Ένας ασύγχρονος κατανεμημένος αλγόριθμος ανταλλαγής μηνυμάτων για το πρόβλημα του K-αμοιβαίου αποκλεισμού

Αναστάσιος Τεμπερεκίδης
ΑΕΜ: 2808

Εισαγωγή

Η γενίκευση του αμοιβαίου αποκλεισμού είναι ο K-αμοιβαίος αποκλεισμός. Το πρόβλημα του K-αμοιβαίου αποκλεισμού ελέγχει το σύστημα με τέτοιο τρόπο ώστε σε κάθε χρονική στιγμή το πολύ K διεργασίες/κόμβοι να μπορούν να εισέλθουν στο κρίσιμο τομέα τους.

Περιγραφή

Μοντέλο συστήματος

Έστω $V = \{ P_1, P_2, ..., P_n \}$ είναι το σύνολο των n κόμβων/διεργασιών και $E \subseteq V \times V$ είναι το σύνολο των αμφίδρομων καναλιών επικοινωνίας στο κατανεμημένο σύστημα. Άρα η τοπολογία του κατανεμημένου συστήματος αναπαριστάται ως ένας μη-κατευθυνόμενος γράφος G=(V,E).

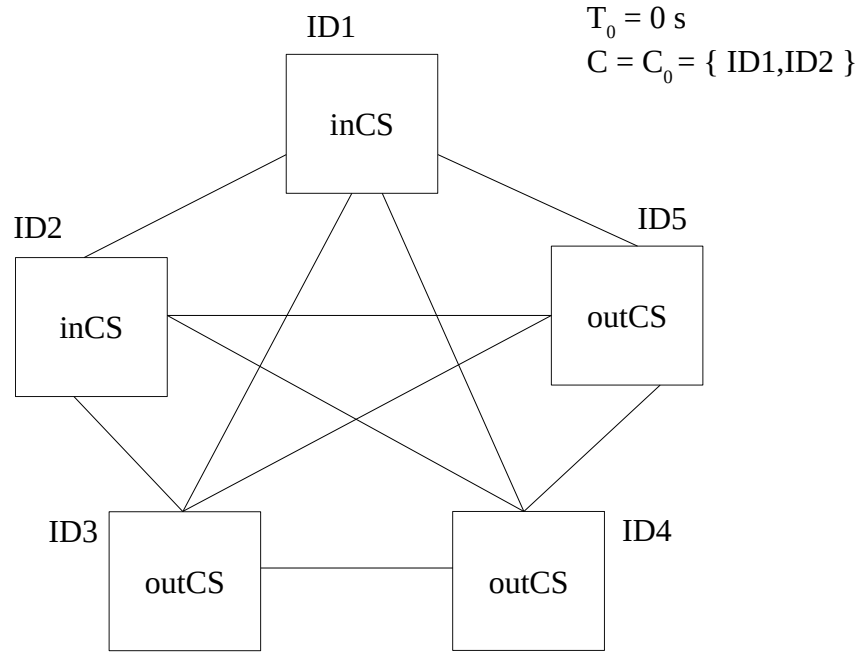
Κάνουμε τις εξής παραδοχές:

- Επίσης θεωρούμε ότι κάθε κόμβος μπορεί να επικοινωνεί με όλους τους υπόλοιπους κόμβους, άρα ο πίνακας E έχει όλα τα στοιχεία του άσσους.
- Κάθε κανάλι επικοινωνίας είναι FIFO.
- Το κατανεμημένο σύστημα είναι ασύγχρονο, δηλαδή δεν υπάρχει κάποιο κεντρικό ρολόι.
- Κάθε μήνυμα παραδίδεται τελικά, αλλά δεν υπάρχει κάποιο άνω όριο στη διάρκεια μεταφοράς του.
- Οι ταχύτητες εκτέλεσης των διεργασιών μπορεί να διαφέρουν.

Το σύνολο των τοπικών μεταβλητών ορίζουν την τοπική κατάσταση του κόμβου. Θεωρώντας Q_i την τοπική κατάσταση του κόμβου $P_i \in V$, μπορούμε να πούμε ότι η ακολουθία $(Q_1, Q_2, ..., Q_n)$ αποτελεί την κατάσταση του κατανεμημένου συστήματος.

Τέλος, θεωρούμε ότι κάθε κόμβος $P_i \in V$ έχει μία μεταβλητή εν ονόματι $state_i \in \{ InCS, OutCS \}$. Για κάθε κατάσταση του κατανεμημένου συστήματος C, ορίζουμε ως CS(C) (και αντίστοιχα CS'(C)) το σύνολο των κόμβων P_i με $state_i$ = InCS (και αντίστοιχα για το συμπληρωματικό σύνολο $state_i$ = OutCS). Ενώ υποθέτουμε ότι υπάρχει πρόοδος στο κρίσιμο τομέα κάθε κόμβου δηλαδή η συμπεριφορά κάθε κόμβου P_i είναι η ακόλουθη: θεωρούμε ότι κάθε κόμβος τελικά καλεί την entry-sequence όταν είναι εκτός του κρίσιμου τομέα του και επίσης ο κόμβος τελικά καλεί την exit-sequence όταν είναι εντός του κρίσιμου τομέα του.

```
state_i = Initial state of Pi
loop {
        if (state_i = OutCS) {
                entry_sequence();
                state_i = InCS;
                /* Critical Section */
        }
        else {
                exit_sequence();
                state_i = OutCS;
                /* Remainder Section */
        }
}
```



Σχήμα 1: Μια τυχαία αρχική κατάσταση κατά την έναρξη του αλγόριθμου

Περιγραφή προβλήματος

Ορισμός: Έστω ο αριθμός k τέτοιος ώστε $0 \le k \le |N|$, όπου |N| το πλήθος των συνδεδεμένων κόμβων του συστήματος. Τότε ένας αλγόριθμος επιλύει το πρόβλημα του k-αμοιβαίου αποκλεισμού για το κρίσιμο τομέα του γράφου G αν και μόνο αν οι δύο παρακάτω προτάσεις διατηρούνται σε κάθε συνολική κατάσταση C_i του συστήματος:

- **Ιδιότητα ασφάλειας:** $| CS(C_i) | \le k$ σε κάθε χρονική στιγμή.
- **Ιδιότητα ζωτικότητας:** Κάθε κόμβος $P_i \in V$ αλλάζει κατάσταση από OutCS σε InCS και το αντίστροφο απείρως συχνά.

Υποθέτουμε ότι η αρχική κατάσταση του συστήματος C_0 είναι ασφαλής, που σημαίνει ότι ισχύει $| CS(C_0) | \le k$.

Αλγόριθμος

Η αρχική κατάσταση C_0 είναι ασφαλής (από τις προηγούμενες υποθέσεις). Όταν ο κόμβος P_i θέλει να μπει στο κρίσιμο τμήμα του, ζητάει την άδεια στέλνοντας ένα μήνυμα τύπου <u>*<Request, ts_i, P_i>*</u> σε όλους τους υπόλοιπους |N| - 1 κόμβους του συστήματος. Όταν ο κόμβος P_i πάρει την άδεια λαμβάνοντας ένα μήνυμα τύπου <u>*<Grant>*</u> από όλους τους |N| - 1 κόμβους αλλάζει την κατάσταση του σε inCS και εκτελεί το κρίσιμο τομέα του. Κάθε κόμβος P_i μπορεί να αδειοδοτήσει μέχρι k κόμβους σε κάθε χρονική στιγμή. Επομένως κάθε χρονική στιγμή τουλάχιστον |N| - k κόμβοι δεν μπορούν να μπουν στο κρίσιμο τμήμα τους ενώ το έχουν ζητήσει. Όταν ένας κόμβος θέλει να αποχωρήσει από το κρίσιμο τμήμα του, αλλάζει την κατάσταση του από inCS σε outCS και στέλνει ένα μήνυμα τύπου <u>*<Release, P_i>*</u> σε καθένα από τους υπόλοιπους |N| - 1 κόμβους του συστήματος, για να μπορούν οι κόμβοι να διαχειριστούν μια νέα αίτηση πρόσβασης στο κρίσιμο τομέα από κάποιον κόμβο. Η προτεραιότητα βασίζεται στην χρονοσφραγίδα που έχει κάθε κόμβος, σε κάθε μήνυμα τύπου Request για την πρόσβαση ενός κόμβου στο κρίσιμο τομέα του, αποστέλλεται και η χρονοσφραγίδα του κόμβου αυτού, έτσι , ένας κόμβος μπορεί να προεκτοπίσει/ακυρώσει μια άδεια που έστειλε σε έναν άλλο κόμβο που ζήτησε άδεια, όποτε αυτό είναι απαραίτητο. Όταν ενας κόμβος θέλει να προεκτοπίσει κάποιον στέλνει ένα μήνυμα τύπου <u><Preempt, P_i></u> ενώ ένας κόμβος ανταποκρίνεται στο αίτημα προεκτόπισης με την αποστολή ενός μηνύματος τύπου <u><Relinquish, P_i></u> στον αρχικό κόμβο.

Κάθε κόμβος P_i διαθέτει τις παρακάτω τοπικές μεταβλητές:

- **$state_i$:** Η τρέχουσα κατάσταση του P_i (inCS / outCS)
- **ts_i :** Η τρέχουσα τιμή του λογικού ρολογιού του κόμβου P_i.
- ***$nGrants_i$*:** Ο αριθμός των αδειών που ο κόμβος P_i έχει λάβει για να εισέλθει στο κρίσιμο τομέα του
- **$grantedTo_i$:** Σύνολο από ζεύγη χρονοσφραγίδων και αντίστοιχων ID κόμβων (ts_j , P_j) , για τα αιτήματα του P_j να εισέλθει στο κρίσιμο τομέα του όπου ο P_i έχει αδειοδοτήσει, αλλά ο κόμβος P_j δεν έχει ακόμα στείλει μήνυμα τύπου Released.
- **$pendingReq_i$** : Σύνολο από ζεύγη χρονοσφραγίδων και αντίστοιχων ID κόμβων (ts_j , P_j) για τα αιτήματα του P_j να εισέλθει στο κρίσιμο τομέα του που ακόμα εκρεμμούν (δεν έχει δοθεί αδειοδότηση με αποστολή μηνύματος τύπου Grant από τον κόμβο P_i)
- **$preemptingNow_i$:** Το αναγνωριστικό του κόμβου P_j όπου η P_i έχει στείλει μήνυμα προεκτόπισης της άδειας εισαγωγής του κόμβου P_j στο κρίσιμο τομέα του και η διαδικασία προεκτόπισης είναι σε εξέλιξη.

* Σε κάθε ζεύγος (ts_j , P_j) , το ts_j αντιστοιχεί στη τιμή του λογικού ρολογιού του κόμβου P_j όταν έστειλε το μήνυμα. Επιπλέον οι χρονοσφραγίδες θεωρούμε ότι διατάσσονται με τον εξής τρόπο:

$$(ts_i, P_i) < (ts_j , P_j) \text{ αν και μόνο αν } ts_i < ts_j \text{ ή } (ts_i = ts_j) \text{ και } (P_i < P_j)$$

Παρακάτω εξηγούμε τις λειτουργίες που μπορεί να εκτελέσει κάθε κόμβος. Όταν ένας κόμβος P_i λαμβάνει ένα μήνυμα, τότε εκτελεί την αντίστοιχη συνάρτηση διαχείρισης του μηνύματος (message handler). Κάθε συνάρτηση διαχείρισης μηνύματος εκτελείται ατομικά. Αυτό σημαίνει πως αν μια συνάρτηση μηνύματος εκτελείται, η άφιξη ενός νέου μηνύματος δεν θα διακόψει την εκτέλεση της.

```
function entry_sequence() {
        ts_i += 1;
        foreach Pj ∈ V send <Request, ts_i , P_i > to P_j ;
        wait until ( nGrants_i = |N| );
        state_i = inCS;
}
```

Όταν ο κόμβος επιθυμεί να εισέλθει στο κρίσιμο τομέα του, τότε εκτελεί την entry_sequence(). Αυξάνει την τιμή του λογικού ρολογιού κατά ένα, και στέλνει μηνύματα τύπου Request στους υπόλοιπους κόμβους. Έπειτα περιμένει μέχρι να λάβει από όλους τους κόμβους μήνυμα τίπου Grant. Όσο περιμένει, η συγκεκριμένη εκτέλεση της συνάρτησης μπλοκάρεται μέχρι η σύνθηκη ($nGrants_i = |N| - 1$) γίνει αληθής. Όσο η διαδικασία είναι μπλοκαρισμένη σε αυτό το σημείο (wait until) , και λαμβάνει ένα μήνυμα, εκτελεί τον αντίστοιχο διαχειριστή μηνύματος. Όταν λάβει όλα τα μηνύματα αδειοδότητης αλλάζει την κατάσταση του από outCS σε inCS.

```
function exit_sequence() {
        state_i = outCS;
        foreach Pj ∈ V send <Release, P_i > to P_j ;
}
```

Όταν ο κόμβος θέλει να αποχωρήσει από το κρίσιμο τομέα του, τότε αλλάζει την κατάσταση του από inCS σε outCS και ενημερώνει τους υπόλοιπους κόμβους στέλνοντας ένα μήνυμα τύπου Release σε καθένα από αυτούς.

Παρακάτω αναλύουμε τους διαχειριστές μηνυμάτων όπου υλοποιεί κάθε κόμβος του κατανεμημένου συστήματος.

```
function on receive_request( <Request, ts_j , P_j > ) {
        pendingReq_i.add( ( ts_j , P_j ) )
        if ( |grantedTo_i| < |k| ) {
                ( ts_h , P_h ) = deleteMin(pendingReq_i);
                grantedTo_i.add( ( ts_h , P_h ) );
                send <Grant> to P_h ;
        }
        else if ( preemptingNow_i = null ) {
                ( ts_h , P_h ) = max(pendingReq_i);
                if (  ( ts_j , P_j ) < ( ts_h , P_h )  ) {
                        preemptingNow_i  = P_h ;
                        send <Preempt, P_i > to P_h ;
                }
        }
}
```

Όταν ένας κόμβος P_i λαμβάνει ένα μήνυμα τύπου Request, προσθέτει την χρονοσφραγίδα-ID στο σύνολο των ζεύγων χρονοσφραγίδων-ID κόμβων που η αδειοδότηση τους εκκρεμεί από τον κόμβο P_i . Στη συνέχεια, αν έχει αδειοδοτήσει τη δεδομένη χρονική στιγμή λιγότερους από |k| κόμβους, τότε στέλνει ένα μήνυμα τύπου Grant σε αυτον κόμβο με την μεγαλύτερη προτεραιότητα και τον αφαιρεί από το σύνολο εκκρεμών αδειοδοτήσεων. Διαφορετικά ελέγχει αν έχει δώσει άδεια σε έναν κόμβο που δεν έχει μεγαλύτερη προτεραιότητα από τον κόμβο P_j , κάτι τέτοιο θα μπορούσε να συμβεί αφού τα μηνύματα μεταφέρονται διαμέσου καναλιών επικοινωνίας και δεν γνωρίζουμε πότε θα ληφθεί ένα μήνυμα. Αν πράγματι ο P_i έχει αδειοδοτήσει έναν κόμβο με μικρότερη προτεραιότητα, τότε στέλνει ένα μήνυμα τύπου Preempt σε αυτόν.

```
function on receive_grant( <Grant> ) {
        nGrants_i += 1;
}
```

Όταν ένας κόμβος λαμβάνει ένα μήνυμα τύπου Grant, απλά αυξάνει τον μετρητή $nGrants_i$ κάτα ένα.

```
function on receive_preempt( <Preempt,P_j> ) {
        if ( state_i = outCS ) {
                nGrants_i -= 1;
                send <Relinquish,  P_i > to  P_j ;
        }
}
```

Όταν ένας κόμβος P_i λαμβάνει ένα μήνυμα τύπου Preempt από τον κόμβο P_j σημαίνει ότι είχε ζητήσει την άδεια από τον P_j και στη συνέχεια του έστειλε ο P_j ένα μήνυμα τύπου Grant. Έπειτα ο P_j θέλησε να προεκτοπίσει τον P_i και του έστειλε μήνυμα τύπου Preempt. Όμως κατά τη λήψη του μηνύματος, ο P_i βρίσκεται σε μια από τις παρακάτω καταστάσεις:

Έχει πάρει την άδεια από όλους τους κόμβους και άρα βρίσκεται στο κρίσιμο τομέα του. Σε αυτή την περίπτωση αγνοεί το μήνυμα προεκτόπισης.

Δεν έχει πάρει την άδεια από όλους τους κόμβους, σε αυτή την περίπτωση ανταποκρίνεται στο μήνυμα προεκτόπισης μειώνοντας κατά ένα το πλήθος τον αδειών που έχει λάβει μέχρι στιγμής, και απαντώντας με ένα μήνυμα τύπου Reliquish (παραίτηση) στον κόμβο P_j .

function on receive_reliquish(<Reliquish,P_j>) {
    $preemptingNow_i$ = null ;
    Delete (*, P_j) from $grantedTo_i$ and let (ts_j , P_j) be the deleted item;
    $pendingReq_i$.add((ts_j , P_j));
    (ts_h , P_h) = deleteMin($pendingReq_i$);
    $grantedTo_i$ .add((ts_h , P_h));
    send <Grant> **to** P_h;
}

Όταν ένας κόμβος P_i λαμβάνει ένα μήνυμα τύπου Reliquish από τον κόμβο P_j σημαίνει ότι ο κόμβος P_j ανταποκρίθηκε στο αίτημα του να προεκτοπιστεί. Και άρα διαγράφει από το σύνολο αδειών, την άδεια που είχε δώσει στον P_j , προσθέτει την αίτηση του P_j στην λίστα εκκρεμών αδειοδοτήσεων, και στέλνει ένα μήνυμα τύπου Grant στον κόμβο που του έχει ζητήσει άδεια και έχει τη μεγαλύτερη προτεραιότητα να εισέλθει στο κρίσιμο τομέα του.

Απόδειξη ορθότητας

Απόδειξη ιδιότητας ασφάλειας: Ισχύει | $CS(C_i)$ | ≤ k σε κάθε χρονική στιγμή για κάθε κατάσταση του κατανεμημένου συστήματος C.

Απόδειξη
Αρχικά θεωρούμε ότι η αρχική κατάσταση του συστήματος C_0 είναι ασφαλής, δηλαδή | $CS(C_0)$ | ≤ k. Έστω ότι σε κάποια χρονική στιγμή έχουμε μια κατάσταση C_r όπου δεν είναι ασφαλής, δηλαδή | $CS(C_r)$ | > k. Επειδή | $CS(C_0)$ | ≤ k , θεωρούμε έναν κόμβο $P_j \in V$ όπου περιμένει να μπει και έχει την (k + 1)$^{\eta}$ χαμηλότερη χρονοσφραγίδα από το σύνολο $CS(C_r)$. Στη συνέχεια, ο κόμβος P_j ζητάει την άδεια για να μπει στο κρίσιμο τμήμα του, από κάθε κόμβο $P_i \in V$, στέλνοντας μηνύματα τύπου Request σε καθένα από αυτούς. Αυτό σημαίνει ότι ένας κόμβος P_i λαμβάνει το μήνυμα αίτησης αδειοδότησης <Request, ts_j , P_j> από τον κόμβο P_j και στέλνει άδεια <Grant> στον P_j . Όμως, επειδή ο κόμβος P_i μπορεί να αδειοδοτήσει το πολύ k κόμβους να μπουν στο κρίσιμο τμήμα τους σε κάθε χρονική στιγμή, αυτό σημαίνει ότι ο κόμβος P_j δε μπορεί να λάβει μήνυμα αδειοδότησης από τον κόμβο P_i και άρα η συνθήκη της διαδικασίας entry_sequence() δεν μπορεί να γίνει αληθής, ώστε να ολοκληρωθεί η εκτέλεση της και να μπει στο κρίσιμο τμήμα του.

Απόδειξη ιδιότητας ζωτικότητας: Κάθε κόμβος $P_i \in V$ αλλάζει κατάσταση από OutCS σε InCS και το αντίστροφο απείρως συχνά.

Απόδειξη

Έστω ότι κάποιοι κόμβοι δεν αλλάζουν την κατάσταση τους από inCS σε outCS και αντίστροφα απείρως συχνά. Έστω ένας κόμβος P_i όπου έχει την υψηλότερη προτεραιότητα να μπει στο κρίσιμο τομέα του (ts_i, P_i). Χωρίς απώλεια της γενικότητας, υποθέτουμε ότι ο κόμβος P_i μπλοκάρεται στην κατάσταση outCS. Αυτό σημαίνει ότι η εκτέλεση του κόμβου P_i έχει κολλήσει στην εντολή "wait-until" της συνάρτησης entry-sequence() - (υπενθυμίζουμε ότι κάθε κόμβος ο οποίος βρίσκεται στο κρίσιμο τμήμα του μπορεί να αλλάξει την κατάσταση του σε outCS χωρίς την αναμονή κάποιας απάντησης από κάποιον άλλο κόμβο). Θεωρούμε έναν οποιονδήποτε κόμβο $P_j \in V$.

- Αν ο κόμβος P_j αλλάζει την κατάσταση του από inCS σε outCS και αντίστροφα απείρως συχνά τότε κάποια στιγμή ο P_j λαμβάνει αίτημα <Request, ts_i, P_i > από τον κόμβο P_i, η τιμή της χρονοσφραγίδας του P_j (ts_j, P_j) από το αίτημα του P_j ξεπερνά την αντίστοιχη τιμή της χρονοσφραγίδας (ts_i, P_i) του αιτήματος του P_i. Επειδή από τον αλγόριθμος, το αίτημα είσχωρησης στο κρίσιμο τομέα με την μικρότερη χρονοσφραγίδα αδειοδοτείται κατά προτίμηση, είναι αδύνατο ο κόμβος P_j να εναλλάσει την κατάσταση του από inCS σε outCS και το αντίστροφο απείρως συχνά. Άρα τελικά ο κόμβος P_j στέλνει ένα μήνυμα τύπου Grant στον κόμβο P_i και ο P_i στέλνει ένα μήνυμα τύπου Grant στον εαυτό του.

- Αν ο κόμβος P_j δεν αλλάζει την κατάσταση του από inCS σε outCS και αντίστροφα απείρως συχνά τότε από τις υποθέσεις επειδή η χρονοσφραγίδα του P_i είναι μικρότερη από την χρονοσφραγίδα του P_j , η αδειοδότηση του κόμβου P_j για εισαγωγή στο κρίσιμο τομέα του, ακυρώνεται, με την αποστολή ενός μηνύματος τύπου Preempt ενώ στέλνεται ένα μήνυμα τύπου Grant από τον P_j στον P_i. Επιπλέον ο κόμβος P_i στέλνει ένα μήνυμα τύπου Grant στον εαυτό του.

Συνεπώς ο κόμβος P_i και στις δύο συμπληρωματικές υποθέσεις λαμβάνει τελικά μήνυμα τύπου Grant από όλους τους κόμβους $P_j \in V$ και η εντολή "wait-until" στην συνάρτηση entry_sequence() δεν μπλοκάρει για πάντα τον κόμβο P_i.

Ανάλυση πολυπλοκότητας

- Ανάλυση χειρότερης περίπτωσης

Η πολυπλοκότητα μηνυμάτων του k-αμοιβαίου αποκλεισμού (k-mutex) για κάθε κόμβο $P_i \in V$ στη χειρότερη περίπτωση είναι 6 | N | , όπου N το σύνολο των κόμβων.

Απόδειξη

Όταν ένας κόμβος P_i θέλει να εισέλθει στο κρίσιμο τομέα του, καλεί την συνάρτηση entry_sequence(). Η συνάρτηση αυτή στέλνει μηνύματα τύπου <Request, ts_i , P_i > σε κάθε ένα από τους |N| κόμβους $P_j \in V$. Στη συνέχεια στη χειρότερη περίπτωση κάθε κόμβος στέλνει μήνυμα προεκτόπισης στον κόμβο P_m τον οποίο ο κόμβος P_j προηγουμένως έστειλε μήνυμα Grant, τότε ο κόμβος P_m ,στέλνει ένα μήνυμα παραίτησης <Reliquish,P_m> πίσω στον κόμβο P_j και στη συνέχεια ο κόμβος P_j στέλνει ένα μήνυμα αδειοδότησης στον κόμβο P_i . Κατά την έξοδο του κόμβου P_i από τον κρίσιμο τομέα του, στέλνει μηνύματα τύπου <Release, P_i > σε κάθε κόμβο $P_j \in V$. Τότε κάθε κόμβος $P_j \in V$ στέλνει ένα μήνυμα τύπου <Grant> στο P_m ,για να επιστρέψει την άδεια που του είχε ακυρώσει ή στέλνει άδεια σε κάποιον κόμβο με υψηλότερη προτεραιότητα στη λίστα εκκρεμών αδειοδοτήσεων $pendingReq_j$. Επομένως συνολικά στέλνονται 6 | N | μηνύματα.

Θέματα υλοποίησης

Ο ασύγχρονος κατανεμημένος αλγόριθμος υλοποιήθηκε σε γλώσσα Rust. Πιο συγκεκριμένα, χρησιμοποιήθηκαν τα κανάλια της γλώσσας (many producers single consumer) και η βιβλιοθήκη thread για την δημιουργία των νημάτων. Επίσης για την απεικόνιση της κατάστασης του συστήματος κάθε χρονική στιγμή, γίνεται print στην κονσόλα όλα τα είδη των μηνυμάτων που λαμβάνουν χώρα εκείνη τη δεδομένη στιγμή. Επιπλέον για την ευκολότερη κατανόηση της κατάστασης του συστήματος, εμφανίζεται σε ένα γραφικό περιβάλλον το σύνολο των κόμβων του συστήματος, ο καθένας χρωματισμένος ανάλογα με την κατάσταση στην οποία βρίσκεται. Πιο αναλυτικά, κάθε κόμβος αναπαριστάται ως ένα τετράγωνο, αν ο κόμβος είναι εκτός κρίσιμου τομέα και δεν θέλει να εισέλθει σε αυτό, τότε είναι χρωματισμένος με κόκκινο. Αν ο κόμβος είναι εκτός κρίσιμου τομέα αλλά θέλει να εισέλθει σε αυτό, τότε είναι χρωματισμένος με μωβ. Τέλος, αν ο κόμβος είναι μέσα στο κρίσιμο τομέα του, τότε είναι χρωματισμένος με πράσινο. Επιπλέον δίνεται η δυνατότητα ρύθμισης παραμέτρων της ανταγωνισιμότητας των κόμβων.Ο αλγόριθμος απαιτεί την ύπαρξη καναλιών δύο κατευθύνσεων για κάθε ζευγάρι κόμβων. Στην υλοποίηση παρακάμψαμε αυτή την απαίτηση με το να θέσουμε το main thread ως διαμεσολαβητή των μηνυμάτων, άρα άτυπα στην υλοποιήση υπάρχει ένας “κεντρικός” κόμβος όπου προωθούνται τα μηνύματα μέσω αυτού στους υπόλοιπους κόμβους. Βέβαια αυτό δεν αναιρεί την αποκεντρισιμότητα του αλγόριθμου αυτού, καθώς μπορούμε να δημιουργήσουμε κανάλια διπλής κατεύθυνσης για κάθε ζευγάρι κόμβων.Τέλος ο αλγόριθμος του K-mutex, σε συνδυασμό με το συμπληρωματικό πρόβλημα του L-mutin, όπου έκει θέλουμε κάθε χρονική στιγμή να βρίσκονται το λιγότερο L κόμβοι στο κρίσιμο τομέα τους, μπορούν να σχεδιάσουν έναν αλγόριθμο για την επίλυση ενός πιο γενικού προβλήματος, του (L,K)-αμοιβαίου αποκλεισμού, όπου θέλουμε κάθε χρονική στιγμή να έχουμε το λιγότερο L κόμβους και το πολύ K κόμβους στο κρίσιμο τομέα τους. Μάλιστα το πρόβλημα μπορεί να γενικοποιηθεί ακόμα περισσότερο, και να σπάσουμε τον κανόνα της απαραίτητης ύπαρξης καναλιών διπλής κατεύθυνσης για κάθε ζευγάρι κόμβων, με την ανάπτυξη ενός αλγόριθμου για την επίλυση του προβλήματος του (L_i , K_i) – αμοιβαίου αποκλεισμού. Στο πρόβλημα αυτό κάθε κόμβος P_i έχει κάποιους συνδεδεμένους κόμβους (γείτονες) του συστήματος N_i , και για κάθε κόμβο ισχύει ο περιορισμός (L_i , K_i) , δηλαδή στη γειτονιά του κόμβου αυτού θέλουμε κάθε χρονική στιγμή να έχουμε το πολύ K_i και το λιγότερο L_i γείτονες-κόμβους στο κρίσιμο τομέα τους. Βέβαια η επίλυση αυτού το προβλήματος δε γίνεται με μια απλή σύνθεση του παραπάνω αλγόριθμου με έναν αντίστοιχο αλγόριθμο για την επίλυση του προβλήματος L-mutin. Σκεφτείται ένα αλγόριθμο όπου θα εχεί 2-τύπους μηνυμάτων, κάποια που αφορούν το mutex και κάποια το mutin. Όταν ένας αλγόριθμος θέλει να εισέλθει στο κρίσιμο τομέα του, θα πρέπει να στείλει requests τύπου mutex στους κόμβους-γείτονες, ενώ μια τέτοια κίνηση δεν έχει επιπτώσεις στον περιορισμό που θέτει το L-mutin πρόβλημα. Αντίθετα, όταν θέλει να βγει ένας κόμβος από το κρίσιμο τομέα του, θα πρέπει να στείλει requests τύπου mutin στους κόμβους γείτονες, ενώ μια τέτοια κίνηση δεν έχει επιπτώσεις στον περιορισμό που θέτει το K-mutex πρόβλημα. Όμως μια απλή σύνθεση αυτών των δύο αλγορίθμων που λύνουν συμπληρωματικά προβλήματα, για την δημιουργία ενός αλγορίθμου επίλυσης του γενικότερου προβλήματος, οδηγεί σε έναν αλγόριθμο με αδιέξοδο. Ας σκεφτούμε το εξής σενάριο: Έστω ότι υπάρχει ένας κόμβος P_r που είναι εντός του κρίσιμου τομέα του. Και έστω ότι το σύνολο των γειτόνων του που είναι μέσα στο κρίσιμο τομέα τους είναι το ελάχιστο δυνατό, δηλαδή L_r ή έχει έναν γείτονα P_w όπου στην γειτονιά του ισχύει ότι το ελάχιστο επιτρεπτό πλήθος γειτόνων είναι στο κρίσιμο τομέα τους. Τότε ο κόμβος P_r δεν μπορεί να αλλάξει την κατάσταση του, μέχρι τουλάχιστον ένας από τους γείτονες του έστω P_w ο οποίος είναι έξω από το κρίσιμο τομέα του, αλλάξει την κατάσταση του. Αν όμως για τη γειτονιά του P_w εκείνη τη χρονική στιγμή ισχύει ότι το μέγιστο επιτρεπτό πλήθος κόμβων βρίσκονται στο κρίσιμο τομέα τους ή ο P_w έχει έναν γείτονα P_x όπου στη γειτονιά του ισχύει ότι το μέγιστο επιτρεπτό πλήθος γειτόνων είναι στο κρίσιμο τομέα τους, τότε ο P_w δε μπορεί να αλλάξει την κατάσταση του από state=outCS σε state=inCS, μέχρι τουλάχιστον ένας από τους γείτονες του P_t που βρίσκεται μέσα στο κρίσιμο τομέα του βγει από αυτόν κ.ο.κ. Αν κάθε κόμβος βρίσκεται σε μια τέτοια κατάσταση, τότε θα οδηγηθούμε σε deadlock.
Μια προσπάθεια για την επίλυση αυτού του προβλήματος, αλλά και περισσότερες λεπτομέριες για αυτό, αναλύονται σε αυτή τη δημοσίευση: https://www.mdpi.com/1999-4893/10/2/38/pdf.