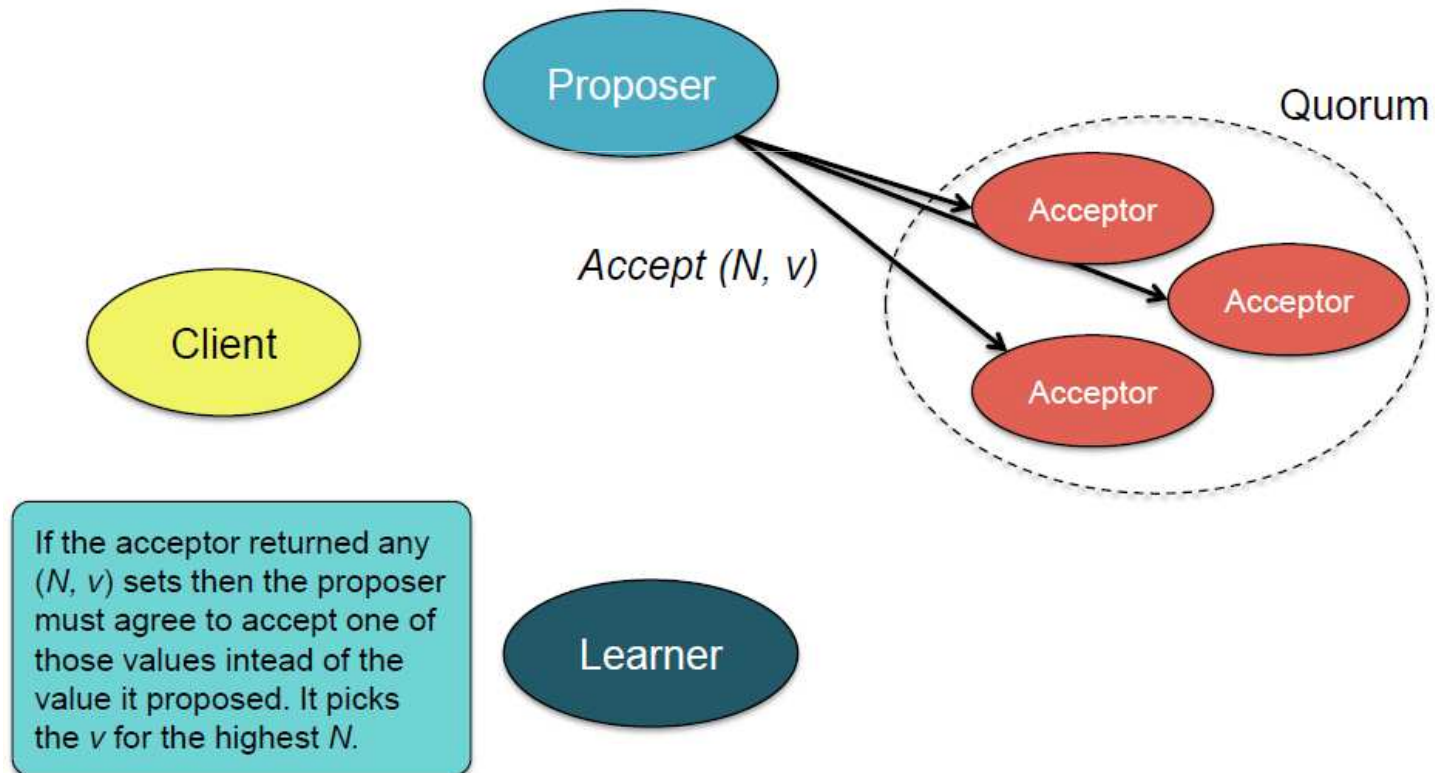
Phase 1a - PROMISE

- **Acceptor:** αν το ID του proposal > οποιοδήποτε προηγούμενο proposal
- Υπόσχεται να αγνοήσει όλα τα αιτήματα με $ID < N$ και απαντάει με πληροφορίες για το πιο πρόσφατο proposal που έχει αποδεχτεί
- Αλλιώς το απορρίπτει



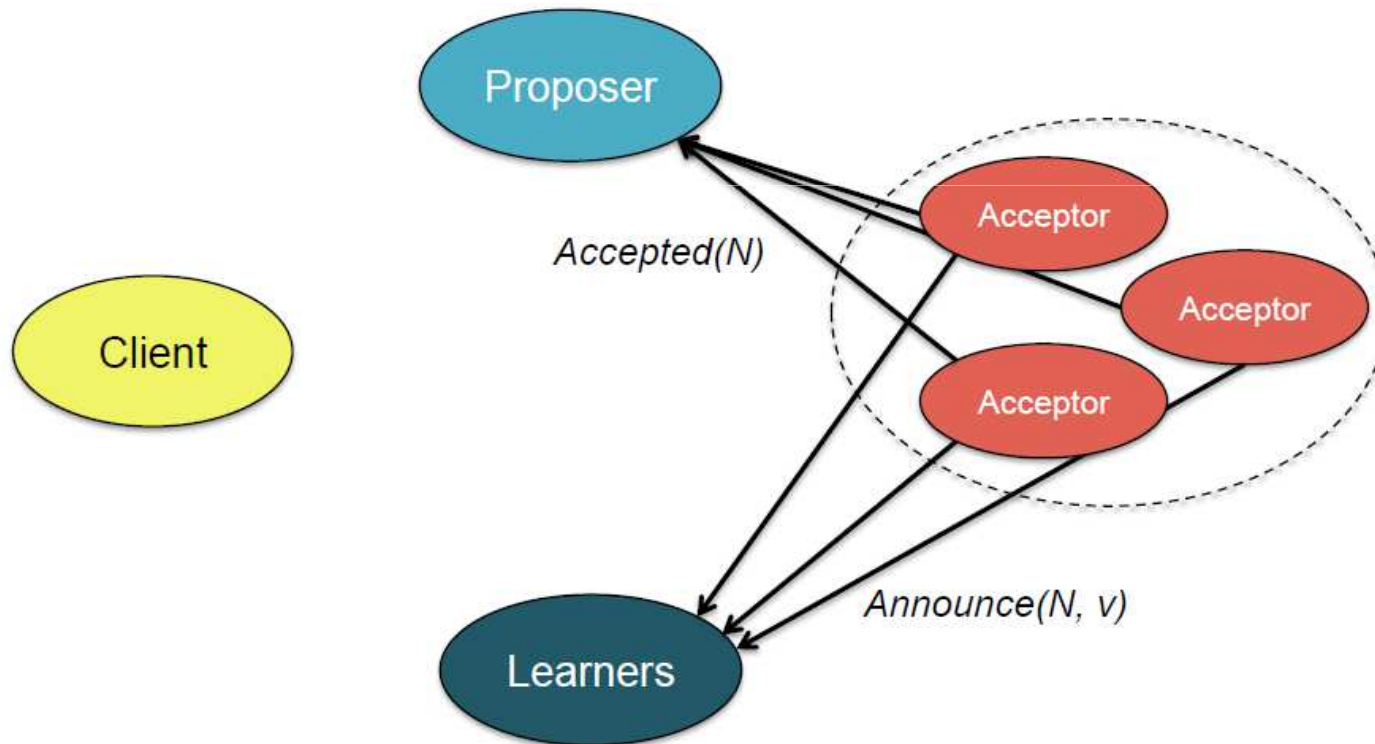
Phase 2a - ACCEPT

- **Proposer:** αν ο proposer λάβει υπόσχεση από την πλειονότητα
- Επισυνάπτει την τιμή v στο *proposal*
- Στέλνει *accept* στο *quorum* με την επιλεγμένη τιμή
- Αν η υπόσχεση ήταν για κάποιο άλλο $\{N, v\}$, ο proposer ΠΡΕΠΕΙ να το αποδεχτεί



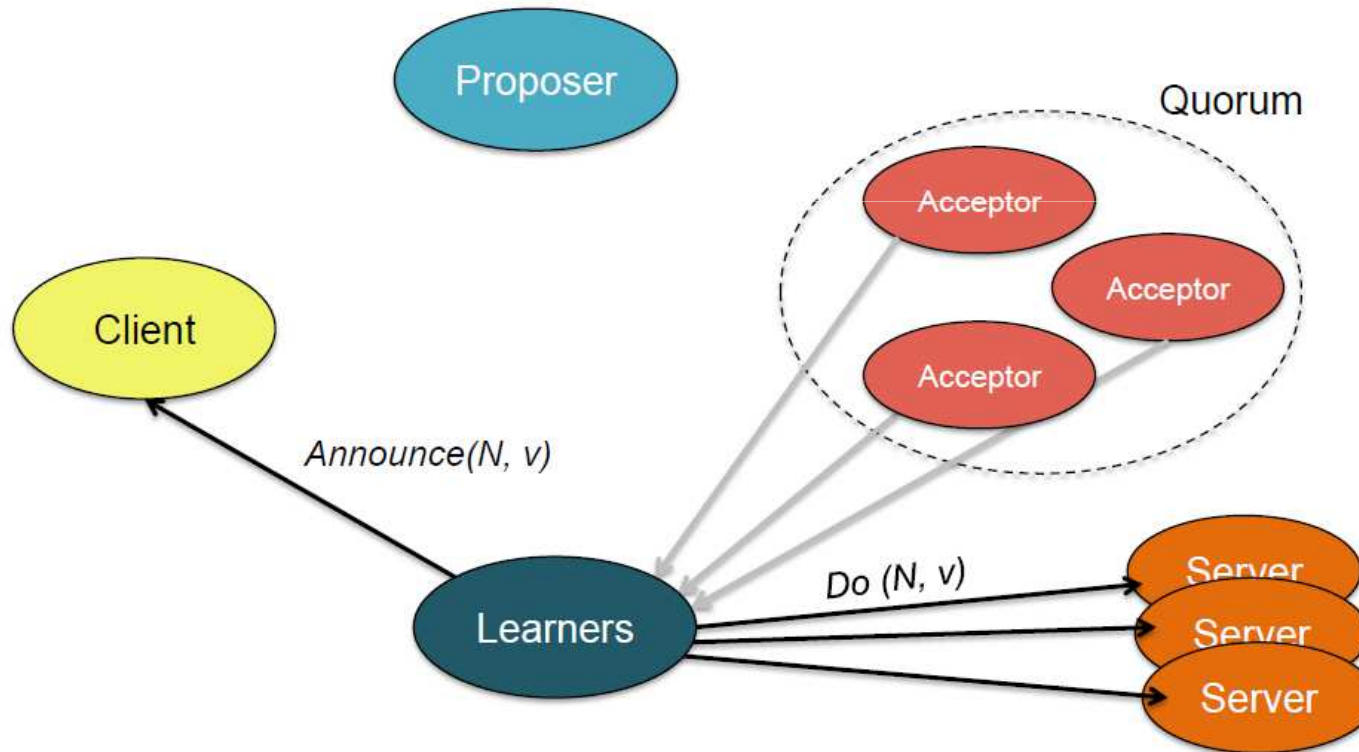
Phase 2b - ANNOUNCE

- **Acceptor:** Αν η υπόσχεση ισχύει ακόμα, ανακοινώνει την τιμή v
- Στέλνει μήνυμα *Accepted* στον *Proposer*
- Αλλιώς αγνοεί το μήνυμα (ή στέλνει *NACK*)

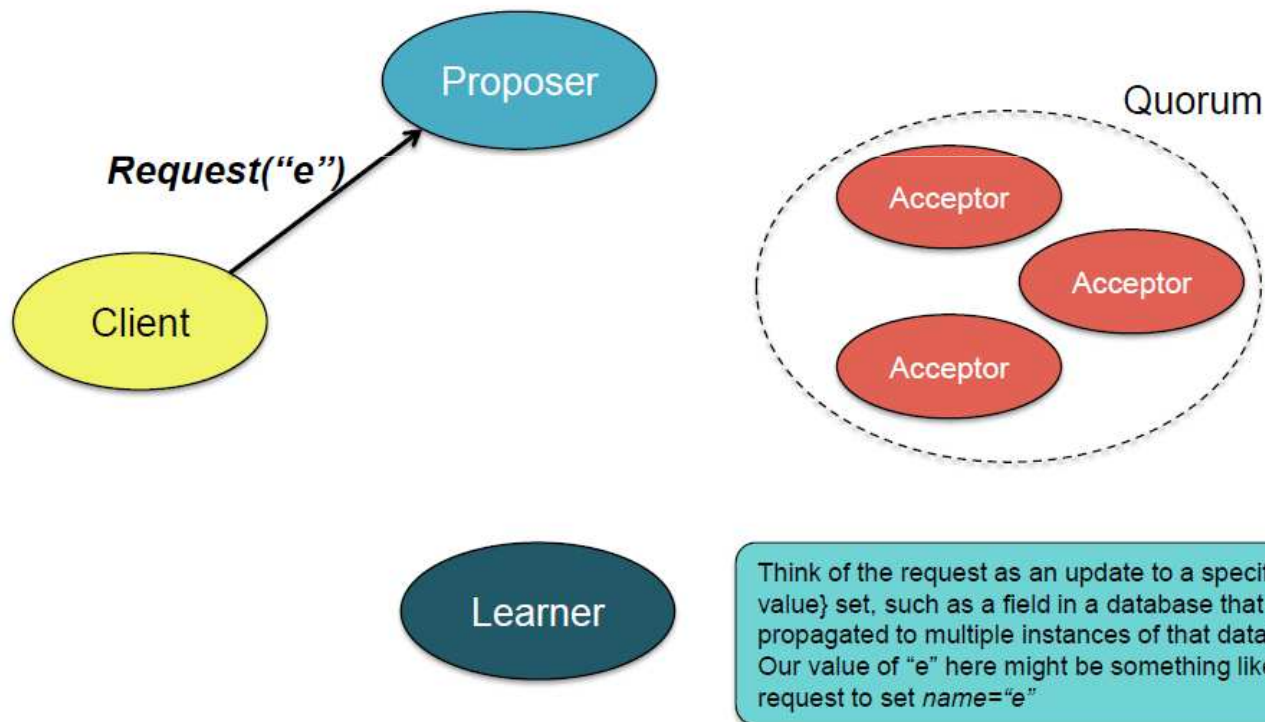


Phase 3

- **Learner:** Απαντά στον client και κάνει τη δουλειά που πρέπει

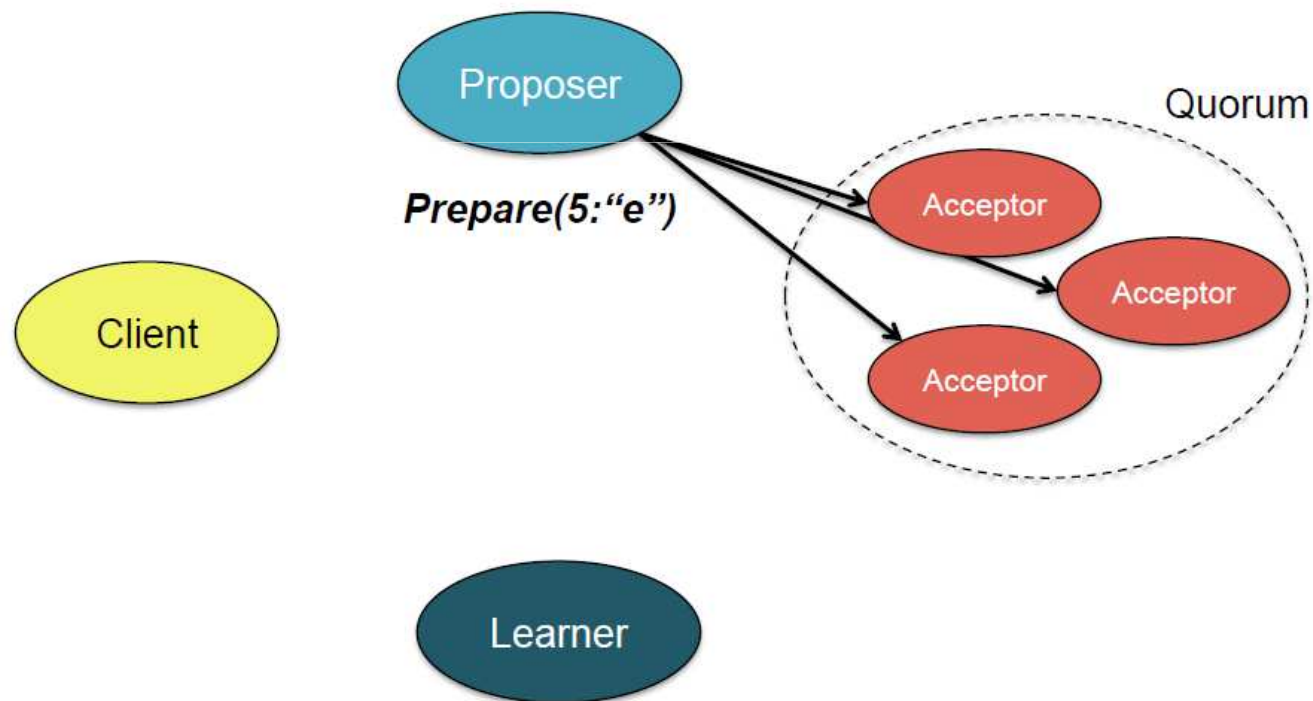


Παράδειγμα 1: phase 0



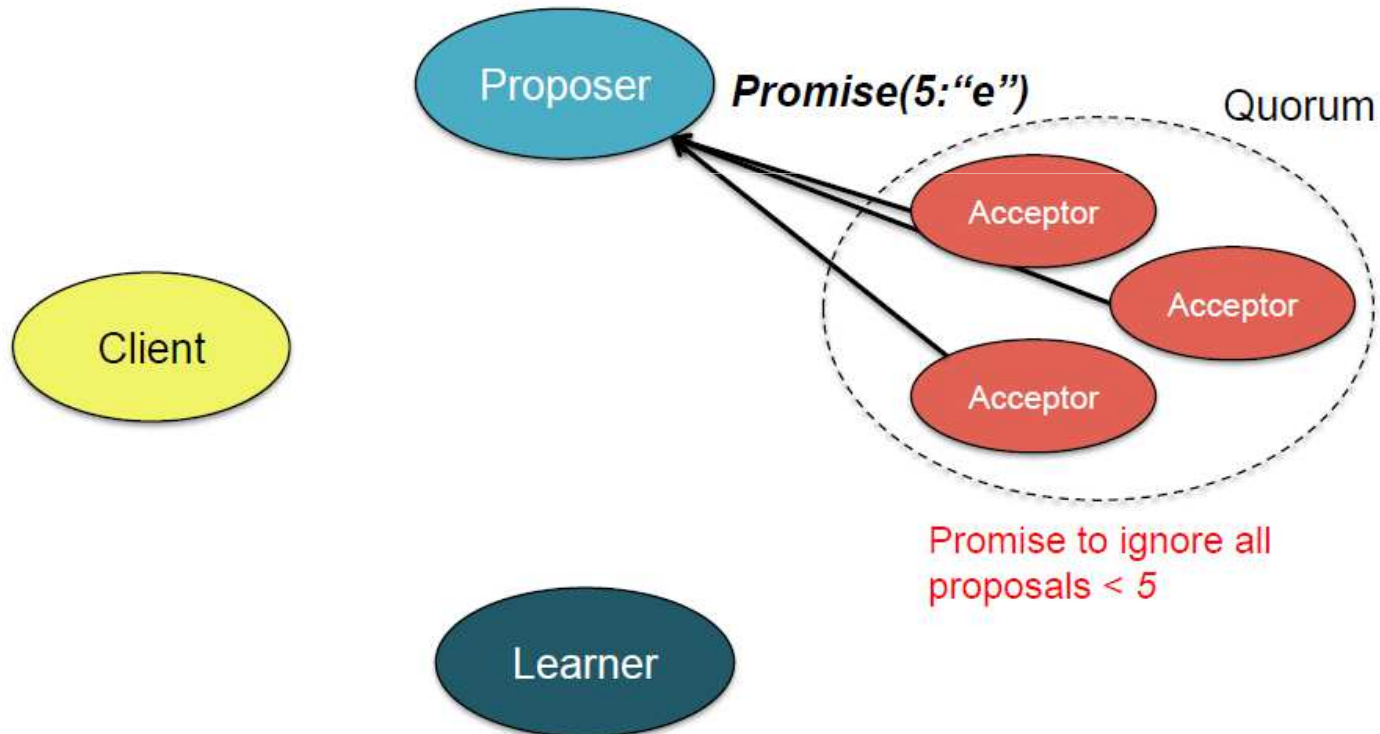
Phase 1a

- **Proposer:** διαλέγει id 5



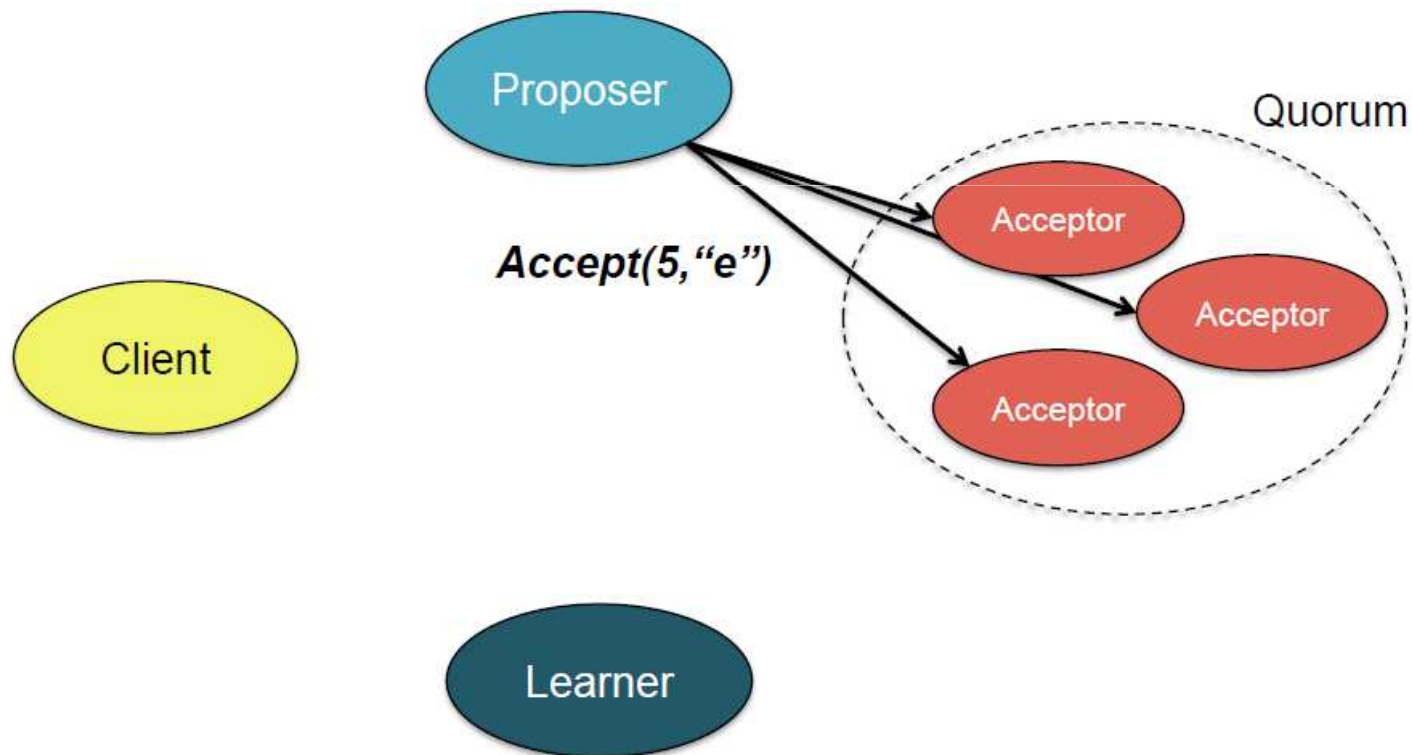
Phase 1b

- **Acceptor:** Αν το 5 είναι το μεγαλύτερο id που έχει δει, στέλνει υπόσχεση να μη δεχτεί μικρότερα ids



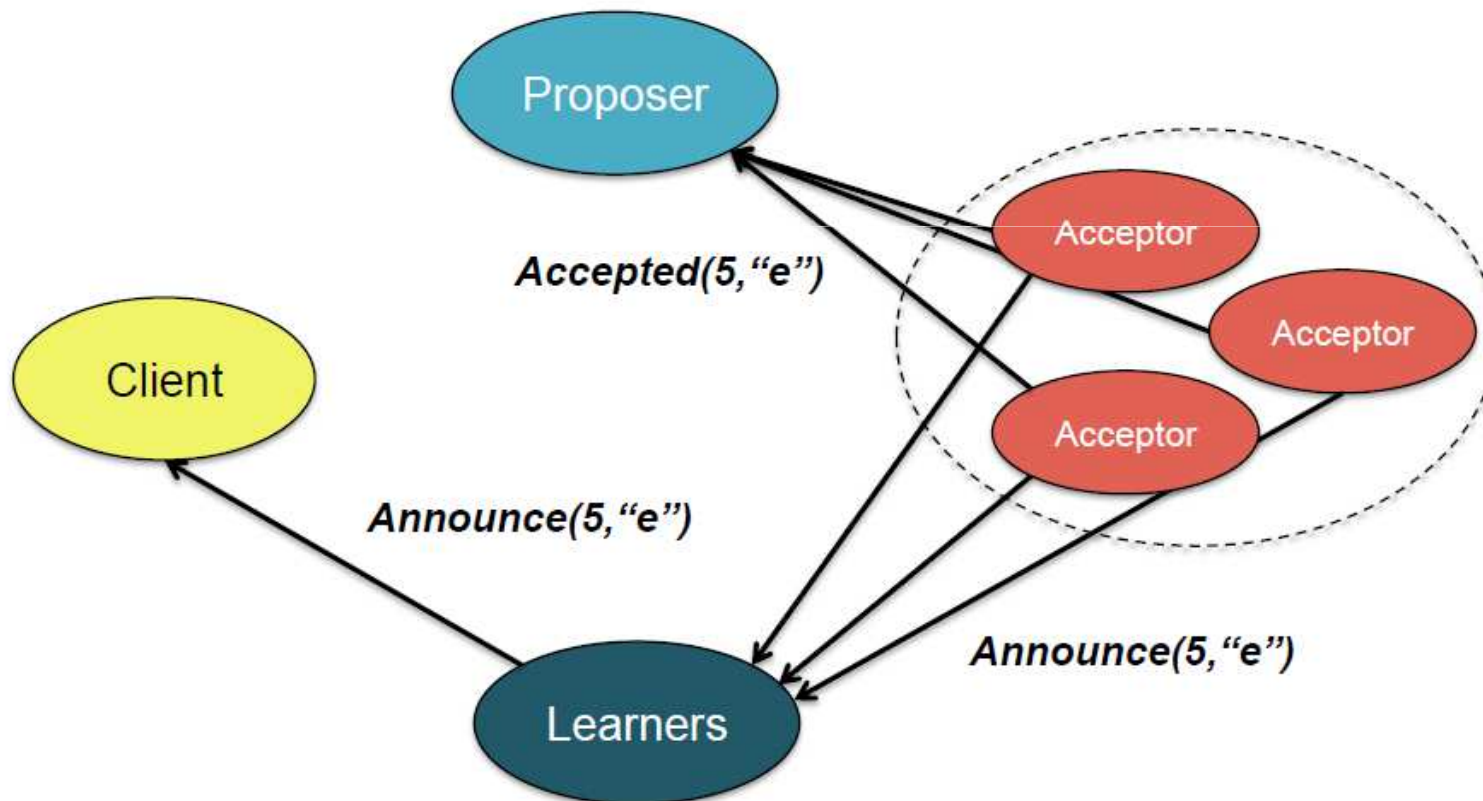
Phase 2a

- **Proposer:** λαμβάνει υπόσχεση από την πλειονότητα των acceptors
- Πρέπει να δεχτεί αυτό το $\langle id, value \rangle$

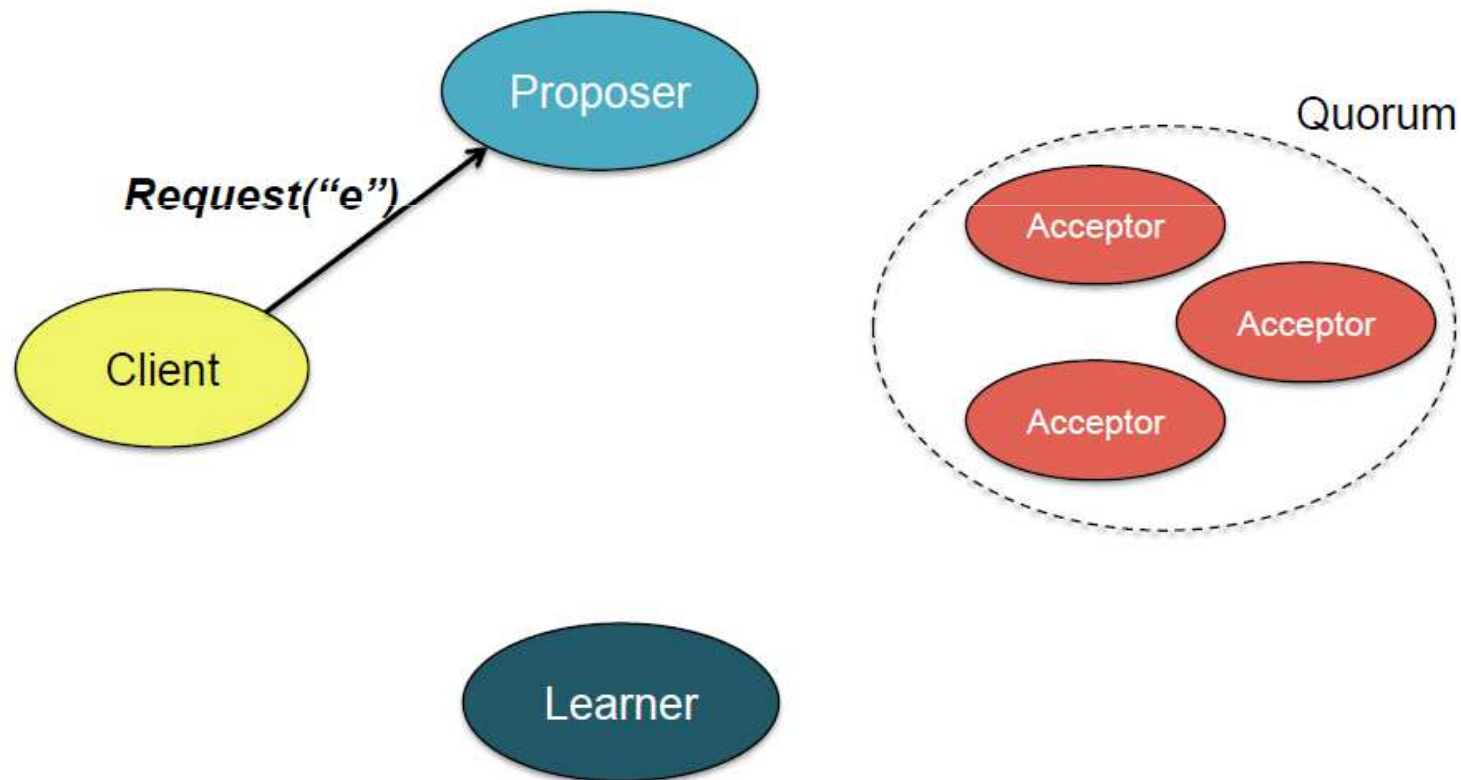


Phase 2b

- **Acceptor:** ανακοινώνουν απόφαση



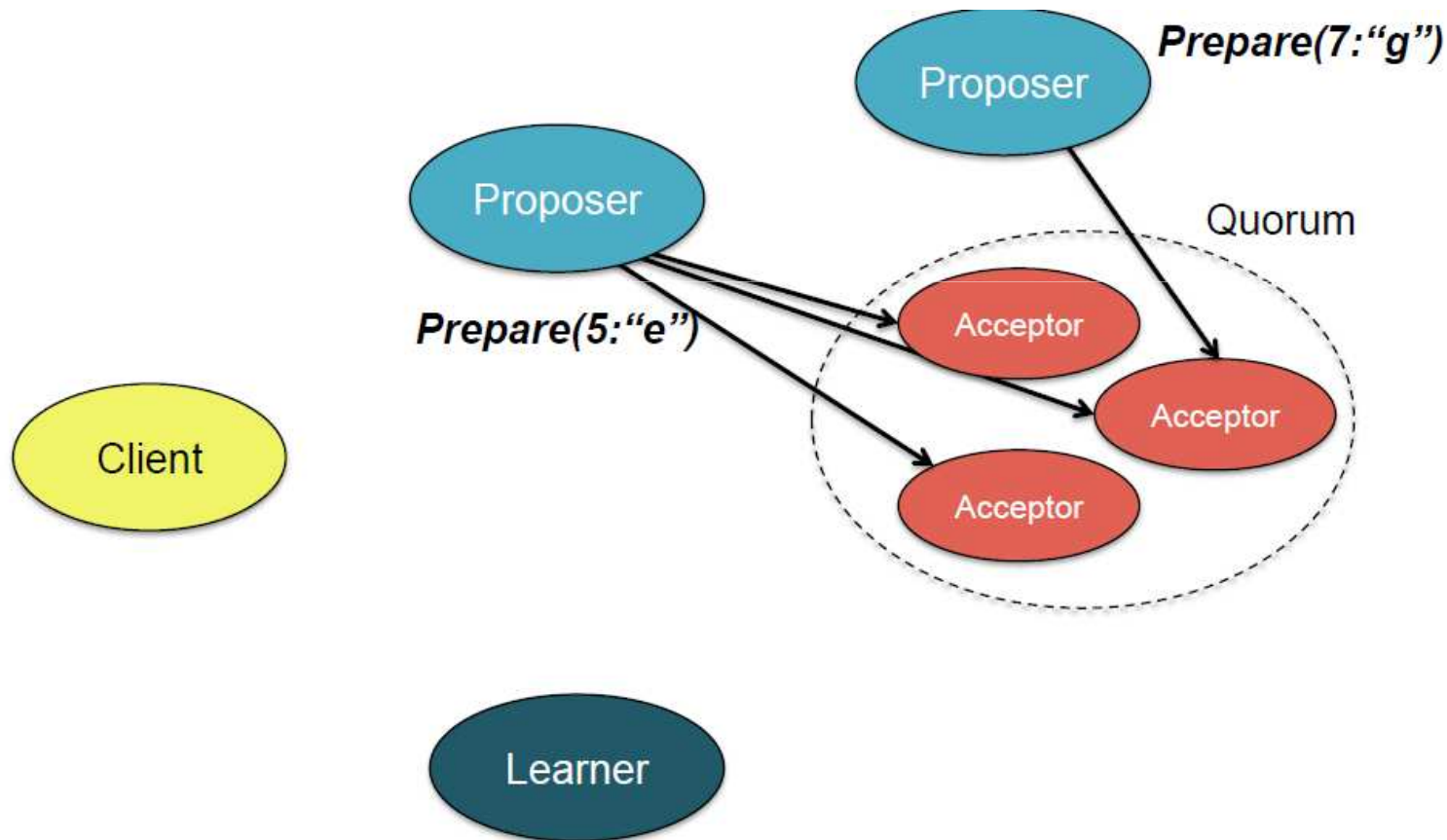
Παράδειγμα 2 – Phase 0



Phase 1a

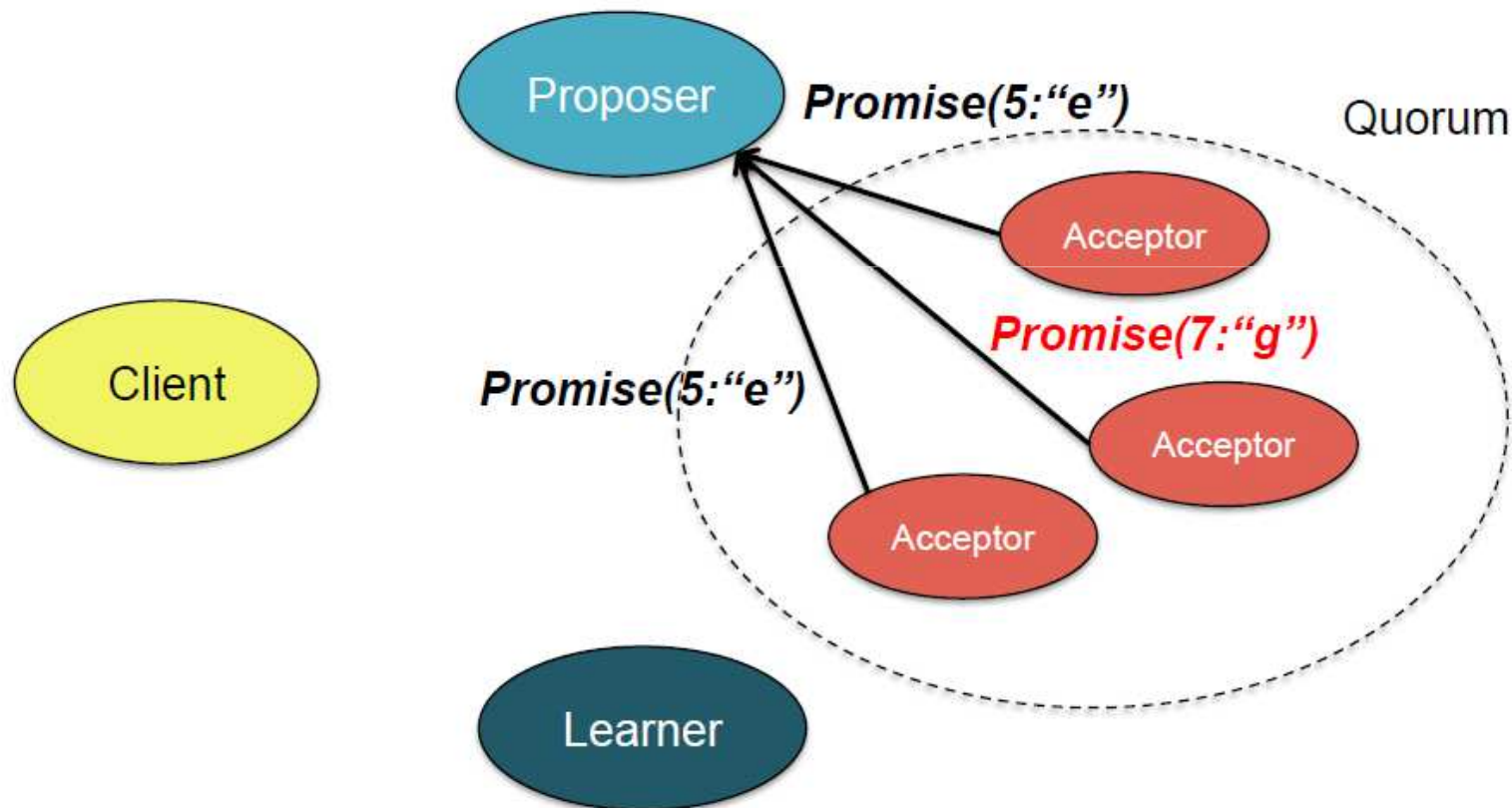
- **Proposer:** Διαλέγει id 5

Ένας acceptor λαμβάνει proposal με μεγαλύτερο id ΠΡΙΝ από το συγκεκριμένο μήνυμα PREPARE



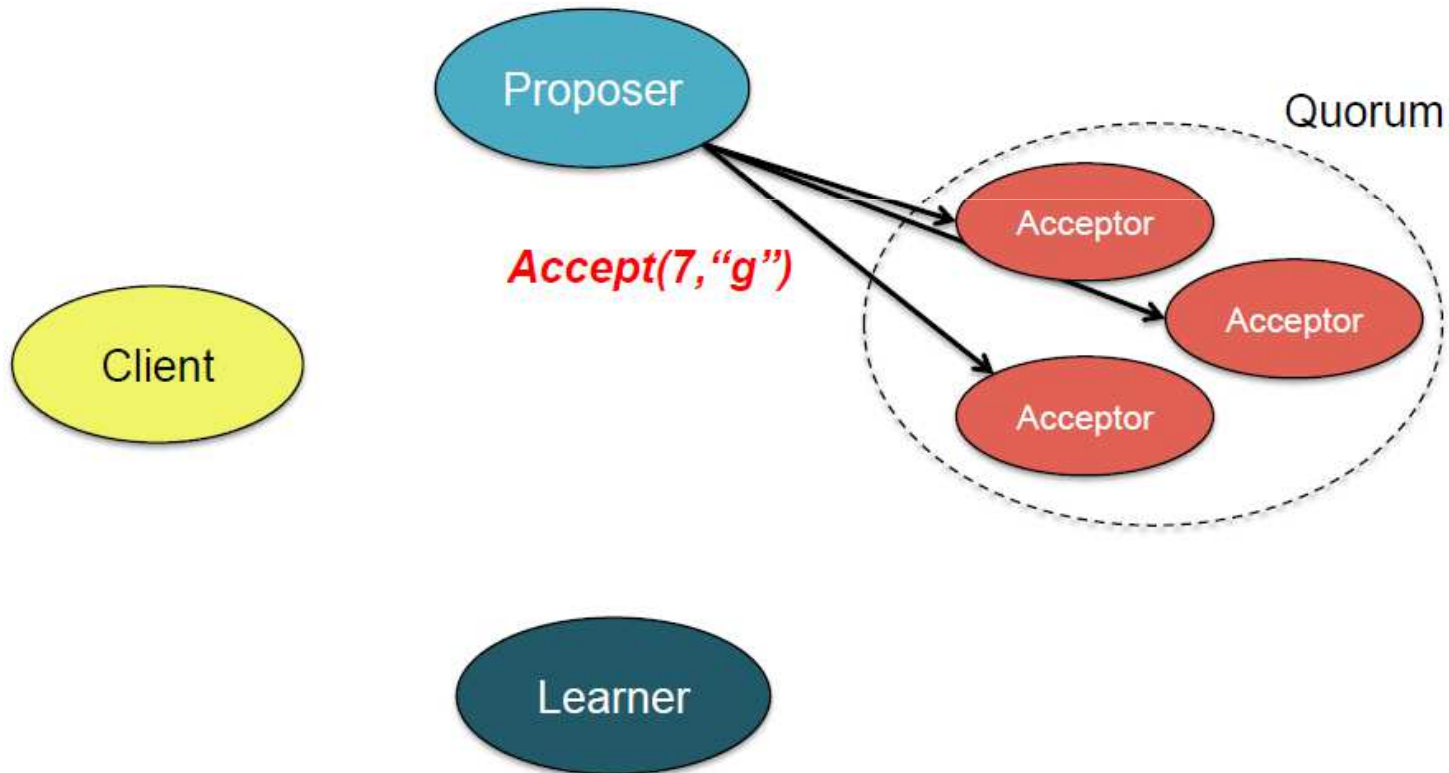
Phase 1b

- **Acceptor:** Καθένας στέλνει υπόσχεση για το μεγαλύτερο id που έχει λάβει

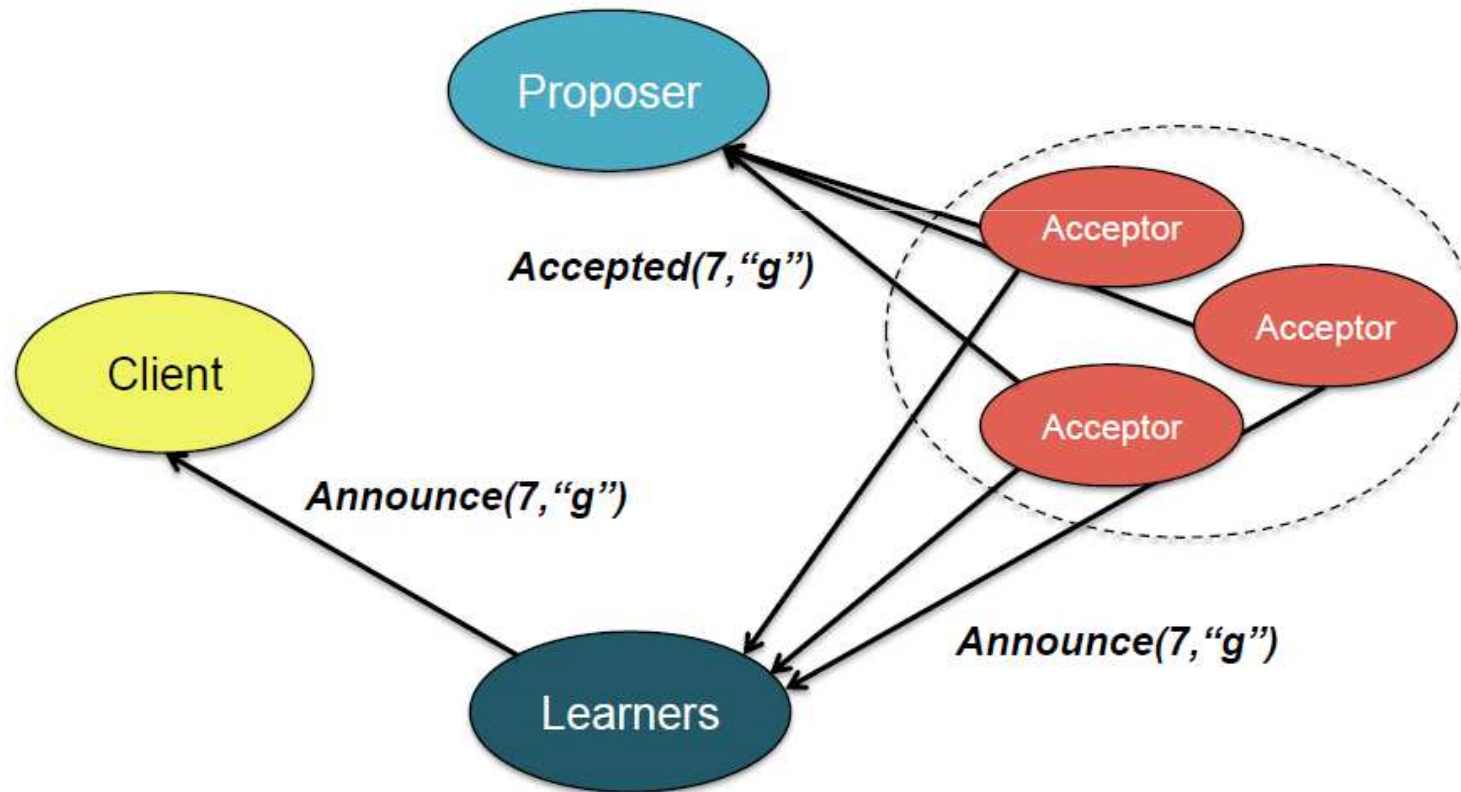


Phase 2a

- **Proposer:** Αν δεχτεί promise για μεγαλύτερο id ΠΡΕΠΕΙ να αλλάξει γνώμη και να αποδεχτεί την τιμή με το μεγαλύτερο id από οποιονδήποτε acceptor



Phase 2b



Συνέχισε να προσπαθείς

- Ένα proposal N μπορεί να αποτύχει γιατί
 - Ένας acceptor μπορεί να έχει δώσει υπόσχεση να αγνοήσει όλα τα proposals με id μικρότερο κάποιου $M > N$
 - Ένας proposer δε λαμβάνει quorum από απαντήσεις είτε στη φάση 1b είτε στη φάση 2b
- Ο αλγόριθμος τότε πρέπει να ξαναξεκινήσει με μεγαλύτερο proposal id

Επιμύθιο

- Το Ραχος μας επιτρέπει να διασφαλίσουμε συνεπή ολική διάταξη σε σύνολο από events
 - Events = εντολές, ενέργειες, ενημερώσεις κατάστασης
- Κάθε κόμβος θα έχει την τελευταία κατάσταση ή κάποια προηγούμενη εκδοχή της
- Χρησιμοποιείται σε:
 - Cassandra lightweight transactions
 - Google Chubby lock manager / name server
 - Google Spanner, Megastore
 - Microsoft Autopilot cluster management service from Bing
 - VMware NSX Controller
 - Amazon Web Services

Σύνοψη του Paxos

- Για να κάνεις αλλαγή στο σύστημα:
 - Λες στον *proposer* το *event* που θες να προσθέσεις
 - Αυτά τα αιτήματα μπορεί να συμβούν ταυτόχρονα
 - *Leader* = εκλεγμένος *proposer*. Δεν είναι απαραίτητος για τη λειτουργία του Paxos, αλλά είναι βελτιστοποίηση για διασφάλιση μοναδικής πηγής από αύξοντες αριθμούς στα *proposals*
 - Ο *proposer* επιλέγει το επόμενο *event ID* και ζητά από τους *acceptors* να το φυλάξουν
 - Αν οποιοσδήποτε *acceptor* έχει δει μεγαλύτερο *event ID*, απορρίπτει την πρόταση και επιστρέφει το *event ID* αυτό
 - Ο *proposer* προσπαθεί ξανά με μεγαλύτερο *event ID*
 - Όταν η πλειονότητα των **acceptors αποδεχτούν το proposal, τα event στέλνονται στους learners** για να αρχίσουν να ενεργούν (π.χ., να ενημερώσουν την κατάσταση του συστήματος)
 - Fault tolerant: χρειαζόμαστε $2k+1$ servers για k fault tolerance

Google Chubby

- Υπηρεσία lock
 - Επιτρέπει σε πολλαπλούς πελάτες να μοιράζονται ένα lock και να συντονίζονται
- A coarse-grained lock service
 - Locks are supposed to be held for hours and days, not seconds.
- Μπορεί να αποθηκεύει και μικρά αρχεία
- Στόχος
 - Low-rate locking/unlocking
 - Low-volume information storage
- Γιατί να χρειαζόμαστε κάτι τέτοιο;

Υποδομή της Google

- Google File System (GFS)
 - Distributed file system
- Bigtable
 - Table-based storage
- MapReduce
 - Programming paradigm & its execution framework
- Όλα αυτά βασίζονται στο Chubby

Google File System (GFS)

- Ένα cluster file system
 - ~12 disks per machine
 - Replication των αρχείων για ανοχή σε σφάλματα
- Τα αρχεία διαιρούνται σε chunks
 - 64MB/chunk
 - Κατανέμονται και αντιγράφονται σε servers
- Δύο οντότητες
 - Ένας *master*
 - Chunk servers

GFS master

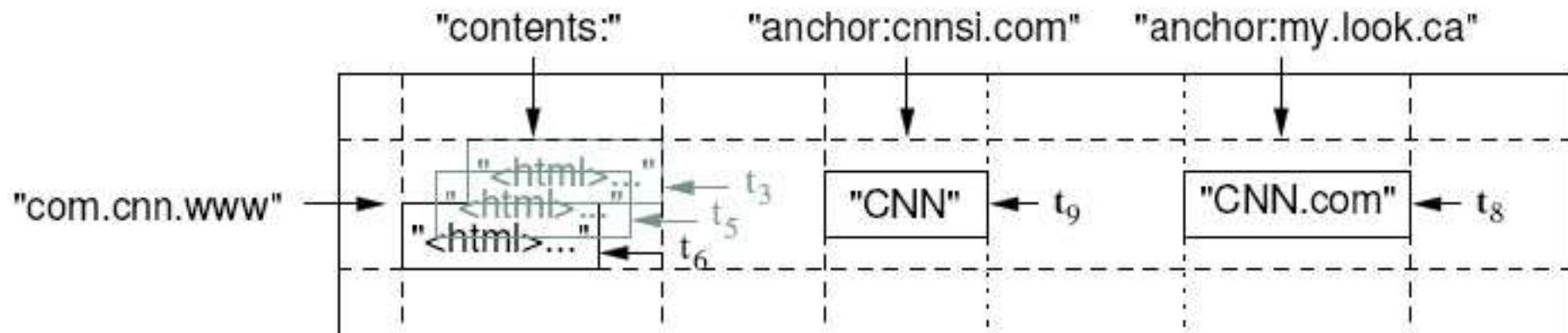
- Διατηρεί metadata του file system
 - Namespace
 - Πληροφορίες για access control
 - Αντιστοιχία filename σε chunks
 - Τρέχουσες θέσεις των chunks
- Αντιγράφει τα δεδομένα για fault tolerance
- Επικοινωνεί περιοδικά με τους chunk servers
 - μέσω heartbeat μηνυμάτων
 - Για να λάβει την κατάστασή τους και να στείλει εντολές
- Οι Chunk servers αποκρίνονται σε read/write αιτήματα και στις εντολές του master

BigTable

- Table-based storage πάνω από το GFS
- Κύριο storage για πολλά Google services
 - Google Analytics
 - Google Finance
 - Personalized search
 - Google Earth & Google Maps
- Παρέχει έναν μεγάλο λογικό πίνακα στους clients
 - Οι λογικοί πίνακες διαιρούνται σε *tablets* και κατανέμονται στους Bigtable servers.
- Τρεις οντότητες
 - Client library
 - *Ένας master*
 - Tablet servers

BigTable

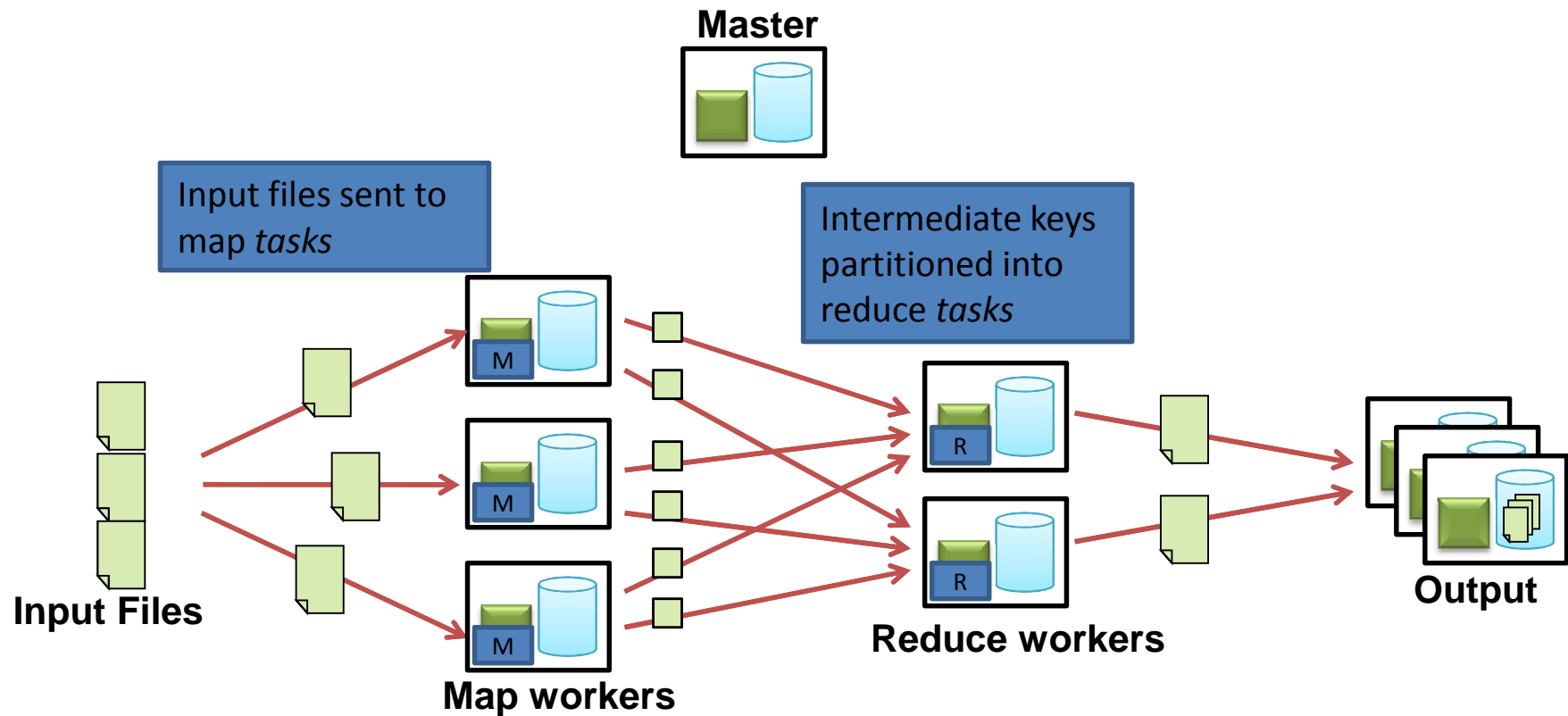
- Table: rows & columns
 - (row, column, timestamp) -> cell contents
- Π.χ., web pages και σχετικές πληροφορίες
 - Rows: URLs
 - Columns: actual web page, (out-going) links, (incoming) links, etc.
 - Versioned: using timestamps



MapReduce

- Προγραμματιστικό μοντέλο
 - Map: (key, value) \rightarrow list of (intermediate key, intermediate value)
 - Reduce: (intermediate key, list of intermediate values) \rightarrow (output key, output value)
 - Οι προγραμματιστές γράφουν συναρτήσεις Map & Reduce
- Execution framework
 - Το Google MapReduce εκτελεί τις συναρτήσεις Map & Reduce σε έναν cluster από servers
 - *Ένας master*
 - Workers

MapReduce



Κοινό σημείο

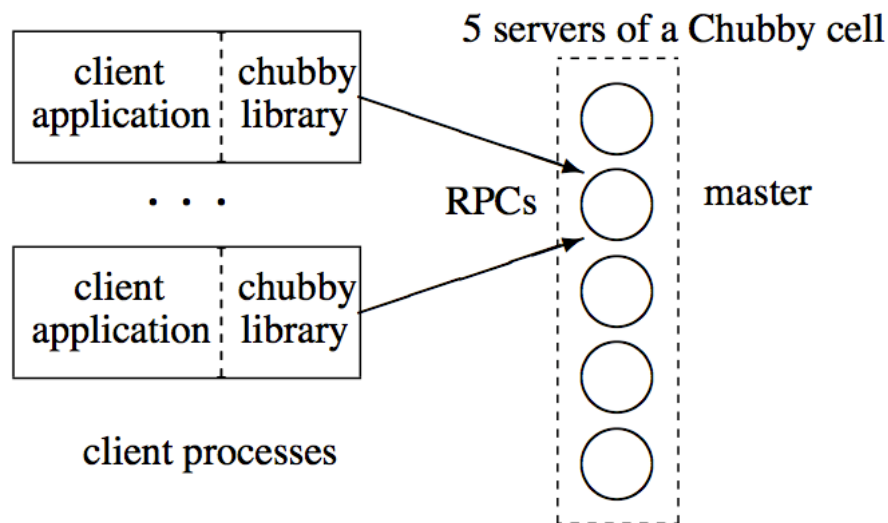
- Ένας master & πολλοί workers
- Γιατί ένας master?
 - Αυτός ο σχεδιασμός απλοποιεί πολλά πράγματα
 - Βασικά χρησιμοποιείται για τον χειρισμό μεταδεδομένων
 - Δε χρειάζεται να ασχοληθούμε με θέματα consistency
 - Χωράνε στη μνήμη → πολύ γρήγορη πρόσβαση
- Προφανές πρόβλημα: σφάλμα
 - Μπορούμε να έχουμε έναν primary και backups.
 - Μπορούμε να εκλέξουμε τον primary

Chubby

- Χρησιμοποιείται για διάφορους σκοπούς (π.χ., την εκλογή του master) στο GFS, Bigtable, MapReduce
 - Πιθανοί masters προσπαθούν να πάρουν ένα lock στο Chubby
 - Ο πρώτος που θα πάρει το lock γίνεται master
- Also used for storing small configuration data and access control lists

Οργάνωση Chubby

- Ένα chubby cell (an instance) έχει συνήθως 5 αντίγραφα
 - Αλλά κάθε κελί εξυπηρετεί δεκάδες χιλιάδες clients
- Ανάμεσα στα 5 replicas, εκλέγεται ένας master
 - Οποιοδήποτε replica μπορεί να γίνει master.
 - Η επιλογή γίνεται με Paxos.
- Ο master χειρίζεται όλα τα αιτήματα



Διεπαφή με τον client

- File system interface
 - Από την πλευρά του πελάτη είναι σαν να προσπελάζεις ένα file system.
- Typical name: `/ls/foo/wombat/pouch`
 - ls (lock service) κοινό σε όλα τα ονόματα του Chubby
 - foo είναι το όνομα του κελιού Chubby
 - /wombat/pouch μεταφράζεται μέσα στο κελί Chubby
- Περιέχει αρχεία και φακέλους που ονομάζονται κόμβοι
 - Οποιοσδήποτε κόμβος μπορεί να είναι lock για read/write: reader (shared) mode & writer (exclusive) mode
 - Τα αρχεία μπορούν να περιέχουν κάποια σύντομη πληροφορία
 - Όπως σε κάθε file system, κάθε αρχείο συσχετίζεται με κάποια metadata όπως access control λίστες

Επικοινωνία με τον client

- Οι clients (library) στέλνουν μηνύματα *KeepAlive*
 - Περιοδικές χειραψίες
 - Αν το chubby δεν λάβει τίποτα από κάποιον client τον θεωρεί νεκρό
- Οι clients εγγράφονται σε *events*.
 - Π.χ., αλλαγή περιεχομένων ενός αρχείου, προσθήκη/αλλαγή/αφαίρεση κόμβου-παιδιού, ακύρωση lock κλπ.
- Οι clients *cache data* (file & meta data)
 - Αν τα cached δεδομένα παλιώσουν, ο Chubby master τα ακυρώνει
- Ο Chubby master *στέλνει events ή ακυρώσεις cache μαζί με τα KeepAlives*
 - Διασφαλίζει ότι η cache είναι consistent

Χρήση locks

- Κάθε lock έχει έναν *“sequencer”* που είναι περίπου σαν version number
- Οι application servers που δέχονται εντολές τσεκάρουν το sequencer του application master με το chubby για να είναι σίγουροι ότι δεν έχει αλλάξει κάτι.

API

- `open()` & `close()`
- `GetContentsAndStat()`
 - Reads the whole file and meta-data
- `SetContents()`
 - Writes to the file
- `Acquire()`, `TryAcquire()`, `Release()`
 - Acquires and releases a lock associated with the file
- `GetSequencer()`, `SetSequencer()`, `CheckSequencer()`