

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

**ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ**



Διπλωματική εργασία
Γάκη Ευάγγελου Α.Μ. 264

**Υλοποίηση αλγορίθμων προγράμματος
(packets scheduling) για ασύρματα δίκτυα με
μερική γνώση καναλιού**

Εισηγητές:
Τασιούλας Λέανδρος
Πάσχος Γεώργιος

Βόλος 2011-2012

Περιεχόμενα

ΑΝΤΙΚΕΙΜΕΝΟ ΤΗΣ ΕΡΓΑΣΙΑΣ	3
ΕΙΣΑΓΩΓΗ.....	3
Προσεκτικότερη παρατήρηση της διαδικασίας μετάδοσης πακέτων	4
Uplink φάση	5
Downlink φάση	5
Feedback φάση.....	6
Η αξία του feedback	7
Αλγόριθμος για ντετερμινιστική γνώση	7
Συστήματα με στατιστική γνώση	8
<i>The virtual network mechanism</i>	8
Αλγόριθμος για στατιστική γνώση	11
Μεταφορά στη C++.....	12
Vectors(διανύσματα)	13
Υλοποίηση του αλγορίθμου.....	14
Main.....	14
Arrive	16
bi2dec, dec2tri, tri2dec, Qmatrix.....	18
Προσομοίωση main_qba	21
Συνάρτηση main_qba	21
Συνάρτηση scheduler	23
Συνάρτηση maxweight	24
Συμπεράσματα για τις προσομοιώσεις	29
Ανάλυση με arrays	30
Διάγραμμα προσομοίωσης.....	33
Σύγκριση αποδοτικότητας προσομοιώσεων	34

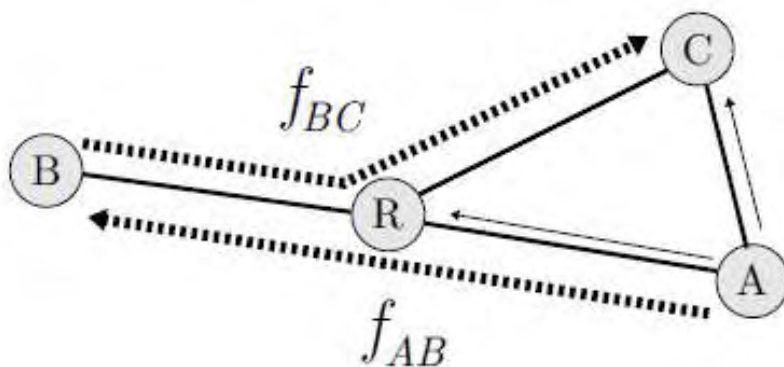
ΑΝΤΙΚΕΙΜΕΝΟ ΤΗΣ ΕΡΓΑΣΙΑΣ

Η εργασία έχει ως αντικείμενο την προσομοίωση της λειτουργίας ενός κόμβου ελέγχου σε ασύρματες επικοινωνίες, ο οποίος επιλέγει και στέλνει τα πακέτα κωδικοποιημένα ή και στην αρχική τους μορφή, σύμφωνα με κάποια κριτήρια που θα παρουσιαστούν στην συνέχεια.

Κύριο αντικείμενο της εργασίας ήταν η δημιουργία του κώδικα της προσομοίωσης στην C++, με σκοπό την εξοικονόμηση χρόνου, καθώς από ότι ξέρουμε η C είναι μια από τις πλέον γρήγορες γλώσσες προγραμματισμού.

ΕΙΣΑΓΩΓΗ

Σαν γενική ιδέα ο κόμβος ελέγχου αποφασίζει ποιο πακέτο θα στείλει, και αν θα το στείλει μόνο του ή κωδικοποιημένο με κάποιο άλλο. Κύριο ρόλο στην απόφασή του αυτή παίζει το αν οι κόμβοι προορισμού των πακέτων έχουν στην κατοχή τους κάποια από τα πακέτα αυτά. Αυτό μπορεί να γίνει είτε λαμβάνοντάς τα τυχαία την στιγμή της μετάδοσης από τον κόμβο πηγή προς τον κόμβο ελέγχου



Σχήμα 1 Όταν η Alice στέλνει προς τον κόμβο ελέγχου για να γίνει η μετάδοση προς τον Bob η Chloe ακούει το πακέτο και το αποθηκεύει ως κλειδί

είτε αν τα είχαν κάποτε στην κατοχή τους οπότε τα έχουν αποθηκεύσει. Την γνώση αυτή των κόμβων προορισμού την μαθαίνει ο κόμβος ελέγχου είτε μέσω κάποιων σημάτων αναγνώρισης που στέλνουν οι κόμβοι προορισμού είτε την υποθέτει βάση στατιστικών μετρήσεων.

Στην χορ κωδικοποίηση τα πακέτα προστίθενται με χορ στον κόμβο ελέγχου και στον κόμβο προορισμού κάθε πακέτο αποκωδικοποιείται αν ξέρουμε τα υπόλοιπα με τα οποία είχε γίνει η κωδικοποίηση στον κόμβο ελέγχου. Με αυτόν τον τρόπο έχουμε ένα κέρδος 4/3 εφόσον για κάθε τέσσερις αποστολές στην απλή μετάδοση με την κωδικοποίηση χρειάζονται τρεις. Το βέλτιστο κέρδος μπορεί να γίνει και 2 εφόσον συνδυάζονται απείρως πολλά downlinks.

Κάθε κόμβος αποθηκεύει όλα τα πακέτα άλλων μεταδόσεων που τυγχάνει να φθάσουν σε αυτόν. Παράλληλα, ο κόμβος ελέγχου πρέπει να ξέρει ποια πακέτα κλειδιά (όπως ονομάζουμε τα πακέτα άλλων μεταδόσεων που χρησιμεύουν για αποκωδικοποίηση) έχει ο κάθε κόμβος προορισμού. Το να τα μάθει ρητά από τους ίδιους τους κόμβους προορισμού έχει μεγάλη ακρίβεια και δεν έχουμε αναμεταδόσεις πακέτων που απέτυχαν να σταλθούν με την πρώτη, αλλά είναι μια διαδικασία αργή και με μεγάλο κόστος, οπότε αναζητήθηκε η λύση υποθέσεων βάση στατιστικών.

Για να βελτιώσουμε την γνώση του κόμβου ελέγχου σχετικά με τα πακέτα κλειδιά των τερματικών προορισμού χρησιμοποιούμε την γνώση που παρέχεται από την επιτυχία ή αποτυχία αποκωδικοποίησης των πακέτων στους κόμβους προορισμού.

Να αναφέρουμε, επίσης, ότι στην όλη διαδικασία τα τερματικά δεν επιτρέπεται να αναμεταδίδουν κωδικοποιημένα πακέτα, αλλά πρέπει πρώτα να τα αποκωδικοποιούν και μετά να γίνεται η αναμετάδοση. Επίσης δεν επιτρέπεται να αποθηκεύουν πακέτα σε κωδικοποιημένη μορφή, ενώ τέλος θεωρούμε ότι κατά τις μεταδόσεις των πακέτων δεν έχουμε απώλειες.

Προσεκτικότερη παρατήρηση της διαδικασίας μετάδοσης πακέτων

Θεωρούμε ότι οι μεταδόσεις πακέτων συμβολικά έχουν την μορφή $i: \sigma_i \rightarrow R \rightarrow \delta_i$ όπου $i \in F = \{1, 2, \dots, F\}$ είναι οι πιθανές ροές μετάδοσης, σ_i ο αντίστοιχος κόμβος πηγή, δ_i ο κόμβος προορισμού και R ο κόμβος ελέγχου.

Η μετάδοση των πακέτων μπορεί να χωριστεί σε δύο φάσεις. Στην uplink φάση όπου τα πακέτα από τον κόμβο πηγής φτάνουν στον κόμβο ελέγχου και στην downlink φάση όπου τα πακέτα από τον κόμβο ελέγχου φεύγουν για τους κόμβους προορισμού.

Uplink φάση

Σε αυτή την φάση τα πακέτα μεταδίδονται χωρίς κωδικοποίηση προς τον κόμβο ελέγχου όμως, μπορεί να ληφθούν και από κόμβους προορισμού. Τότε οι κόμβοι προορισμού τα αποθηκεύουν και φυλάσσονται για μελλοντική αποκωδικοποίηση. Όμως τα πακέτα αποθηκεύονται και από τους ίδιους τους κόμβους πηγής για την περίπτωση που κάποιος κόμβος πηγής χρησιμοποιηθεί στο μέλλον ως κόμβος προορισμού.

Τα πακέτα που φτάνουν στον κόμβο ελέγχου αποθηκεύονται ως P_k^i . Όπου k είναι ο χρόνος που κατέφθασαν και i ο κόμβος πηγή που τα έστειλε.

Κάθε πακέτο στον κόμβο ελέγχου σχετίζεται με ένα διάνυσμα S_k^i που έχει τόσα στοιχεία όσες είναι οι ροές της μετάδοσης και κάθε στοιχείο $S_k^i(j)$ παίρνει την τιμή 1 αν το πακέτο ροής i έχει ακουστεί από τον αντίστοιχο κόμβο προορισμού j και 0 αν δεν έχει ακουστεί. Για παράδειγμα, αν οι κόμβοι είναι δύο, το διάνυσμα θα έχει τη μορφή: $[1\ 0]$ ή $[1\ 1]$ ή $[0\ 1]$ ή $[1\ 1]$.

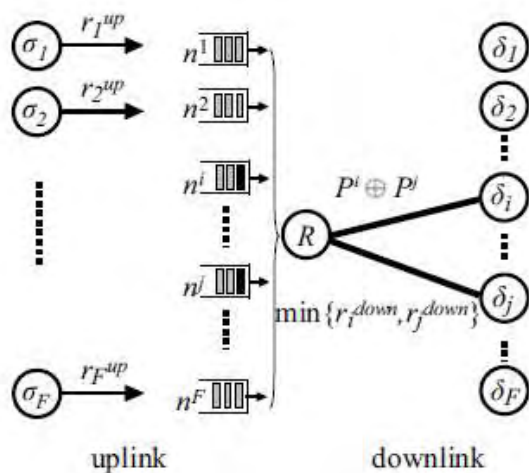
Ο κόμβος ελέγχου μαθαίνει για αυτό το διάνυσμα από τους κόμβους προορισμού ή υποθέτει για αυτό το διάνυσμα βάση στατιστικών.

Κατά τη **ντετερμινιστική γνώση** στέλνονται πακέτα ελέγχου με αποτέλεσμα πολύπλοκη λειτουργία.

Κατά την **στατιστική γνώση** η γνώση αποκτάται μέσα από έρευνα σε βάθος χρόνου και παίρνουμε την πιθανότητα ο κόμβος j να ακούσει τον i q_{ij} . Η γνώση αυτή μπορεί να βελτιωθεί κατά την διάρκεια με την βοήθεια της ανατροφοδότησης κάθε φορά που ένα πακέτο αποτυγχάνει να αποκωδικοποιηθεί.

Downlink φάση

Τα πακέτα στέλνονται είτε μόνα τους σε μη κωδικοποιημένη μορφή είτε μαζί με άλλα με χωρ κωδικοποίηση. Στην παρούσα φάση εξετάζεται μόνο η περίπτωση να κωδικοποιούνται πακέτα ανά ζεύγη ή να φεύγουν μόνα τους. Δηλαδή δεν γίνεται κωδικοποίηση με πάνω από δύο πακέτα.



Σχήμα 2 uplink και downlink φάση

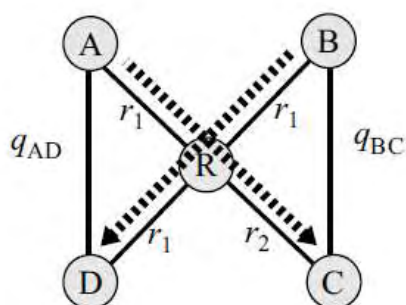
Feedback φάση

Ο κόμβος προορισμού i προσπαθεί να αποκωδικοποιήσει το πακέτο P_k^i και αν τα καταφέρει το πακέτο διαγράφεται από την αποθήκη του κόμβου ελέγχου. Κάτι τέτοιο γίνεται πάντοτε στην περίπτωση αποστολής ενός πακέτου ενώ στην περίπτωση αποστολής δύο κωδικοποιημένων μεταξύ τους πακέτων υπάρχει περίπτωση αποτυχίας αποκωδικοποίησης ενός από τα δύο ή και των δύο.

Σε αυτή την περίπτωση λαμβάνουμε επιπλέον πληροφορία .Συγκεκριμένα:

α) αν δεν αποκωδικοποιηθεί κανένα από τα δύο χορ κωδικοποιημένα πακέτα, σημαίνει ότι και τα δύο δεν έχουν ακουστεί από τον κόμβο προορισμού του άλλου

β) αν μόνο ένα από τα δύο δεν αποκωδικοποιήθηκε, σημαίνει ότι ο κόμβος προορισμού του δεν έχει ακούσει το άλλο πακέτο. Όμως το ίδιο έχει ακουστεί από τον κόμβο προορισμού του δεύτερου που για αυτό τον λόγο και αποκωδικοποιήθηκε .



Σχήμα 3 Ένα παράδειγμα με δύο ροές A->C, B->D. Το D ακούει αυτό που στέλνει το A με πιθανότητα q_{AD} , ενώ το C ακούει αυτό που στέλνει το B με πιθανότητα q_{BC}

Η αξία του feedback

Ας θεωρήσουμε στατιστική αντιμετώπιση ενός συστήματος με δύο κόμβους πηγές A,B και δύο κόμβους προορισμού C,D αντίστοιχα. Δηλαδή: A->C (από τον σταθμό A στέλνουμε πακέτα προς το C) και B->D με πολιτική κωδικοποίησης δύο πακέτων αν το άθροισμα των πιθανοτήτων $q_{AD}+q_{BC} \geq 1$ (όπου q_{AD} η πιθανότητα το D να ακούσει το A) που σημαίνει ότι θα αποκωδικοποιείται τουλάχιστον ένα πακέτο ανά μέσο όρο.

Αν πάρουμε την περίπτωση στατιστικά η παραπάνω σχέση να ισχύει, τότε θα σταλθεί το χοι άθροισμα αυτών των δύο. Αν, όμως, στην πραγματικότητα δεν έχουν ακουστεί και δεν πάρουν σήμα αναγνώρισης θα ξανασταλούν με τον ίδιο τρόπο σύμφωνα με την πολιτική κωδικοποίησης. Επομένως, αυτή η διαδικασία θα συμβαίνει συνέχεια μέχρι να προκύψει αδιέξοδο.

Για την αντιμετώπιση του παραπάνω προβλήματος το feedback πληροφορεί τον κόμβο ελέγχου τι πραγματικά έγινε για τα πακέτα που δεν έχουν αποκωδικοποιηθεί και πρέπει να ξανασταλούν. Έτσι, στην συγκεκριμένη περίπτωση αν χρησιμοποιήσουμε feedback, θα σταλούν την επόμενη φορά μόνα τους.

Η απόφαση του πότε θα σταλούν κωδικοποιημένα τα πακέτα και πότε όχι επηρεάζει το αν οι ουρές που είναι αποθηκευμένα τα πακέτα στον κόμβο ελέγχου θα είναι σταθερές ή όχι. Με τον όρο σταθερές (stable) εννοούμε το αν θα μένουν σε χαμηλά μεγέθη πακέτων καθώς ο χρόνος λειτουργίας του συστήματος τείνει στο άπειρο.

Αλγόριθμος για ντετερμινιστική γνώση

Στην περίπτωση της ντετερμινιστικής γνώσης το feedback δεν προσφέρει επιπλέον γνώση. Ο κόμβος ελέγχου έχει 2^{F-1} ουρές για κάθε τερματικό πηγής. Με την άφιξη τους στον κόμβο ελέγχου τα πακέτα εισέρχονται ανάλογα με το διάνυσμα s που αναφέρθηκε προηγουμένως στην αντίστοιχη ουρά. Στο διάνυσμα S κατά σύμβαση θεωρούμε $S_K^i(i)=0$. Οπότε για σύστημα με δύο ροές έχουμε μόνο δύο διαφορετικά πιθανά διανύσματα S ανά ροή και άρα δύο ουρές ανά ροή (2^{F-1}).

Το ποιο ή ποια πακέτα θα επιλέξει ο κόμβος ελέγχου για να στείλει το κάνει με βάση το ποια πακέτα μεγιστοποιούν την εξίσωση

α) $Z(I)=x_{s1}^i * \mu_{n s1}^i(I)$ για ένα πακέτο και

β) $Z(I)=x_{s1}^i * \mu_{n s1}^i(I)+ x_{s2}^j * \mu_{n s2}^j(I)$ για δύο πακέτα.

Όπου $\mu_{n s1}^i(I)$ ο ρυθμός εξυπηρέτησης που είναι ίσος με τον ρυθμό r_i^{down} σε περίπτωση που στέλνεται ένα πακέτο.

Σε περίπτωση που στέλνονται δύο πακέτα ο ρυθμός εξυπηρέτησης είναι
α) $\mu_{ns1}^i(I)=0$, αν $s_2(i)=0$ (δηλαδή το δεύτερο πακέτο δεν έχει ακουστεί από το
τερματικό προορισμού του πρώτου οπότε δεν γίνεται αποκωδικοποίηση)
β) $\min\{r_i^{\text{down}}, r_j^{\text{down}}\}$ σε περίπτωση που $s_2(i)=1$.

Και αντίστοιχα για το $\mu_{ns2}^j(I)$

Το x_{s1}^i είναι ο αριθμός των πακέτων που έχει κάθε μια ουρά για κάθε ένα
τερματικό πηγή.

Σε αυτή την ντετερμινιστική περίπτωση αναμένεται να μην υπάρχουν πολλά
πακέτα στις ουρές. Όμως, δεν μπορεί να λειτουργήσει εύκολα για πάνω από
20 ροές.

Συστήματα με στατιστική γνώση

The virtual network mechanism

Ο virtual network mechanism χρησιμοποιείται για να ελέγξουμε τις
αλλαγές στη γνώση που προκαλούνται από το feedback.

Στην περίπτωση που έχω 2 τερματικά πηγές και χορ κωδικοποίηση ανά
ζεύγη ακολουθείται η παρακάτω μέθοδος. Ονομάζουμε τρεις καταστάσεις:

1) Unknown state(u)

Σε αυτή την κατάσταση δεν γνωρίζουμε το πακέτο που εξετάζουμε P_K^i που
έχει πηγή το i αν έχει ληφθεί από το j . Η πιθανότητα να έχει ληφθεί είναι q_{ij} .
Το ίδιο ισχύει και αν εξετάζουμε πακέτο με πηγή το j .

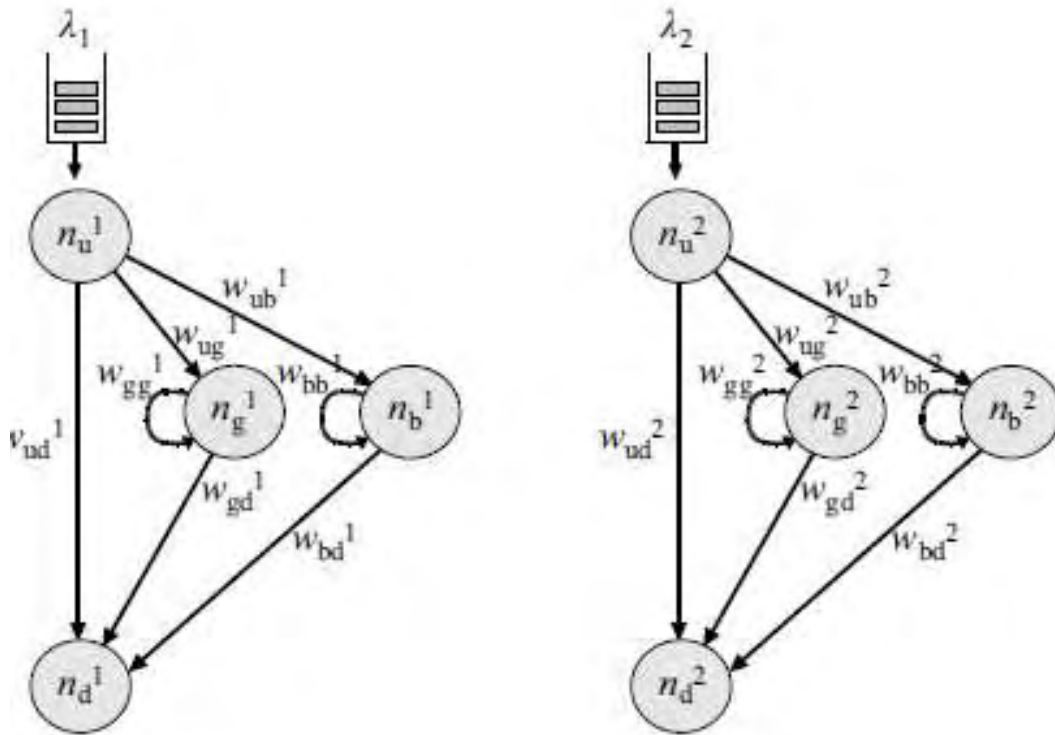
2) Good state(g)

Σε αυτή την περίπτωση ξέρουμε ότι το πακέτο P_K^i έχει ληφθεί και από το
άλλο τερματικό j . Η πιθανότητα να έχει ληφθεί είναι 1 δηλαδή.

3) Bad state(b)

Σε αυτή την περίπτωση ξέρουμε ότι το πακέτο P_K^i δεν έχει ληφθεί και από το
άλλο τερματικό j . Η πιθανότητα να έχει ληφθεί είναι 0 δηλαδή.

Έτσι, για κάθε ροή πακέτου i δημιουργούμε ένα υποδίκτυο που έχει έναν
κόμβο για κάθε κατάσταση και έναν κόμβο για τον προορισμό
 $N_i=\{n_u^i, n_g^i, n_b^i, n_d^i\}$.



Σχήμα 4 Το virtual network στην περίπτωση δύο ροών

Κάθε κατάσταση συνδέεται με μία ουρά όπου και πάνε τα πακέτα ανάλογα με την κατάστασή τους (ο κόμβος n_u^i έχει την ουρά x_u^i). Ο κόμβος προορισμού έχει άδεια ουρά, επειδή όταν φθάνουν σε αυτόν τα πακέτα φεύγουν από το σύστημα. Τα νεοεισερχόμενα πακέτα μπαίνουν στον κόμβο n_u αλλά δεν ξέρουμε ποιος θα είναι ο κόμβος που θα καταλήξουν μετά από μια απόπειρα αποστολής. Αυτό εξαρτάται από τις πιθανότητες $w_{km}^i(l)$ να πάει ένα πακέτο από τον κόμβο k στον m . Το άθροισμα όλων των πιθανοτήτων μίας μετάβασης από έναν κόμβο σε όλους τους άλλους είναι 1(μονάδα). $\sum_m w_{km}^i(l) = 1$ όπου m είναι οι γειτονικοί κόμβοι του k .

I	w_{ud}^1	w_{ug}^1	w_{ub}^1	w_{gd}^1	w_{gg}^1	w_{bd}^1	w_{bb}^1
$\{n_u^1, n_u^2\}$	q_{21}	$(1 - q_{21})q_{12}$	$(1 - q_{21})(1 - q_{12})$	0	0	0	0
$\{n_u^1, n_g^2\}$	1	0	0	0	0	0	0
$\{n_u^1, n_b^2\}$	0	q_{12}	$1 - q_{12}$	0	0	0	0
$\{n_g^1, n_u^2\}$	0	0	0	q_{12}	$1 - q_{12}$	0	0
$\{n_g^1, n_g^2\}$	0	0	0	1	0	0	0
$\{n_g^1, n_b^2\}$	0	0	0	0	1	0	0
$\{n_b^1, n_u^2\}$	0	0	0	0	0	q_{12}	$1 - q_{12}$
$\{n_b^1, n_g^2\}$	0	0	0	0	0	1	0
$\{n_b^1, n_b^2\}$	0	0	0	0	0	0	1

TABLE I
TRANSITION WEIGHTS IN THE SUBNETWORK \mathcal{G}_1 FOR ALL PAIR CONTROLS.

Για παράδειγμα, το πακέτο ροής 1 που είναι στον κόμβο u και στέλνεται κωδικοποιημένο με πακέτο ροής 2, κόμβου $u \{n_u^1, n_u^2\}$ το που θα πάει εξαρτάται από τις πιθανότητες. Αν το δεύτερο έχει ακουστεί (πιθανότητα q_{21}) το πρώτο πακέτο έχει τα κλειδιά και αποκωδικοποιείται, οπότε πάει στον κόμβο d με πιθανότητα q_{21} .

Αν το δεύτερο δεν έχει ακουστεί (πιθανότητα $1 - q_{21}$), το πακέτο 1 μένει στο σύστημα. Ο κόμβος που θα πάει εξαρτάται και από το ίδιο και την επιρροή του στο αν αποκωδικοποιήθηκε σωστά το πακέτο 2. Οπότε η πιθανότητα να πάει στο g , επηρεάζεται επίσης από την πιθανότητα να έχει ακουστεί από το 2, οπότε το πακέτο ροής 2 να έχει αποκωδικοποιηθεί σωστά. Η πιθανότητα αυτή είναι q_{12} . Αν πολλαπλασιάσουμε επί την πιθανότητα που αναφέρθηκε προηγουμένως, δηλαδή το πακέτο 1 να έχει μείνει στο σύστημα ($1 - q_{21}$), παίρνουμε την πιθανότητα να πάει σε g .

Δηλαδή $q_{12} * (1 - q_{21})$.

Αντίστοιχα η πιθανότητα να πάει στον κόμβο b είναι $(1 - q_{12}) * (1 - q_{21})$. Στο σύνολο αυτές οι τρεις πιθανότητες κάνουν ένα.

Σε περίπτωση που το πακέτο είναι ήδη σε ένα κόμβο g ή b δεν υπάρχει περίπτωση να πάει σε άλλον. Ή θα μείνει στον ίδιο κόμβο ή θα φύγει από το σύστημα πηγαίνοντας στο d .

Ανάλογα επεκτείνεται ο μηχανισμός και για περισσότερες ροές, αν και υπάρχει μία μικρή διαφοροποίηση στην λογική λόγω μεγαλύτερης πολυπλοκότητας. Στην περίπτωση αυτή αντί για 4 κόμβους θα έχουμε 3^{F-1} κόμβους συν έναν ακόμη ως κόμβο προορισμού για κάθε ροή. Οι κόμβοι αυτοί προκύπτουν από ένα διάνυσμα c , F στοιχείων $\{u, g, b\}$ με $c(i) = b$. Οπότε και αυτό το στοιχείο διανύσματος παραλείπεται στο παρακάτω σχήμα για την ροή 1

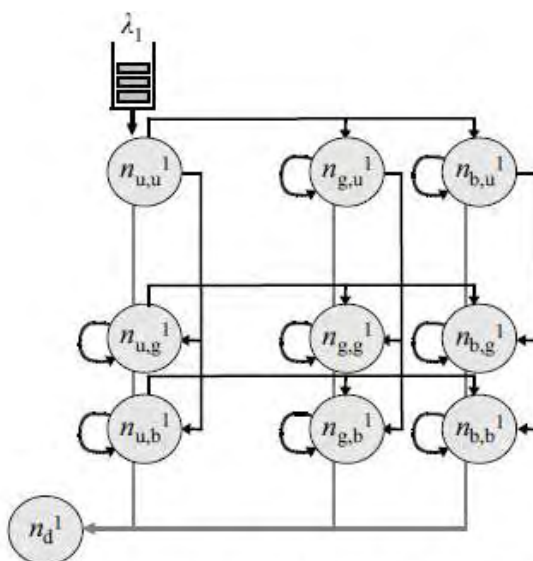


Fig. 5. The virtual subnetwork of flow 1 for the case of 3 flows.

Αλγόριθμος για στατιστική γνώση

Το ποιο ή ποια πακέτα θα επιλέξει για να στείλει, το κάνει με βάση το ποια πακέτα μεγιστοποιούν την εξίσωση:

$Z(I) = x_c^i * \mu_{n_c^i}(I)$ για ένα πακέτο με

$$\mu_k(I) = \begin{cases} r_i^{\text{down}} & \text{if } k = j \\ 0 & \text{otherwise} \end{cases}$$

r_i^{down} ο μέγιστος αριθμός πακέτων που μπορούν να σταλθούν από τον κόμβο ελέγχου προς τον κόμβο προορισμού i , και j ο κόμβος κατάστασης που βρισκόμαστε.

και για δύο πακέτα βρίσκουμε τα βάρη

$$z_i(I) = \max \left\{ x_{c_1}^i - \sum_{k \in \mathcal{N}^{n_{c_1}^i}} w_{(n_{c_1}^i, k)}(I) x_k^i, 0 \right\}$$

$$z_j(I) = \max \left\{ x_{c_2}^j - \sum_{k \in \mathcal{N}^{n_{c_2}^j}} w_{(n_{c_2}^j, k)}(I) x_k^j, 0 \right\}$$

Και επιλέγουμε τα πακέτα που μεγιστοποιούν την εξίσωση

$$Z(I) = z_i(I) \mu_{n_{c_1}^i}(I) + z_j(I) \mu_{n_{c_2}^j}(I).$$

Για δύο πακέτα

$$\mu_k(I) = \begin{cases} \min\{r_{i_1}^{\text{down}}, r_{i_2}^{\text{down}}\} & \text{if } k = j_1 \text{ or } k = j_2 \\ 0 & \text{otherwise} \end{cases}.$$

Η κωδικοποίηση πάνω από δύο πακέτων μαζί με χορ αυξάνει κατά πολύ την πολυπλοκότητα και προς το παρόν είναι δύσκολο να υλοποιηθεί κάτι τέτοιο.

Τον συγκεκριμένο αλγόριθμο τον χρησιμοποιούμε στον κώδικά μας όπως θα δούμε και στην συνέχεια στην προσομοίωση `main_qba` που είναι στοχαστικής γνώσης, ενώ τον αλγόριθμο που είδαμε προηγουμένως για ντετερμινιστική γνώση τον χρησιμοποιούμε στην προσομοίωση `main_qba_det`.

Μεταφορά στη C++

Καταρχήν επιλέξαμε να χρησιμοποιήσουμε στο πρόγραμμα μας τους βέκτορες (vectors) που είναι ο τρόπος που χειρίζεται η C++ οργανωμένα και τυποποιημένα τους δυναμικούς πίνακες που η συγκεκριμένη προσομοίωσή μας βασίζεται πολύ σε υπολογισμούς δυναμικών πινάκων.

Οι απλοί πίνακες στην C++ είναι δύσχρηστοι αν έχουμε να κάνουμε με μεταβαλλόμενους(δυναμικούς) πίνακες. Τα προβλήματα που προκύπτουν έχουν να κάνουν με το ότι πρέπει κάπου να κρατάμε το μέγεθος των πινάκων, ώστε να γίνεται έλεγχος μήπως έχουμε βγει έξω από τα όρια και χειριζόμαστε μνήμη που δεν ανήκει στον πίνακα.

Και αυτό γιατί οι απλοί πίνακες δεν έχουν τρόπο αυτόματου ελέγχου και υπολογισμού του μεγέθους τους και του αν έχουμε ξεπεράσει τα όρια της μνήμης που καταλαμβάνουν με τους υπολογισμούς μας.

Έτσι, πρέπει να δημιουργούμε μεγάλους πίνακες ώστε να μην βγαίνουμε έξω από τα όρια τους και παράλληλα να κρατάμε τα στοιχεία που έχουν πάρει τιμές από το πρόγραμμα μας. Δηλαδή την πραγματικά χρησιμοποιούμενη μνήμη του πίνακα από την μεγάλη που έχει κρατηθεί.

Μια λύση που εφαρμόσαμε είναι να δημιουργήσουμε κλάσεις που να έχουν σαν μέλη τους πίνακες, το μέγεθός τους αλλά και τον αριθμό των χρησιμοποιούμενων στοιχείων τους. Αυτό προσπαθήσαμε στη δεύτερη υλοποίηση.

Όμως, δεν καταφέραμε να πιάσουμε μεγαλύτερη ταχύτητα από τους βέκτορες.

Ίσως κάτι τέτοιο να είναι και πολύ δύσκολα εφικτό εφόσον ουσιαστικά με τις κλάσεις πινάκων ουσιαστικά προσπαθούμε να δημιουργήσουμε παρόμοια λειτουργία με αυτή του βέκτορα .

Πηγές αναφέρουν ότι οι βέκτορες είναι σχεδόν το ίδιο αποδοτικοί όπως οι απλοί πίνακες. Έτσι αποδείχτηκε και από την δική μας προσπάθεια σύγκρισης αλλά δεν μπορούμε να είμαστε βέβαιοι για αυτό.

Χαρακτηριστικό είναι όμως ότι σε απλά loops γεμίσματος πίνακα και vector με πολλά στοιχεία οι απλοί πίνακες γεμίζουν αρκετά πιο γρήγορα.

Συγκεκριμένα, παρατηρήθηκε μια διαφορά της τάξης των 15 φορές ταχύτερο γέμισμα στον απλό πίνακα!! Όμως υπάρχει το ενδεχόμενο αυτό να γίνεται μόνο σε απλές διαδικασίες επειδή με κάποιο τρόπο ο compiler

μπορεί να καταλάβει τι κάνει το loop και να αποφύγει κάποιες ενέργειες πάνω στον πίνακα επειδή έχει απλή δομή. Κάτι τέτοιο δεν μπορεί να κάνει με τον vector. Σε σύνθετες διαδικασίες όμως δεν φαίνεται να έχουν διαφορά.

Ένας πίνακας ορίζεται ως εξής
`int sample[10];` Είναι ένα παράδειγμα.

Vectors(διανύσματα)

Ανήκουν στην καθιερωμένη βιβλιοθήκη προτύπων(standard template library,STL) και έχουν κατά πάσα πιθανότητα την μεγαλύτερη χρήση από όλα τα αντικείμενα της βιβλιοθήκης STL(αποδέκτες).

Δημιουργήθηκαν για να καλύψουν την ανάγκη για δυναμικούς πίνακες και έχουν την ίδια έννοια με τους πίνακες ενώ μπορεί να χρησιμοποιηθεί και η καθιερωμένη σημειογραφία των αριθμοδεικτών.

Οι δυναμικοί πίνακες μίας, δύο και τριών διαστάσεων ορίζονται ως
`Vector<int> v;`

`Vector<vector<int>> v2;`

`Vector<vector<vector<int>>> v3;`

Μπορούμε ανά πάσα στιγμή να βρούμε το μέγεθός τους με την `size`
`v.size();`

Στους vectors ο αντίστοιχος πόιντερ που χρησιμοποιείται στην υπόλοιπη γλώσσα είναι ο `iterator`.

`Vector<int>::iterator p=v.begin();`

`cout<<*p;`

Ενώ η διαγραφή στοιχείων γίνεται με την `erase` που χρησιμοποιεί `iterators`
`v.erase(p,p+10)` διαγράφει δέκα στοιχεία μετά τον `Iterator p`

Υλοποίηση του αλγορίθμου

Η όλη διαδικασία ξεκινάει από την συνάρτηση main που έχει μπει στο αρχείο arrive όπου βρίσκεται και η συνάρτηση arrive.

Γενικά κάθε συνάρτηση έχει μπει σε αρχείο με το ίδιο όνομα εκτός από την main που έχει μπει στο αρχείο arrive

Main

```
int main()
{
    int time_slots=1000;
    int figure_points=8;
    double min_lambda=0.4;
    double max_lambda=0.7;

    double q12=0.5;
    double q21=0.8;
    double alpha=1;
    vector<double> l1,l2,QS_qba,QS_qba_det,QS_cope_det,QS_cope_sto,QS_cope_adv;
    for(int i=0;i<figure_points;i++)
    {
        l1.push_back(min_lambda+i*(max_lambda-min_lambda)/figure_points);
    }
    clock_t t1,t2;
    t1=clock();
    for (int i=0;i<figure_points;i++)
    {
        cout<<"i "<<i<<endl;
        l2.push_back(l1[i]*alpha);

        QS_qba.push_back(main_qba(l1[i],l2[i],q12,q21,time_slots));
        cout<<endl<<QS_qba[i]<<endl;

        QS_qba_det.push_back(main_qba_det(l1[i],l2[i],q12,q21,time_slots));
        cout<<endl<<QS_qba_det[i]<<endl;

        QS_cope_det.push_back(main_cope_det(l1[i],l2[i],q12,q21,time_slots));
        cout<<endl<<QS_cope_det[i]<<endl;

        QS_cope_sto.push_back(main_cope_sto(l1[i],l2[i],q12,q21,time_slots));
        cout<<endl<<QS_cope_sto[i]<<endl;

        QS_cope_adv.push_back(main_cope_adv(l1[i],l2[i],q12,q21,time_slots));
        cout<<endl<<QS_cope_adv[i]<<endl;

    }
    cout<<endl<<"lamda "<<endl;print(l1);
    cout<<endl<<"QS_qba "<<endl;print(QS_qba);
    cout<<endl<<"QS_qba_det "<<endl;print(QS_qba_det);
    cout<<endl<<"QS_cope_det "<<endl;print(QS_cope_det);
    cout<<endl<<"QS_cope_sto "<<endl;print(QS_cope_sto);
    cout<<endl<<"QS_cope_adv "<<endl;print(QS_cope_adv);
    t2=clock();
    double dif = (double) (t2-t1)/CLOCKS_PER_SEC ;
    cout<<"dif "<<dif;
    int M;
    cin>>M;
}
```

Δηλώνουμε τα διανύσματα l1,l2 που παίρνουν μέσα τιμές για τους ρυθμούς άφιξης των πακέτων 1,2 από τα τερματικά πηγές 1 και 2 αντίστοιχα

Κατόπιν για figure_points επαναλήψεις καλούμε τις ρουτίνες main_qba, main_qba_det, main_cope_det, main_cope_sto, main_cope_adv

Αυτές οι ρουτίνες κάνουν όλη την προσωμοίωση για διαφορες περιπτώσεις ντετερμινιστικής η στοχαστικής γνώσης. Το αποτελέσματα που γυρνάνε(μέσος αριθμός πακέτων στην ουρά) μπαίνει στα διανύσματα QS_qba,QS_qba_det,QS_cope_det,QS_cope_sto,QS_cope_adv ώστε να μπορούμε να τα έχουμε συγκεντρωμένα κάπου για να τα συγκρίνουμε.Στο τέλος της main και άρα όλης της διαδικασίας προσωμοίωσης εκτυπώνουμε τα αποτέλεσμα καθώς και τον χρόνο που πήρε για να τρεξει η προσομοίωση.

Arrive

```
void arrive(vector<vector<int>> &Mem_in,vector<int> A,vector<vector<double>> &P)
{
    int M=2;
    for (int i=0;i<M;i++)
    {
        for (int k=0;k<A[i];k++)
        {
            vector<int> voithima;
            voithima.push_back(find_Column_Max(Mem_in,0)+1);

            voithima.push_back(i+1);
            voithima.push_back(1);
            change_row_elements2(voithima,3,M+2,P[i]);
            Mem_in.push_back(voithima);
        }
    }
    int asxeto;
}
```

Η συνάρτηση arrive είναι αυτή που καλείται για όλες τις διαδικασίες προσομοίωσης για κάθε μια στιγμή από τις time_slots που τρέχει κάθε ρουτίνα προσομοίωσης. Δέχεται ως είσοδο την κεντρική μνήμη Mem_in που υπάρχει στον κόμβο ελέγχου και την αλλάζει σύμφωνα με το πόσα πακέτα έχουν έρθει από τον κάθε κόμβο πηγή. Σαν είσοδο δέχεται και τον πίνακα πιθανοτήτων P(στην προκειμένη περίπτωση διάνυσμα δύο διαστάσεων αλλά για ευκολία θα το ονομάζουμε και πίνακα γιατί έχει την ίδια έννοια). Ο P έχει σαν στοιχεία στην i γραμμή και j στήλη την πιθανότητα το j τερματικό να ακούσει το i και είναι στην περίπτωση μας δύο διαστάσεων.

Η arrive καλεί την συνάρτηση find_Column_Max για να βρει τον μεγαλύτερο αριθμό πακέτου που υπάρχει στο σύστημα ώστε να περάσει πακέτο με αριθμό ένα μεγαλύτερο από τα μεγαλύτερο μέγιστο. Αυτό το περνάει στην πρώτη στήλη. Στην δεύτερη περνάει τον αριθμό του πακέτου δηλαδή από ποια πηγή προέρχεται. Στην Τρίτη στήλη περνάει την κατάσταση του πακέτου που ξεκινάει πάντα όπως είδαμε από την κατάσταση u(που την περνάμε ως 1). Όταν αργότερα σε άλλες συναρτήσεις

το πακέτο δεν καταφέρει να αποκωδικοποιηθεί η κατάσταση του αλλάζει σε $g(2)$ ή $b(3)$. Στις στήλες 4 και 5 περνάμε το αν τελικά έχει εισακουστεί το πακέτο ή όχι δηλαδή το διάνυσμα s . Περνάμε 1 αν έχει εισακουστεί και 0 αν δεν έχει εισακουστεί. Από σύμβαση περνάμε τα στοιχεία με ίδιο αριθμό γραμμής και στήλης του s με την τιμή 1 ενώ τα άλλα είτε θα είναι μηδέν είτε ένα. Η διαδικασία δημιουργίας του διανύσματος s γίνεται από την `change_row_elements2`

```
void change_row_elements2(vector<int> &diplospinakas, int first_column, int
last_column, vector<double> &P)
{
    int Pcolumn=0;
    for(int j=first_column; j<=last_column; j++)
    {
        diplospinakas.push_back(get_resultof_probability(P[Pcolumn]));
        Pcolumn++;
    }
}

int get_resultof_probability(double prob)
{
    double anumber_0_1=(rand()%1000)/1000.0;
    //cout<<"anumber_0_1"<<" "<<anumber_0_1<<" "<<prob<<endl;
    if(anumber_0_1<prob)
    {
        //cout<<1<<endl;
        return 1;
    }
    else
    {
        //cout<<0<<endl;
        return 0;
    }
}
```

Ένα παράδειγμα για το πώς τρέχει η ρουτίνα arrive

A=[1 0]

i = 1

k = 1

Mem_in =

33 2 1 1 1

34 2 1 1 1

35 2 1 1 1

36 2 1 1 1

37 2 1 1 1

39 2 1 1 1

40 1 1 1 0

0 0 0 0 0

Mem_in =

33 2 1 1 1

34 2 1 1 1

35 2 1 1 1

36 2 1 1 1

37 2 1 1 1

39 2 1 1 1

40 1 1 1 0

41 0 0 0 0

Mem_in =

33 2 1 1 1

34 2 1 1 1

35 2 1 1 1

36 2 1 1 1

37 2 1 1 1

39 2 1 1 1

40 1 1 1 0

41 1 0 0 0

Mem_in =

33 2 1 1 1

34 2 1 1 1

35 2 1 1 1

36 2 1 1 1

37 2 1 1 1

39 2 1 1 1

40 1 1 1 0

41 1 1 0 0

Mem_in =

33 2 1 1 1

34 2 1 1 1

35 2 1 1 1

36 2 1 1 1

37 2 1 1 1

39 2 1 1 1

40 1 1 1 0

41 1 1 1 1

bi2dec, dec2tri, tri2dec, Qmatrix

Είναι βοηθητικές συναρτήσεις που καλούνται κατά τη διάρκεια της προσομοίωσης και κάνουν απλές λειτουργίες που είναι προφανείς και από το όνομά τους.

Η bi2dec μετατρέπει έναν δυαδικό αριθμό σε δεκαδικό

```
int bi2dec(vector<int> &in_vec )
{
    int number=0;
    for (int n=in_vec.size();n>=1;n--)
    {
        number=number+in_vec[in_vec.size()-n]*pow(2.0, (n-1));
    }
    return number;
}
```

Η dec2tri μετατρέπει έναν δεκαδικό αριθμό σε τριαδικό

```
vector<int> dec2tri(int number )
{
    vector<int> out_vec;
    int rem=number;
    int n=floor(log((double) number)/log(3.0));
    while(n>-1)
    {
        out_vec.push_back(floor(rem/pow(3.0,n)));
        rem=rem-pow(3.0,n)*out_vec[out_vec.size()-1];
        n=n-1;
    }
    if(number<=0) out_vec.push_back(0);
    return out_vec;
}
```

Η tri2dec μετατρέπει έναν τριαδικό σε δεκαδικό

```
int tri2dec(vector<int> &in_vec )
{
    int number=0;
    for (int n=in_vec.size();n>=1;n--)
    {
        number=number+in_vec[in_vec.size()-n]*pow(3.0, (n-1));
    }
    return number;
}
```

Η Qmatrix δημιουργεί τον πίνακα P που περιέχει τις πιθανότητες να ακουστεί ένα τερματικό πηγή από ένα τερματικό προορισμού.

```
vector<vector<double>> Qmatrix(double p12, double p21, int M )
{
    vector<vector<double>> P;
    for(int i=0; i<M; i++)
    {
        P.push_back(vector<double>());
        for(int j=0; j<M; j++)
        {
            if(i==j) P[i].push_back(1);
            else P[i].push_back(0);
        }
    }
    switch(M)
    {
        case 2:
            p12=0.5;
            p21=0.8;
            P[0][1]=p12;
            P[1][0]=p21;
            break;
        case 3:
            P[0][1]=0.8308;
            P[0][2]=0.9172;
            P[1][0]=0.4733;
            P[1][2]=0.2858;
            P[2][0]=0.9517;
            P[2][1]=0.9497;
            break;
        default:
            cout<<"so far we have built probabilities only for M=3";
    }
    return P;
}
```

Το P στην περίπτωση μας και σε όλη την διάρκεια του προγράμματος έχει την τιμή

P =

1.0000	0.5000
0.8000	1.0000

Που την παίρνει κατά την κλήση της Qmatrix με ορίσματα σταθερά q12 =0.5, q21=0.8 και M=2 όπως περνάει από την main.

Προσομοίωση main_qba

Συνάρτηση main_qba

```
double main_qba(double Lambda_1, double Lambda_2, double q12, double q21, int
time_slots)
{
vector<vector<vector<int>>> B;
int M=2;
int N=pow(3.0, (M-1));
int max_comb_size=2;
vector<vector<int>> Q(M, vector<int>(N));

vector<double> Lambda_i(M);
Lambda_i[0]=Lambda_1;
Lambda_i[1]=Lambda_2;
vector<vector<double>> P=Qmatrix( q12,q21,M );
vector<vector<int>> Memory;
Memory.reserve(200);
int metritis=0;
vector<int> A;

for (int t=0;t<time_slots;t++)
{
    vector<vector<int>> I=scheduler(Q,max_comb_size,P);

    transmit(I,Q,Memory);

    A=random_poisson ( Lambda_i);

    arrive(Memory,A,P);

    for(int i=0;i<Q.size();i++) Q[i][0]=Q[i][0]+A[i];

    B.push_back(Q);

}

int sumb=0;
for(int i=0;i<B.size();i++)
{
    for(int j=0;j<B[i].size();j++)
    {
        for(int k=0;k<B[i][j].size();k++)
            sumb=sumb+B[i][j][k];
    }
}
double queue_size=(double) sumb/time_slots;
return queue_size;
}
```

Παίρνει ως ορίσματα τα Λ_1 , Λ_2 , q_{12} , q_{21} , $time_slots$. Όπου Λ_1 και Λ_2 είναι ο ρυθμός άφιξης πακέτων από τα τερματικά 1 και 2 αντίστοιχα.

Τα q_{12} και q_{21} είναι η πιθανότητα να ακούσουν τα τερματικά προορισμός αυτά που έστειλαν τα τερματικά πηγές.

Και $time_slots$ είναι ο αριθμός των επαναλήψεων που θα τρέξει η διαδικασία δηλαδή είναι μια προσωμοίωση του χρόνου.

Αφού μπει στο for loop και τρέξει για $figure_times$ φορές θα προχωρήσει και θα υπολογίσει τον μέσο όρο του αριθμού των πακέτων στην ουρά. Αυτό γίνεται αφού ο αριθμός των πακέτων κάθε κατάστασης για τα πακέτα κάθε ροής σε κάθε χρονική στιγμή (το διπλό διάνυσμα Q) αποθηκεύεται στον διάνυσμα τριών διαστάσεων B . Από εκεί μπορούμε να υπολογίσουμε τον μέσο όρο που μας ενδιαφέρει.

Όσον αφορά τις συναρτήσεις που καλούνται κατά την διάρκεια του for loop, εκτός της `arrive` η οποία έχει αναφερθεί.

Συνάρτηση scheduler

```
vector<vector<int>>> scheduler (const vector<vector<int>>> &Q,int max_comb_size,const
vector<vector<double>>> &P )
{
    int N=Q[0].size();
    int M=Q.size();
    vector<int> Ymax,Xmax;
    Ymax.reserve(max_comb_size);
    Xmax.reserve(max_comb_size);
    vector<vector<vector<int>>>> Qmax;
    Qmax.reserve(max_comb_size);
    for(int i=0;i<max_comb_size;i++)
    {
        Qmax[i].reserve(3);
        for(int j=0;j<3;j++) Qmax[i][j].reserve(4);
    }
    vector<vector<int>>> I(M,vector<int>(N));
    for (int n=0;n<max_comb_size;n++)
    {
        vector<int> zero_to_one_under_M;
        for(int i=0;i<M;i++) zero_to_one_under_M.push_back(i+1);
        vector<int> cb;
        for(int i=0;i<n+1;i++) cb.push_back (i+1);
        vector<vector<int>>> CombNK;
        recursive_combination(zero_to_one_under_M.begin
        (),zero_to_one_under_M.end(),0,cb.begin(),cb.end(),0,zero_to_one_under_M.size() -
        (n+1),display,CombNK);
        vector<vector<vector<int>>>> result=maxweight(CombNK, Q,P);
        Xmax.push_back(result[0][0][0]);
        Ymax.push_back(result[1][0][0]);
        Qmax.push_back(result[2]);}
    vector<int> Options;
    int maxymax=0;
    for(int j=0;j<Ymax.size();j++)
    {
        if(Ymax[j]>maxymax) maxymax=Ymax[j];
    }
    for(int j=0;j<Ymax.size();j++)
    {
        if(Ymax[j]==maxymax) Options.push_back(j);
    }
    vector<int> Randomized_Options=random_permutation(Options);
    int Selection=Randomized_Options[0];
    vector<int> zero_to_one_under_M;
    for(int i=0;i<M;i++) zero_to_one_under_M.push_back(i);
    vector<int> cb;
    for(int i=0;i<Selection+1;i++) cb.push_back (i+1);
    vector<vector<int>>> Available_combinations;
    recursive_combination(zero_to_one_under_M.begin
    (),zero_to_one_under_M.end(),0,cb.begin(),cb.end(),0,zero_to_one_under_M.size() -
    (Selection+1),display,Available_combinations);
    vector<int> CombinationToTransmit=Available_combinations[Xmax[Selection]-1];
    for (int k=0;k<CombinationToTransmit.size();k++)
    {
        int i=CombinationToTransmit[k];
        for(int j=0;j<Qmax[CombinationToTransmit.size()-1][i].size();j++)
        {
            if(Qmax[CombinationToTransmit.size()-1][i][j]==1) I[i][j]=1;
        }
    }
}
```

Της περνάμε τα ορίσματα Q (που είναι όπως είπαμε ο αριθμός πακέτων στην ουρά σε κάθε κατάσταση), max_comb_size (ο μέγιστος αριθμός πακέτων που μπορεί να συνδυαστούν για να προστεθούν με χοr), P (ο πίνακας πιθανοτήτων όπως είπαμε προηγουμένως). Το Q έχει την παρακάτω μορφή (τυχαίο παράδειγμα).

Q=

0 1 0

0 0 0

Με το Memory στην συγκεκριμένη περίπτωση να έχει ένα πακέτο το οποίο να είναι σε κατάσταση good

Memory=[1 1 2 1 1]

Δηλαδή στην πρώτη γραμμή του Q μπαίνουν τα πακέτα από την πρώτη ροή και στην δεύτερη από την δεύτερη ροή.

Στην πρώτη στήλη μπαίνει ο αριθμός των πακέτων που είναι σε κατάσταση u, στην δεύτερη σε κατάσταση g και στην τρίτη σε κατάσταση b .

Η συνάρτηση scheduler καλώντας την συνάρτηση maxweight που είπαμε στην θεωρία δίνει τιμές στον πίνακα I που δείχνει ποια πακέτα θα φύγουν.

Η συνάρτηση maxweight επιλέγει το μεγαλύτερο από την μέγιστη τιμή του

```
for(int k=0;k<N;k++) weight1=weight1+G1[q1][k]*(Q[i][q1]-Q[i][k]);
```

για ένα πακέτο και για δύο πακέτα τη μέγιστη τιμή weight2_1+ weight2_2 με weight2_1 και weight2_2 να δίνονται στο πρόγραμμα όπως παρακάτω

```
double weight2_1,weight2_2=0;
for(int k=0;k<N;k++)
weight2_1=weight2_1+G1[q1][k]*(Q[i1-1][q1]-Q[i1-1][k]);
weight2_1=weight2_1+G1[q1][N]*Q[i1-1][q1];
for(int k=0;k<N;k++)
weight2_2=weight2_2+G2[q2][k]*(Q[i2-1][q2]-Q[i2-1][k]);
weight2_2=weight2_2+G2[q2][N]*Q[i2-1][q2];
```

Συνάρτηση maxweight


```

        double weight2_2=0;
        for(int k=0;k<N;k++) weight2_2=weight2_2+G2[q2][k]*(Q[i2-1][q2]-
        Q[i2-1][k]);
        weight2_2=weight2_2+G2[q2][N]*Q[i2-1][q2];
        if(weight2_2<0) weight2_2=0;

        weight2=weight2_1+weight2_2;

        if(weight2>maxweight2)
        {
            comb2.clear();
            maxweight2=weight2;
            comb2.push_back(q1);
            comb2.push_back(q2);
        }
        G1.clear();
        G2.clear();

    }

    }
    if(maxweight2>Ymax)
    {
        Xmax=i+1;
        Ymax=maxweight2;
        for(int i=0;i<M;i++)
        {
            for(int j=0;j<N;j++)
            {
                Qmax[i][j]=0;
            }
            i1=CombNK[i][0];
            i2=CombNK[i][1];
            Qmax[i1-1][comb2[0]]=1;
            Qmax[i2-1][comb2[1]]=1;
        }
        break;
    }
    default:
        cout<<"error:n is larger than 4!!!";
}

}

return_value.push_back(vector<vector<int>>());
return_value[0].push_back(vector<int>());
return_value[0][0].push_back(Xmax);
return_value.push_back(vector<vector<int>>());
return_value[1].push_back(vector<int>());
return_value[1][0].push_back(Ymax);
return_value.push_back(Qmax);

return return_value;
}

```

Το G1 και το G2 βρίσκονται από το connectivity και έχουν την μορφή

π.χ.

0	0.8000	0.2000	0
0	1.0000	0	0
0	0	1.0000	0

Και εκφράζει τις πιθανότητες να πάει από τη μια κατάσταση στην άλλη.

Transmit

```
void transmit(const vector<vector<int>> &I,vector<vector<int>> &Q,vector<vector<int>> &Mem_in )
{
    int row;
    vector<vector<vector<int>>> returntable;
    int N=Q[0].size();
    int M=Q.size();
    int entries=Mem_in.size();
    vector<vector<int>> Packets;
    vector<int>::iterator iter;
    vector<int> i,j;
    for(int q=0;q<I.size();q++)
    {
        for(int w=0;w<I[0].size();w++)
        {
            if(I[q][w]>0)
            {
                i.push_back(q);
                j.push_back(w);
            }
        }
    }
    for (int k=0;k<i.size();k++)
    {
        if(Q[i[k]][j[k]]>0)
        {
            int z=0;
            while(z>=0)
            {
                if((Mem_in[z][1]==i[k]+1) && (Mem_in[z][2]==j[k]+1))
                {
                    Packets.push_back(Mem_in[z]);
                    z=-1;
                }
                else
                {
                    z=z+1;
                }
            }
        }
    }
    int m=Packets.size();
    for (int k=0;k<m;k++)
    {
        int id_k=Packets[k][0];

        int paketo_poustelnetai;
        int prod_packets=1;
        for(int i=0;i<Packets.size();i++)
        {
            if(id_k==Packets[i][0]) paketo_poustelnetai=Packets[i][1];
        }

        for(int i=0;i<Packets.size();i++)
        {
            prod_packets*=Packets[i][2+paketo_poustelnetai];
        }
        if(prod_packets==1)
        {
            Q[Packets[k][1]-1][Packets[k][2]-1]=Q[Packets[k][1]-1][Packets[k][2]-1]-1;
            for(int i=0;i<Mem_in.size();i++)
            {
                if(Mem_in[i][1]==id_k)
            }
        }
    }
}
```

```

        vector<vector<int>>::iterator iter_row;
        iter_row=Mem_in.begin()+i;
        Mem_in.erase(iter_row);
        i-=1;
    }
}

else
{
    int current_network;
    int current_queue;
    for(int i=0;i<Mem_in.size();i++)
    {
        if(Mem_in[i][0]==id_k)
        {
            current_network=Mem_in[i][1];
            current_queue=Mem_in[i][2];
        }
    }
    vector<int> other_networks;
    for(int i=0;i<Packets.size();i++)
    {
        other_networks.push_back(Packets[i][1]);
    }
    for(int i=0;i<other_networks.size();i++)
    {
        if(other_networks[i]==current_network)
        {
            iter=other_networks.begin()+i;
            other_networks.erase(iter);
        }
    }

    vector<int> knowledge_vector=dec2tri(current_queue-1);
    for(int i=0;i<M-knowledge_vector.size()-1;i++)
    {
        iter=knowledge_vector.begin();
        knowledge_vector.insert(iter,0);
    }
    vector<int> index_vector;
    for(int i=0;i<M;i++) index_vector.push_back(i+1);
    iter=index_vector.begin()+current_network-1;
    index_vector.erase(iter);
    int other_network=other_networks[0];
    vector<int> knowledge_vector_tocompare;
    int sum_of_knowledge_vector_tocompare=0;
    for(int i=0;i<index_vector.size();i++)
    {
        if(index_vector[i]==other_network)
        {
            knowledge_vector_tocompare.push_back(knowledge_vector[i]);
            sum_of_knowledge_vector_tocompare+=knowledge_vector[i];
        }
    }

    if(sum_of_knowledge_vector_tocompare==0)
    {
        for(int i=0;i<Packets.size();i++)
        {
            if(Packets[i][0]==id_k) row=i;
        }
        if(Packets[row][other_network+2]==1)
        {
            for(int w=0;w<index_vector.size();w++)
            {
                if(index_vector[w]==other_network) knowledge_vector[w]=1;
            }
        }
    }
}

```

```

else
{
    for(int w=0;w<index_vector.size();w++)
    {
        if(index_vector[w]==other_network) knowledge_vector[w]=2;
    }
}
else
{
}

int new_queue=tri2dec(knowledge_vector)+1;
Q[Packets[k][1]-1][new_queue-1]=Q[Packets[k][1]-1][new_queue-1]+1;
Q[Packets[k][1]-1][current_queue-1]=Q[Packets[k][1]-1][current_queue-1]-1;
for(int i=0;i<Mem_in.size();i++)
{
    if(Mem_in[i][0]==id_k) row=i;
}
Mem_in[row][2]=new_queue;
}
}
}

```

Η transmit παίρνει σαν είσοδο τις I, Q και Mem_in και αλλάζει τις Q και Mem_in με βάση το I.

Δηλαδή στην transmit γίνεται η αποστολή των πακέτων και είτε αποκωδικοποιούνται και φεύγουν από την ουρά είτε γυρνάνε με επιπλέον πληροφορία. Ανάλογα τρέχουν και οι υπόλοιπες προσομοιώσεις.

Συμπεράσματα για τις προσομοιώσεις

Η προσομοίωση με το main_qba είναι στοχαστικής γνώσης, όπως στοχαστικής γνώσης είναι και το main_core_sto. Είναι χαρακτηριστικό ότι το main_qba δίνει πολύ καλύτερα αποτελέσματα επειδή διαλέγει ποια πακέτα θα στείλει με βάση έναν πολύ αποδοτικό αλγόριθμο.

Η main_core_sto διαλέγει τυχαία ποια πακέτα θα στείλει σύμφωνα με έναν κανόνα ότι πρέπει να στέλνει δύο πακέτα μόνο αν όλα τα q_{ij} είναι πάνω από 0.8 αλλά αυτό δεν συμβαίνει ποτέ οπότε στέλνει πάντα 1 πακέτο. Αυτό έχει σαν αποτέλεσμα να μην έχουμε ποτέ αποτυχία αλλά και το να αργεί να στείλει όλα τα πακέτα.

Οι υπόλοιπες προσομοιώσεις είναι ντετερμινιστικές με την `core_det_adv` να διαλέγει να στείλει μόνο τους πακέτα που αν είναι δυνατόν να μην έχουν ακουστεί από άλλα τερματικά προορισμού. Αυτό είναι καλό γιατί αν κάποιο έχει ακουστεί και από τον άλλο κόμβο μπορούμε να το χρησιμοποιήσουμε αργότερα για κωδικοποίηση. Αν όμως είναι δεν υπάρχουν πακέτα που έχουν ακουστεί και από άλλα τερματικά διαλέγει κανονικά ένα από αυτά που είναι στην ουρά και αν μην έχει ακουστεί από το άλλο τερματικό.

Η `main_qba_det` διαλέγει πακέτα σύμφωνα με τον αλγόριθμο ελαχιστοποίησης του

$$Z(I) = x_{s1}^i * \mu_{n\ s1}^i(I) + x_{s2}^j * \mu_{n\ s2}^j(I)$$

Όπως είχαμε πει και στη θεωρία. Όπου $\mu_{n\ s1}^i(I)$ ο ρυθμός εξυπηρέτησης που είναι ίσος με τον ρυθμό r_i^{down} σε περίπτωση που στέλνεται ένα πακέτο και σε περίπτωση που στέλνονται δύο πακέτα είναι μηδέν αν $s_2(i) = 0$ ή $\min\{r_i^{\text{down}}, r_j^{\text{down}}\}$ σε περίπτωση που $s_2(i) = 1$. Το x_{s1}^i είναι ο αριθμός των πακέτων που έχει κάθε μια ουρά για κάθε ένα τερματικό πηγή.

Ανάλυση με arrays

Η όλη διαδικασία έγινε για το λόγο της αποδοτικότητας του αλγορίθμου.

Θεωρήθηκε ότι με απλούς πίνακες θα είχαμε πιο γρήγορες προσομοιώσεις.

Ακολουθήθηκε η ίδια ακριβώς μεθοδολογία αλλά για να μην έχουμε πρόβλημα με τα μεγέθη των πινάκων δημιουργήθηκαν κλάσεις που εμπεριείχαν πίνακες καθώς και τα μεγέθη τους, τον αριθμό των χρησιμοποιούμενων στοιχείων και κάποιες βασικές μεθόδους για κάθε μια κλάση.

Το αποτέλεσμα δεν ήταν αυτό που περιμέναμε και μάλλον οι vectors είναι εκτός από πιο εύχρηστοι το ίδιο σχεδόν αποδοτικοί.

Ίσως βέβαια υπάρχει και περιθώριο βελτίωσης για την μέθοδο με arrays. Παρακάτω δίνεται η κλάση που χρησιμοποιήσαμε για τον πίνακα μίας διάστασης.

```

template <typename T>
struct Array1D
{
private : int length;
          int multiply;

public : T *Data ;
        int size;

        Array1D()
        {
            length    = 10;
            multiply = 2;
            Data = (T *)malloc(length*sizeof(T));
            size = 0;
        }
        Array1D(int _length) //constructors
        {
            length    = _length;
            multiply = 2;
            Data = (T *)malloc(length*sizeof(T));
            size = 0;
        }

        Array1D (const Array1D &a)
        {
            Data = (T*)malloc(a.length*sizeof(T));

            for(int k=0;k<a.size;k++) Data[k]=a.Data[k];

            length = a.length;
            size = a.size;
            multiply =a.multiply ;
        }
        Array1D (T *a,int sizea)
        {
            Data = (T*)malloc(sizea*sizeof(T));

            if (Data == a) return;

            for(int i=0;i<sizea;i++) Data[i]=a[i];
        }

        ~Array1D()
        {
            if (Data)
            {
                free(Data);
                Data = NULL;
            }

            size =0;
        }

        Array1D<T>& Array1D::operator = (Array1D <T> &a)
        {
            if (this == &a) return *this;

            if (a.size == 0)
                clear();

            if (a.length > length)
            {
                length = a.length;
                Data = (T *)realloc(Data, sizeof(T)*length);
            }
        }
    
```

```

        size = a.size;

        memcpy(Data, a.Data, sizeof(T)*a.size);

        return *this;
    }

    void clear()
    {
        size =0;

    }

    void SetSize(int newsize)
    {
        length= multiply*newsize;
        Data = (T*) realloc(Data,length*sizeof(T));
    }

    void insertstart(int times,T value) // at start
    {

        if(times+size >length)
        {
            SetSize(times+size);
        }
        for(int i=0;i<size;i++)
        {
            Data[i+times]= Data[i];
        }
        for(int i=0;i<times;i++)
        {
            Data[i] =value;
        }
        size= size+times;
    }

    void insertzeros(int times) // at start. Same. Value = 0;
    {
        insertstart(times,0);
    }

    void erase(int position)
    {
        for(int i=position;i<size-1;i++)
        {
            Data[i] =Data[i+1];
        }

        size = size - 1 ;
    }

    int *begin()
    {
        return Data;
    }

    int *end()
    {
        return Data + size;
    }

    void print()
    {
        cout <<endl;
        for (int i=0;i<size;i++)
            cout <<Data[i]<<" ";

        cout <<endl;
    }

};

```


Διάγραμμα προσομοίωσης



Σύγκριση αποδοτικότητας προσομοιώσεων

Μετά από τρέξιμο και των τριών τρόπων (matlab,vectors,arrays) φαίνεται ότι οι βέκτορες υπερτερούν κατά πολύ σε αποδοτικότητα ενώ τα αποτελέσματα όπως αναμένονταν είναι τα ίδια. Σε έναν πίνακα σύγκρισης βλέπουμε τα παρακάτω όσον αφορά τον χρόνο που ήθελαν να τρέξουν τα τρία προγράμματα σε 100,1000 και 50000 επαναλήψεις.

Επαναλήψεις	Matlab Χρόνος(sec)	Vectors(sec)	Arrays(sec)
100	6,53	0,95	6,3
1000	65	7,706	62
50000	17179	1636	

Δηλαδή στις 50000 επαναλήψεις βλέπουμε ότι με τα vectors είχαμε βελτίωση του χρόνου κατά 10 φορές(που σημαίνει ότι το πρόγραμμα έτρεξε στο 10% του χρόνου του προγράμματος στο matlab).

REFERENCES

- [1] Georgios S. Paschos, Leonidas Georgiadis and Leandros Tassiulas, "Optimal scheduling with pairwise XORing of packets under statistical overhearing information and feedback" University of Thessaly, Volos, Greece
- [2] *Bjarne Stroustrup* "The C++ Programming Language" ,AT&T, 2000
- [3] Scott Meyers "*Effective C++*",*Addison Wesley* ,2005
- [4] Driscoll "Learning Matlab" , SIAM ,2009