Ατομική Διπλωματική Εργασία

ΠΡΟΣΟΜΟΙΩΣΗ ΑΛΓΟΡΙΘΜΩΝ ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΠΑΚΕΤΩΝ ΣΕ ΑΣΥΡΜΑΤΑ ΔΙΚΤΥΑ ΣΤΗΝ ΠΑΡΟΥΣΙΑ ΚΑΚΟΒΟΥΛΗΣ ΔΙΑΤΑΡΑΧΗΣ ΤΗΣ ΕΠΙΚΟΙΝΩΝΙΑΣ

Κυριάκος Ιακώβου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2016

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Προσομοίωση Αλγορίθμων Χρονοπραγραμματισμού Πακέτων Σε Ασύρματα Δίκτυα Στην Παρουσία Κακόβουλης Διαταραχής Της Επικοινωνίας

Κυριάκος Ιακώβου

Επιβλέπων Καθηγητής
Δρ. Χρύσης Γεωργίου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2016

Ευχαριστίες

Φτάνοντας στο τέλος της παρούσας διπλωματικής εργασία, θα ήθελα να ευχαριστήσω όλα τα άτομα που με βοήθησαν. Που ίσως χωρίς την δική τους στήριξη και βοήθεια να μην ήταν δυνατή η υλοποίηση και ολοκλήρωση της παρούσας διπλωματικής εργασίας.

Αρχικά, θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου Δρ. Χρύση Γεωργίου για την επίβλεψη της εργασίας, τις πολύτιμες συμβουλές και την συμπαράσταση του σε όλη τη διάρκεια της εκπόνησης της εργασίας. Επιπλέων για την εμπιστοσύνη που μου έδειξε, επιτρέποντας μου να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου, που χωρίς την ψυχική και υλική βοήθεια τους καθώς και την συμπαράσταση τους όποτε την χρειαζόμουν, δεν θα ήταν δυνατό να ολοκληρώσω τις προπτυχιακές μου σπουδές.

Περίληψη

Ως φυσικό επακόλουθο της γρήγορης ανάπτυξης της τεχνολογίας, ο άνθρωπος εφεύρε πολλές ασύρματες συσκευές όπως τα έξυπνα τηλέφωνα. Συνεπώς για να μπορεί να εκμεταλλευτή την κινητικότητα του χρήστη και της συσκευής ταυτόχρονα σε μέγιστο βαθμό, ανέπτυξε μηχανισμούς ασύρματης επικοινωνίας. Λόγο της μεγάλης δημοτικότητας αυτών τον συσκευών η ασύρματη μετάδοση δεδομένων έγινε ολοένα και πιο δημοφιλής. Παρόλα αυτά υπάρχει η πρόκληση της αντιμετώπισης διαταραχών της ασύρματης επικοινωνίας, κυρίως όταν αυτές γίνονται κακόβουλα/εσκεμμένα. Η πρόκληση υπάρχει λόγω του ότι αυτές οι συσκευές υπάρχουν ήδη και μπορεί να τις προμηθευτεί ο καθένας.

Η παρούσα διπλωματική εργασία ασχολείται με την προσομοίωση αλγορίθμων χρονοπρογραμματισμού πακέτων στην παρουσία σφαλμάτων με την χρήση της πλατφόρμας Network Simulation (NS-3). Οι αλγόριθμοι έχουν ως κύριο στόχο την μεγιστοποίηση του ωφέλιμου φορτίου που αποστέλλεται, από τον αποστολέα, στον παραλήπτη, λαμβάνοντας υπόψη την ανατροφοδότηση για τα πακέτα που έχει καταστρέψει η κακόβουλη συσκευή.

Τέλος γίνεται πειραματική αξιολόγηση των αλγορίθμων και εξαγωγή συμπερασμάτων.

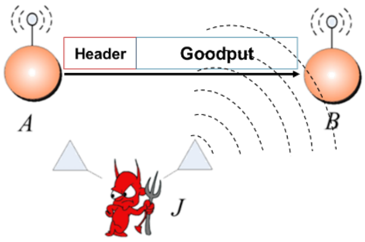
Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

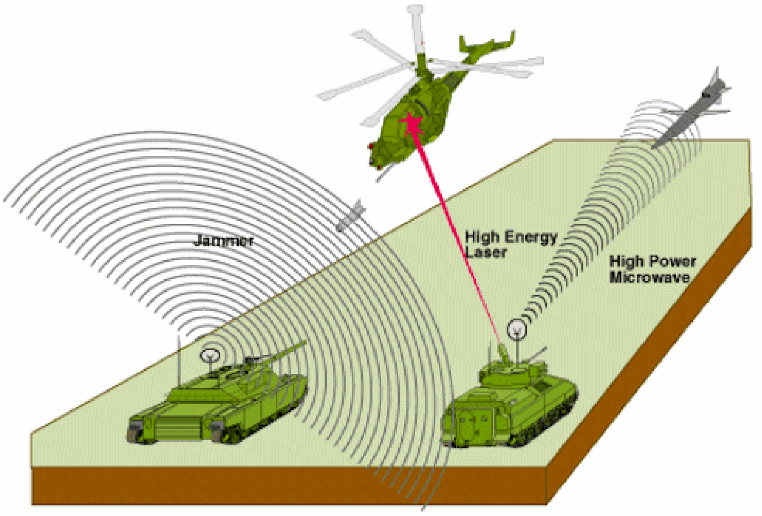
1.1 Κίνητρο

Λόγω της ραγδαίας ανάπτυξης της τεχνολογίας και κυρίως των ασύρματων συσκευών όπως τα έξυπνα κινητά τηλέφωνα, η μετάδοση δεδομένων μέσω ασύρματων δικτύων/μέσων γίνεται ολοένα και πιο δημοφιλής. Για αυτό το λόγο αποτελεί ελκυστικό πεδίο έρευνας για πολλούς ερευνητές εδώ και αρκετό καιρό [3, 4, 5, 7, 8, 12, 13, 14, 15]. Παρόλα αυτά υπάρχει η πρόκληση της αντιμετώπισης διαταραχών της ασύρματης επικοινωνίας, κυρίως όταν αυτές γίνονται κακόβουλα/εσκεμμένα κάτω από διαφορετικές συνθήκες [8, 11]. Για να γίνουμε πιο συγκεκριμένη, υπάρχει ανάγκη να αναπτυχθούν μηχανισμοί που θα αυξήσουν όσο το δυνατό γίνεται το ωφέλιμο φορτίο που στέλνεται διαμέσου των ασύρματων μέσων. Με άλλα λόγια να ελαττώνουν όσο το δυνατό γίνεται τη ζημιά που προκαλούν εσκεμμένα οι κακόβουλες συσκευές στα πακέτα που διαδίδονται, αλλάζοντας κάποιες πληροφορίες μέσα σε αυτά, ώστε να «αχρηστέψουν» την πληροφορία που περιέχουν.



Εικόνα 1.1 [17]

Με τη βοήθεια της Εικόνας 1.1 μπορούμε να κατανοήσουμε καλύτερα το πρόβλημα που αναφέραμε στην προηγούμενη παράγραφο. Πιο συγκεκριμένα ο χρήστης A (αποστολέας) θέλει να επικοινωνήσει με το χρήση B (παραλήπτης), μέσω ενός ασύρματου μέσου επικοινωνίας. Όμως τι γίνεται όταν υπάρχει η κακόβουλη συσκευή J, η οποία καταστρέφει κάποια από τα πακέτα που στέλνονται διαμέσου του ασύρματου δικτύου.



Εικόνα 1.2 [16]

Μια από της πρώτες εφαρμογές των κακόβουλων συσκευών ήταν για στρατιωτικούς σκοπούς. Μέσω της Εικόνας 1.2 μπορούμε να δούμε ένα απλό παράδειγμα όπου τα δύο θωρακισμένα οχήματα διαθέτουν δύο συσκευές που παρεμβάλουν στο ασύρματο μέσο (αέρας). Με αυτό τον τρόπο διακόπτουν την επικοινωνία ανάμεσα στο ελικόπτερο και το τηλεκατευθυνόμενο βλήμα. Ομοίως το δεύτερο παρεμβάλει στον τηλεκατευθυνόμενο πύραυλο έτσι ώστε να μην μπορεί να δεχτεί οδηγίες και να βρει ακριβείς στόχο [16].

Όμως σήμερα τέτοιου είδους συσκευές μπορεί να της προμηθευτεί ο οποιοσδήποτε, μέσω διαδικτυακών καταστημάτων [18, 19, 20]. Έτσι είναι συνετό να αναπτυχθούν μηχανισμοί αντιμετώπισης, οι οποίοι θα βοηθούν στην μείωση της ζημίας που προκαλούν οι κακόβουλες συσκευές, στο μέγιστο δυνατό βαθμό. Επιπλέον με αυτό το τρόπου θα έχουμε την δυνατότητα να εκμεταλλευόμαστε στο μέγιστο δυνατό βαθμό τους ασύρματους πόρους/ασύρματα δίκτυα. Ας μην ξεχνάμε ότι οι ασύρματοι πόροι (συχνότητες αποστολής διαμέσου ασύρματων μέσων) είναι πολύ ακριβοί και περιορισμένη [7]. Επίσης με αυτό τον τρόπο θα αυξήσουμε την αξιοπιστία των ασύρματων δικτύων/μέσων επικοινωνίας, που μας παρέχουν αρκετά πλεονεκτήματα όπως τη κινητικότητα συσκευής (device mobility) και τη κινητικότητα χρήστη (user mobility) [7].

1.2 Σκοπός

Η παρούσα διπλωματική εργασία έχει ως σκοπό την υλοποίηση, επέκταση και πειραματική αξιολόγηση ντετερμινιστικών αλγόριθμων χρονοπρογραμματισμού που αποφασίζει τα μήκη των πακέτων που πρόκειται να μεταφερθούν μέσω ενός ασύρματου μέσου. Με κύριο στόχο τη μεγιστοποίηση του συνολικού οφέλους (χρήσιμο/ωφέλιμο φορτίο) που αποστέλλεται σε μια χρονική περίοδο T, λαμβάνοντας υπόψη την ανατροφοδότηση για τα πακέτα που έτυχαν παρεμβολών.

Μελετώντας κάποιος τις κακόβουλες συσκευές μπορεί εύκολα να δει ότι αυτές έχουν την δυνατότητα να παρεμβάλουν σχεδόν όλους τους τύπους δικτύων, όπως GSM, 3G, 4G, 4G LTE, WIFI, GPS. Επίσης ότι ανάλογα με την χωρητικότητα της μπαταρίας τους, μπορούν να λειτουργήσουν για ένα συγκεκριμένο χρονικό διάστημα. Επιπλέων ότι χρειάζονται ένα συγκεκριμένο χρονικό διάστημα ώστε να επαναφορτίσουν την μπαταρία τους, μέσω του μηχανισμού επαναφόρτισης που διαθέτουν [18, 19, 20]. Συνοψίζοντας αυτές οι συσκευές διαθέτουν ένα συγκεκριμένο ρυθμό επαναφόρτισης η κάθε μια και ένα μέγιστο αριθμό πακέτων που μπορούν να παρεμβάλουν ανάλογα με την χωρητικότητα της μπαταρίας τους. Πιο κάτω στην Εικόνα 1.3 μπορούμε να δούμε ένα παράδειγμα μίας κακόβουλης συσκευής μαζί με τα χαρακτηριστικά που έχουμε προαναφέρει.

Εικόνα 1.3 [18, 19, 20]

Οι αλγόριθμοι που έχουμε υλοποίηση και επεκτείνει λαμβάνουν υπόψη το ρυθμό επαναφόρτισης της μπαταρίας (*ρ*) και το μέγιστο αριθμό πακέτων που μπορεί να παρεμβάλει η κακόβουλη συσκευή (*error token* ή *f*). Αυτοί έχουν αναλυθεί σε θεωρητικό επίπεδο στα δυο ερευνητικά άρθρα [1, 2]. Όμως κανείς μέχρι σήμερα δεν έχει ασχοληθεί με την πρακτική εφαρμογή/εξέταση τους. Έτσι ένας από τους στόχους της παρούσας διπλωματικής εργασίας είναι η ανάλυση της πρακτικής εφαρμογής των αλγορίθμων μέσω προσομοίωσης. Όμως η κύρια συνεισφορά της παρούσας διπλωματικής εργασίας είναι να δώσει απαντήσεις σε ερωτήματα, τα οποία λόγο μαθηματικής πολυπλοκότητας είναι δύσκολο να απαντηθούν μέσω θεωρητικής ανάλυσης. Εν συντομία να συμπληρώσουν και να επιβεβαιώσουν την θεωρητική ανάλυση των αλγορίθμων.

Οι αλγόριθμοι υλοποιήθηκαν και προσομοιώθηκαν με τη χρήση του εργαλείου προσομοιώσεις δικτύων Network Simulation 3 (NS-3) [9, 10]. Ακολούθως τα αποτελέσματα που προέκυψαν από τις προσομοιώσεις επεξεργάστηκαν για τον καθορισμό του κατά πόσο οι αλγόριθμοι αυτοί μειώνουν στο μέγιστο δυνατό την απώλεια πληροφορίας, λόγο αλλοιώσεις του πακέτου από την κακόβουλη συσκευή.

1.3 Μεθοδολογία

Για την εκπόνηση της παρούσας διπλωματικής εργασίας ακολουθήθηκε η ακόλουθη η μεθοδολογία.

Αρχικά έγινε μελέτη των αλγορίθμων που περιγράφονται στα δύο ερευνητικά άρθρα [1, 2], έτσι ώστε να κατανοηθεί πλήρως η λογική των αλγορίθμων.

Για την πειραματική αξιολόγηση χρησιμοποιήθηκε ο προσομοιωτής δικτύων Network Simulator 3, όπου χρειάστηκε να γίνει εκμάθηση του εργαλείου. Αυτή έγινε μέσω των επίσημων εγχειριδίων του προσομοιωτή NS-3 [9, 10]. Ακολούθως τρέξαμε μικρά παραδείγματα, έτσι ώστε να είμαστε σίγουροι ότι έγινε σωστή εκμάθηση του προσομοιωτή.

Ακολούθως έγινε υλοποίηση των αλγορίθμων και πειραματική αξιολόγηση αυτών. Σε τελικό στάδιο αναλύθηκαν τα αποτελέσματα έτσι ώστε να γίνει εξαγωγή συμπερασμάτων.

1.4 Δομή Ατομικής Διπλωματικής Εργασίας

Δομή Εργασίας	
Κεφάλαιο 2	▪ Σύντομη περιγραφή του Αλγορίθμου Περιγραφή Αλγορίθμου ADP(*T*,*1*) ▪ Σύντομη περιγραφή του Αλγορίθμου Περιγραφή Αλγορίθμου ADP(*T*,*f*) με *f* > 1 ▪ Σύντομη περιγραφή της δυναμικής εκδοχής του Αλγορίθμου ADP_CHARCHING(*T*,*f*) με *f* > 1 ▪ Σύντομη περιγραφή του Αλγορίθμου Uniform
Κεφάλαιο 3	▪ Σύντομη εισαγωγή για το Network Simulation 3, δίδονται οι παραμέτροι προσομοίωσης, τα σενάρια προσομοιώσεις και τα αποτελέσματα της πειραματικής αξιολόγησης για αυτά.
Κεφάλαιο 4	▪ Εξαγωγή των γενικών συμπερασμάτων και αναφορά για μελλοντική εργασία.

Κεφάλαιο 2

Περιγραφή Αλγορίθμων

Σε αυτό το Κεφάλαιο θα περιγράψουμε τους αλγόριθμους που υλοποιήθηκαν στην παρούσα διπλωματική εργασία. Πιο συγκεκριμένα στο Κεφάλαιο 2.1 περιγράφεται ο αλγόριθμος ADP(*T,1*), όπου η κακόβουλη συσκευή μπορεί να καταστρέψει μόνο ένα πακέτο. Στο Κεφάλαιο 2.2 περιγράφεται ο αλγόριθμος ADP(*T,f*) με $f \geq 1$, όπου η κακόβουλη συσκευή μπορεί να καταστρέψει περισσότερα από 1 πακέτα. Στο Κεφάλαιο 2.3 περιγράφεται ο αλγόριθμος ADP_CHARCHING (*T,f*) με $f \geq 1$, όπου η κακόβουλη συσκευή μπορεί να καταστρέψει περισσότερα από 1 πακέτα και συγχρόνως μπορεί να επαναφόρτιση την μπαταρία της, με ένα συγκεκριμένο ρυθμό. Στο Κεφάλαιο 2.4 περιγράφεται ο αλγόριθμος Uniform (αποστολή πακέτων σταθερού μήκους), όπου η κακόβουλη συσκευή μπορεί να καταστρέψει περισσότερα από 1 πακέτα και συγχρόνως μπορεί να επαναφόρτιση την μπαταρία της, με ένα συγκεκριμένο ρυθμό.

2.1 Περιγραφή Αλγορίθμου *ADP(T,1)* ADP(T,1)

2.1.1 Βασική ιδέα ADP(*T*,*1*)

Ο ADP(*T*,*1*) δέχεται ως παράμετρο το συνολικό φορτίο που θέλουμε να στείλουμε και το μέγιστο αριθμό πακέτων που μπορεί να παρεμβάλει η κακόβουλη συσκευή (εδώ θεωρούμε ότι η κακόβουλη συσκευή μπορεί να καταστρέψει μόνο ένα πακέτο). Ο ADP(*T*,*1*) λειτουργεί ως εξής, αρχικά επιλέγει το μήκος *p* του πακέτου ανάλογα με *T* (γραμμές 1,6). Εάν το πακέτο μπλοκαριστεί, τότε στέλνει ένα πακέτο μήκους *T*-*p* που εγγυημένα δεν θα μπλοκαριστεί (γραμμές 7,8). Εάν όμως το πακέτο σταλεί επιτυχώς τότε καλείτε ξανά ο αλγόριθμος ADP(*T*-*p*,*1*) (γραμμή 10). Ακολουθείτε η ίδια διαδικασία μέχρι να σταλεί ολόκληρο το φορτίο που θέλουμε να στείλουμε. Πιο κάτω δίνεται ο ψευδοκώδικας του αλγορίθμου. Αυτό που δεν έχουμε εξήγηση μέχρι τώρα είναι ότι το *i* παίρνει μια τιμή έτσι ώστε το *T* να ανήκει στα όρια που ορίζονται στη γραμμή 4. Στη συνέχεια με βάση την τιμή που θα πάρει το *i* θα οριστούν τα α και β (γραμμή 5), ώστε να μπορεί να οριστεί το μήκος του πακέτου (γραμμή 6). Επίσης ο αλγόριθμος ADP(*T*,*1*) αποτελεί την τερματική περίπτωση του αλγόριθμου ADP(*T*,*f*) με $f \geq 2$ που περιγράφεται στο επόμενο υποκεφάλαιο.

Αλγόριθμος ADP(*T*,*1*) – ADP1

1: **If** $T \in [1, 2)$
2:    **Send** packet with length p = T
3: **else**
4:    **Let** i be integer such that T ε $[\frac{(i-1)i}{2} + 1, \frac{(i+1)i}{2} + 1)$
5:    **Let** α = *i* – 2, and $\beta = \frac{(i-1)j}{2} - 1$
6:    **Send** packet π with length $p = \frac{T+\beta}{\alpha+2} = \frac{T-1}{i} + \frac{i-1}{2}$
7:    **If** packet π is jammed then
8:       **Send** packet with length p′ = T - p
9:    **else**
10:       Call ADP(T-p, 1)

2.1.2 Δυσκολίες και προβληματισμοί

Ο συγκεκριμένος αλγόριθμος υλοποιήθηκε χωρίς να υπάρξουν οποιεσδήποτε δυσκολίες ή προβληματισμοί. Αυτό οφείλεται στο γεγονός ότι αυτός αναλύεται πολύ καλά στα δύο ερευνητικά άρθρα [1, 2]. Επίσης στα ίδια ερευνητικά άρθρα έχει αποδειχθεί με μαθηματική ακρίβεια (θεωρητικά) ότι αυτή η εκδοχή του αλγορίθμου αποτελεί και την βέλτιστή λύση για αυτό το σενάριο. Δηλαδή όταν η κακόβουλη συσκευή μπορεί να καταστρέψει μόνο ένα πακέτο.

2.2 Περιγραφή Αλγορίθμου *ADP(*T, f*)* με f > 1

2.2.1 Βασική ιδέα ADP(*T, f*) *με f > 1*

Τώρα έχουμε μια πιο ρεαλιστική προσέγγιση, όπου η κακόβουλη συσκευή διαθέτει περισσότερα από ένα *error token* (δηλαδή $f > 1$) για μια χρονική περίοδο *T*. Μια πρώτη ιδέα είναι ότι στην περίπτωση που η κακόβουλη συσκευή διαθέτει περισσότερα *error token* από το υπολειπόμενο φορτίο να στέλνουμε όλο το φορτίο (αφού ακόμα έχει την δυνατότητα να το καταστρέψει όλο *f* > εναπομείναντος φορτίου) (γραμμή 1,2 ψευδοκώδικα). Επίσης σε περίπτωση που η κακόβουλη συσκευή καταστρέψει ένα πακέτο δεν μπορούμε να εγγυηθούμε ότι δεν θα καταστρέψει και το επόμενο. Έτσι αν συμβεί αυτό καλούμε τον αλγόριθμο μας ξανά μειώνοντας το T – μέγεθος πακέτου και *f – 1* (γραμμή 5,6 ψευδοκώδικα). Αν το πακέτο σταλεί επιτυχώς τότε καλούμε τον αλγόριθμο μας με *T – μέγεθος πακέτου* και *f* (γραμμή 7,8 ψευδοκώδικα). Όταν μας μείνει μόνο ένα *error token* (*f=1*) τότε καλούμε τον αλγόριθμο που περιγράψαμε στο κεφάλαιο 2 που αποτελεί την βέλτιστη λύση για αυτό την εκδοχή του προβλήματος. Τέλος πρέπει να αναφέρουμε ότι μέχρι και σήμερα δεν έχουμε ορίσει κάποια βέλτιστη λύση, για τον καθορισμό του μεγέθους του πακέτου (για την αρχικοποίηση των α, β, γ τα οποία μπορούμε να δούμε στο ψευδοκώδικα του αλγορίθμου πιο κάτω στην γραμμή 4).

Αλγόριθμος ADP(*T,f*) με *f* >1– ADP2

1: **If** T < *f* +1 then
2:   **Send** packet with length p = T
3: **else**
4:   **Send** packet π with length p $= \frac{aT+\beta}{\gamma}$
5:   **If** packet π is jammed then
6:     **Call** ADP(T-p, *f*-1)
7:   **else**
8:     **Call** ADP(T-p, *f*)

2.2.2 Δυσκολίες και προβληματισμοί

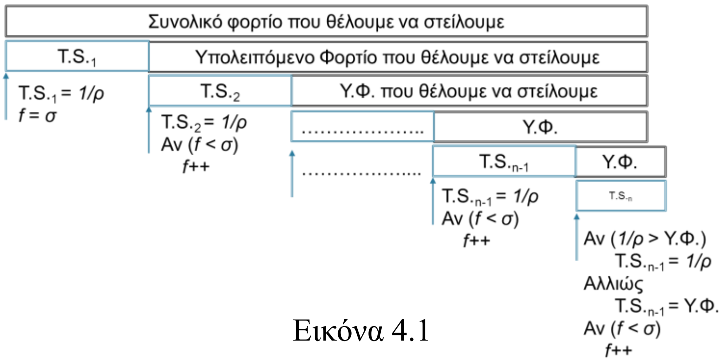
Στο συγκεκριμένο αλγόριθμο, υπήρχε ένα σημείο το οποίο με είχε προβληματίσει αρκετά. Αυτό ήταν το πώς να υπολογίσω το μήκος του πακέτου. Πιο συγκεκριμένα τι τιμές πρέπει να πάρουν τα α, β και γ για τον υπολογισμό του μήκους το πακέτου (γραμμή 4 ψευδοκώδικα). Αυτό που ήταν ξεκάθαρο από το ερευνητικό άρθρο [1] είναι το ότι το β > γ > α (υπάρχει μαθηματική απόδειξη). Σε διαφορετική περίπτωση το μήκος τους πακέτου ενδέχεται να πάρει αρνητικές τιμές. Έτσι μετά από αρκετές δοκιμές (με διαφορετικές εισόδους για τη χρονική περίοδο και την αρχικοποίηση των α, β και γ) για το μήκος του πακέτου αποφασίσαμε να δώσουμε ως αρχική τιμή για το β = 16, για το γ = 8 και για το α = 2. Έπειτα ανάλογα με τα δεδομένα μας αυτά αυξάνονται διατηρώντας τον περιορισμό β > γ > α, έτσι ώστε το μήκος του πακέτου να μην ξεπερνά το φορτίο που θέλουμε να στείλουμε. Αυτό φυσικά δεν αποτελεί την βέλτιστη λύση. Όμως στο παρών στάδιο θέλουμε να δούμε τον τρόπο με τον οποίο ανταποκρίνεται ο συγκεκριμένος αλγόριθμος, έστω αν δεν είναι η βέλτιστη εκδοχή του.

2.3 Περιγραφή Αλγορίθμου ADP_CHARCHING *(*T,f*)* με *f >1*

2.3.1 Βασική ιδέα αλγορίθμου ADP_CHARCHING (*T,f*) με f >1

Σε αυτή την εκδοχή του αλγορίθμου μελετάμε μια ακόμα πιο ρεαλιστική εκδοχή του αλγορίθμου ADP(*T, f*) με $f > 1$ που περιγράψαμε στο Κεφάλαιο 3 (ίσως να είναι και η πιο ρεαλιστικότερη έκδοση που μπορεί να υπάρξει). Μπορούμε να πούμε ότι αυτή η εκδοχή αποτελεί ένας είδος επέκτασης του αλγορίθμου ADP(*T, f*) με $f > 1$. Αφού το μόνο που αλλάζει είναι ότι η κακόβουλη συσκευή διαθέτει ένα μηχανισμό επαναφόρτισης (όπως ηλιακή επαναφόρτιση). Απαραίτητη προϋπόθεση για να τρέξει ο αλγόριθμος μας είναι να γνωρίζουμε ακριβώς ή κατά προσέγγιση τον ρυθμό επαναφόρτισης της κακόβουλης συσκευής. Τώρα θα εξηγήσουμε την βασική ιδέα του αλγορίθμου, αρχικά χωρίζουμε την αρχική μας περίοδο *T* σε μικρότερες περιόδους που ισούνται με τον ρυθμό επαναφόρτισης ($T=1/\rho$) της κακόβουλης συσκευής. Ακολούθως καλούμε τον αλγόριθμο ADP(*T, f*) με $f > 1$ με είσοδο $T=1/\rho$ και $f=\sigma$, μέχρι να στείλουμε όλο μας το φορτίο. Κάθε φορά που περνά η αυτή η χρονική περίοδος, εάν η κακόβουλη συσκευή έχει χρησιμοποιήσει έστω και ένα error token, τότε το $\sigma = \sigma+1$ αλλιώς $\sigma = \sigma$.

2.3.2 Βασική ιδέα αλγορίθμου με τη βοήθεια σχήματος



Εικόνα 4.1

Αρχικά έχουμε το συνολικό φορτίο που θέλουμε να στείλουμε. Στη συνέχεια το μοιράζουμε σε Χρονικές Περιόδους (Time Slot = T.S). Το πρώτο T.S. θα ισούται με το ρυθμό επαναφόρτισης. Ακολούθως ακολουθούμε την ίδια διαδικασία. Το σημείο στο οποίο πρέπει να είμαστε προσεχτική είναι το τελευταίο T.S. το οποίο μπορεί να είναι μικρότερο από το ρυθμό επαναφόρτισης. Αν ισχύει αυτό τότε το τελευταίο T.S. ισούται με το υπολειπόμενο φορτίο. Επίσης εσωτερικά σε κάθε T.S. καλείται ο αλγόριθμος ADP(*T.S., 1*) ή ο ADP(*T.S., f*) όταν το $f > 1$ και ακολουθείται η διαδικασία που περιγράψαμε στα προηγούμενα Κεφάλαια.

2.3.3 Δυσκολίες και προβληματισμοί

Φυσικά η λύση που προτείνεται στο παρών στάδιο δεν αποτελεί τη βέλτιστη δυνατή. Αυτό οφείλεται στην εξής σύμβαση: η συσκευή έχει την δυνατότητα να φορτίσει στην αρχή της νέας περιόδου (Τ=1/ρ) εκτός από την πρώτη κλήση που είναι πλήρως φορτισμένη. Στην πραγματικότητα όμως η συσκευή αρχίζει να φορτίζει από την στιγμή που χρησιμοποιεί το πρώτο της *error token*. Παρόλα αυτά θέλουμε να μελετήσουμε πειραματικά τον τρόπο με τον οποίο ανταποκρίνεται ο αλγόριθμος για αυτή την εκδοχή του προβλήματος. Σε μελλοντικό στάδιο θα ήταν καλό να μελετήσουμε και αυτή την εκδοχή.

2.4 Περιγραφή Αλγορίθμου Uniform

2.4.1 Βασική ιδέα Αλγορίθμου Uniform

Ο αλγόριθμος Uniform στέλνει πάντα πακέτα του ιδίου μήκος. Για τον υπολογισμό του μήκους του πακέτου λαμβάνει υπόψη το μέγιστο αριθμό *error token* που διαθέτει η κακόβουλη συσκευή, τη χρονική περίοδο και το ρυθμό επαναφόρτισης της. Για αυτή την εκδοχή του προβλήματος, έχει αποδειχθεί στο ερευνητικό άρθρο [2] ότι αποτελεί την βέλτιστή λύση.

2.4.2 Σκοπός Υλοποίησης Αλγορίθμου Uniform

Ο αλγόριθμος αυτός έχει αποδειχθεί ότι είναι βέλτιστος για το συγκεκριμένο πρόβλημα (ιδίου μήκους πακέτα). Έτσι εμείς θα των θέσουμε ως σημείο αναφοράς για να εξάγουμε συμπεράσματα για τους αλγόριθμους που περιγράφονται στα προηγούμενα κεφάλαια. Για το λόγο αυτό έχουμε υλοποιήσει δύο εκδοχές του αλγορίθμου. Η μόνη διαφορά σε αυτές είναι ότι η μια εκδοχή ξαναστέλνει τα πακέτα που έχουν καταστραφεί ενώ η άλλη όχι. Έτσι αφού θέσαμε ως σύμβαση από την αρχή ότι δεν θα ξαναστέλνουμε τα πακέτα που καταστρέφονται, οι συγκρίσεις θα γίνουν για αυτή την εκδοχή του προβλήματος. Γιατί στο παρών στάδιο θέλουμε να μελετήσουμε την ανταπόκριση των αλγορίθμων για μια χρονική περίοδο P. Επίσης ας μην ξεχνάμε ότι αυτό που μας ενδιαφέρει στο παρών στάδιο είναι να δούμε πόσο χρήσιμο φορτίο μπορούμε να στείλουμε για μια χρονική περίοδο.

Επιπλέον μέσα στο ερευνητικό άρθρο [3], μπορούμε να δούμε την λεπτομερή απόδειξη της ορθότητας του αλγορίθμου. Επίσης μπούμε να δούμε ότι για τον υπολογισμό του μήκους του πακέτου, ο αλγόριθμος Uniform λαμβάνει υπόψη τη Χρονική Περίοδο (P), το μέγιστο αριθμό σφαλμάτων (σ) και το ρυθμό επαναφόρτισης της μπαταρίας (ρ ώστε να υπολογίσει το μήκος του πακέτου (l). Ο ακριβής τύπος για τον υπολογισμό του μήκους του πακέτου είναι $l^* = \frac{\sqrt{P\big(P\rho+(\sigma-1)(1-\rho)\big)}-P\rho}{P\rho+\sigma-1}$.

Κεφάλαιο 3

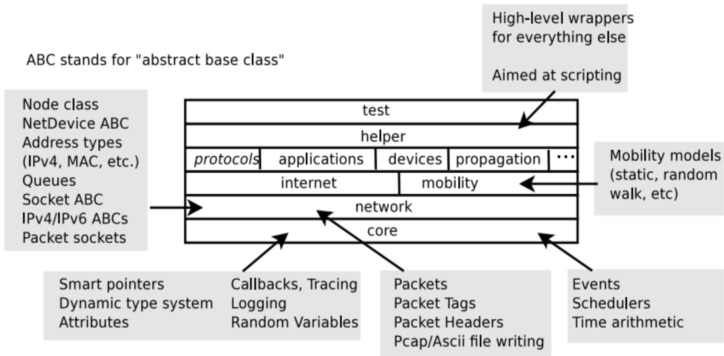
Πειραματική Αξιολόγηση

Σε αυτό το Κεφάλαιο θα περιγράψουμε εν συντομία το προσομοιωτή δικτύων Network Simulation 3 [9, 10] στο Κεφάλαιο 3.1. Στη συνέχεια θα καθορίσουμε τις παραμέτρους προσομοιώσεις το Κεφάλαιο 3.2. Ακολούθως θα ορίσουμε τα σενάρια προσομοιώσεις στο Κεφάλαιο 3.3. Τέλος στο Κεφάλαιο 3.4 θα δούμε τα αποτελέσματα των προσομοιώσεων μέσω γραφικών παραστάσεων και θα εξηγήσουμε την συμπεριφορά τους.

3.1 Προσομοιωτής Network Simulator 3

Ο NS-3 [9,10], ο οποίος χρησιμοποιήθηκε για την προσομοίωση των αλγορίθμων της παρούσας διπλωματικής εργασίας, θεωρείτε ένας από τους δημοφιλέστερους προσομοιωτές διακριτών γεγονότων. Έχει ως κύριο στόχο την έρευνα δικτύων και συγχρόνως τυγχάνει χρήσης για εκπαιδευτικούς σκοπούς. Είναι ένα λογισμικό ανοικτού κώδικα, με στόχο να καλύψει τις σύγχρονες ανάγκες προσομοίωσης στα δίκτυα, το οποίο διέπετε από την άδεια GNU GPLv2 license. Αποτελεί το διάδοχο του προσομοιωτή NS2 που αναπτύχθηκε το 1996 από τον Steve McCanne [9].

Ο NS-3 είναι γραμμένος σε C++. Παρόλα αυτά διαθέτει κάποιες δομές που είναι γραμμένες σε Python. Όσο αφορά τα αρχεία προσομοίωσης (script), ο προσομοιωτής μας δίνει την δυνατότητα να τα γράψουμε σε C++ ή Python. Έτσι οι χρήστες μέσω της γραμμής εντολών σε συνδυασμό με τα εργαλεία για C++ / Python μπορούν να αναπτύξουν τα προγράμματα τους. [9]



Εικόνα 3.1 Οργάνωση Λογισμικού του ns-3 [9,10]

Με τη βοήθεια της Εικόνας 3.1 μπορούμε να δούμε πως είναι οργανωμένος ο πηγαίος κώδικας στο φάκελο *src* του NS-3. Καταρχάς όλα τα στοιχεία που είναι κοινά για όλα τα πρωτόκολλά αναμαρτήτως υλικού και περιβάλλοντος βρίσκονται υλοποιημένα στο φάκελο *src/core*. Επίσης, τα πακέτα που αφορούν τα βασικά αντικείμενα υλοποιούνται στο φάκελο *src/network*. Οι δύο προαναφερθέντες ενότητες έχουν ως κύριο στόχο να δημιουργήσουν την βάση για ένα γενικό πυρήνα προσομοίωσης, ώστε να μπορούν να χρησιμοποιηθούν για διαφορετικά είδη δικτύων. [9]

Υπό την προϋπόθεση ότι ο πηγαίος κώδικας του NS-3 βρίσκεται στο τοπικό σύστημα, πρέπει να μεταγλωττιστεί έτσι ώστε να δημιουργηθούν τα εκτελέσιμα αρχεία. Υπάρχουν πολλά διαθέσιμα εργαλεία για αυτή τη δουλειά. Ένα από αυτά είναι το wa*f*, το οποίο χρησιμοποιήσαμε και στην παρούσα διπλωματική εργασία. Το οποίο έχει αναπτυχθεί χρησιμοποιώντας τη γλώσσα Python. (μέσω του waf μπορούμε να εγκαταστήσουμε και τον NS-3 στον υπολογιστή μας) [9].

Επίσης ο προσομοιωτής χρησιμοποιείται κυρίως σε λειτουργικά συστήματα Linux ή Linux-like περιβάλλοντα αν και υπάρχει η δυνατότητα να τρέξει και σε άλλα λειτουργικά συστήματα μέσω του FreeBSD, Cygwin (για Windows). Επίσης υπάρχει και η επιλογή χρήσης visual machine, όπως το VMware server στο οποίο θα εγκατασταθεί Linux visual machine (λειτουργικό σύστημα). Για την παρούσα διπλωματική εργασία χρησιμοποιήθηκε το λειτουργικό σύστημα Ubuntu 10.4 [9].

3.2 Παράμετροι Προσομοίωσης

Προτού ορίσουμε τα σενάρια προσομοίωσης, είναι φρόνιμο να ορίσουμε τις μετρικές με τις οποίες θέλουμε να εξετάσουμε τους αλγόριθμους μας. Πριν όμως από αυτό θα μπορούμε να δούμε τα χαρακτηριστικά του ηλεκτρονικού υπολογιστή όπου έγινε η υλοποίηση και η πειραματική αξιολόγηση.

3.2.1 Χαρακτηριστικά Υπολογιστή Προσομοίωσης

Οι μετρήσεις έγιναν σε ένα ηλεκτρονικό υπολογιστή με Intel® i7-4700MQ -2.40-3.40 GHz, λειτουργικό σύστημα Ubuntu 14.04.4, 16GB RAM και σκληρό δίσκο SSD χωρητικότητας 512GB & SSD χωρητικότητας 60 GB.

3.2.2 Μετρικές Προσομοίωσης

Για την αξιολόγηση των αλγορίθμων καθορίστηκαν οι πιο κάτω μετρικές:

- Ορίζουμε ως **χρήσιμο φορτίο (Goodput)** το ωφέλιμο φορτίο που μεταφέρεται κατά την διάρκεια της προσομοίωσης, δηλαδή είναι το συνολικό φορτίο, το οποίο δεν δέχτηκε παρεμβολές από τη κακόβουλη συσκευή.
- Ορίζουμε ως **επιπλέων φορτίο** (**Overhead)** το συνολικό φορτίο το οποίο αποτελεί το header size. Δηλαδή είναι το συνολικό φορτίο το οποίο δεν είναι ωφέλιμο για το χρήστη, όμως είναι αναγκαίο κακό ώστε να μπορούν να μεταδοθούν τα πακέτα στον παραλήπτη. Επίσης το header size έχει οριστεί σε 1. Συνεπώς το Overhead ισούται και με το συνολικό αριθμό πακέτων που στέλνονται κατά τη διάρκεια της προσομοίωσης.

<u>Σημείωση</u>: μια άλλη μετρική θα μπορούμε να είναι το μη ωφέλιμο φορτίο (dropped). Όμως λόγω του ότι στο παρών στάδιο δεν μελετάμε αλγόριθμους που στέλνουν ξανά τα πακέτα που έτυχαν παρεμβολής. Έχοντας στοιχειώδη μαθηματική ωριμότητα καταλαβαίνουμε ότι αυτά τα δύο είναι αντιστρόφως ανάλογα και συμμετρικά. Με άλλα λόγια είναι συμμετρικά. Για αυτό το λόγο δεν υπάρχει λόγος να το συμπεριλάβουμε στα αποτελέσματα μας. Παρόλα αυτά θα το ελέγχουμε κατά την διάρκεια των προσομοιώσεων έτσι ώστε να βεβαιωθούμε ότι το συνολικό φορτίο που στέλνεται ισούται με το συνολικό φορτίο που θέλουμε να στείλουμε.

3.3 Σενάρια πειραματικής αξιολόγησης

Αφού καθορίσαμε τις μετρικές αξιολόγησης, τώρα πρέπει να ορίσουμε τα σενάρια στα οποία θα αποτελέσουν την βάση για την πειραματική αξιολόγηση των αλγορίθμων έτσι ώστε να μπορέσουμε να εξαγάγουμε τα συμπεράσματα μας. Τα σενάρια αυτά χωρίζονται σε δύο κατηγορίες όπως μπορούμε να δούμε στην συνέχεια.

3.3.1 Στοχευμένα Σενάρια

Αρχικά ορίσαμε δύο στοχευμένα σενάρια έτσι ώστε να δούμε τα αποτελέσματα των αλγορίθμων κάτω από συγκεκριμένα σενάρια.

Σενάριο 1

Περιγραφή Σεναρίου 1: Αρχικά θα εξετάσουμε τους αλγόριθμος μας για την περίπτωση που όποτε έχει διαθέσιμο *error token*η κακόβουλη συσκευή τότε καταστρέφει το πακέτο που μεταδίδεται από τον αποστολέα στον παραλήπτη. Άρα θα θέσουμε πιθανότητα παρεμβολής να είναι 1, έτσι όποτε η κακόβουλη συσκευή όταν διαθέτη έστω και ένα *error token*τότε το χρησιμοποιεί για να καταστρέψει ένα πακέτο. Αρχικά θα τρέξουμε τους αλγόριθμους για είσοδο T=2048 και το f θα αυξάνεται γραμμικά ανάλογα με τα αποτελέσματα. Ακολούθως θα τρέξουμε το ίδιο σενάριο με T=99999999, έτσι ώστε να δούμε ότι ο αλγόριθμος πως ο αλγόριθμος μας ανταποκρίνεται και για μεγαλύτερο φορτίο. Επίσης οι αλγόριθμοι που χρήζουν επαναφόρτιση θα τρέξουν με την ίδια είσοδο και ρυθμό επαναφόρτισής 1/r με r=0.1, r=0.01, r=0.001 και r=0.0001. Επιλέξαμε αυτούς τους ρυθμούς επαναφόρτισης έτσι ώστε να είναι αρκετά μικροί σε σχέση με το φορτίο έτσι να δούμε πως ανταποκρίνεται οι αλγόριθμοι μας.

Στόχος σεναρίου: Να δούμε πως ανταποκρίνεται ο αλγόριθμος μας για τις 3ης μετρικές που θέσαμε στο σημείο 3.2.2 Μετρικές Προσομοίωσης, όταν η κακόβουλη συσκευή καταστρέφει τα πακέτα όποτε διαθέτη έστω και ένα error token
Αριθμός επαναλήψεων: για αυτό το σενάριο χρειάζεται να τρέξουμε μια φορά τον κάθε αλγόριθμο, αφού πάντα θα καταστρέφει το ίδιο συνολικό αποτέλεσμα. Αυτό είναι απολύτως φυσικό γιατί πάντα θα καταστρέφονται τα ίδια πακέτα.

Εξαιρέσεις**:** για αυτό το σενάριο εξαιρείται αλγόριθμος 1, αφού δεν μπορούμε να αυξήσουμε το *f* του. Αν θυμηθούμε ο αλγόριθμος 1 είναι η τερματική περίπτωση του αλγόριθμου ADP(*T,f*) με *f* >1 και ADP_CHARGING(*T,f*) με *f* >1, δηλαδή εκτελείται όταν έχουμε στη διάθεση μας το πολύ 1 *error token* ($0 \leq f \leq 1$)

Σενάριο 2

Περιγραφή Σεναρίου 2: Τώρα θέλουμε να εξετάσουμε την αντίθετη περίπτωση, δηλαδή όταν η κακόβουλη συσκευή ενώ διαθέτει *error token*δεν καταστρέφει κανένα πακέτο. Για να το πετύχουμε αυτό θέτουμε τη πιθανότητα παρεμβολής να είναι 0. Αρχικά θα τρέξουμε τους αλγόριθμους για είσοδο T=2048 και το *f* θα αυξάνεται γραμμικά ανάλογα με τα αποτελέσματα. Ακολούθως θα τρέξουμε το ίδιο σενάριο με T=99999999, έτσι ώστε να δούμε ότι ο αλγόριθμος πως ο αλγόριθμος μας ανταποκρίνεται και για μεγαλύτερο φορτίο. Επίσης οι αλγόριθμοι που χρήζουν επαναφόρτιση θα τρέξουν με την ίδια είσοδο και ρυθμό επαναφόρτισής 1/r με r=0.1, r=0.01, r=0.001 και r=0.0001. Επιλέξαμε αυτούς τους ρυθμούς επαναφόρτισης έτσι ώστε να είναι αρκετά μικροί σε σχέση με το φορτίο.

Στόχος σεναρίου: Να δούμε πως ανταποκρίνεται ο αλγόριθμος μας για τις 3ης μετρικές που θέσαμε στο σημείο 3.2.2 Μετρικές Προσομοίωσης, όταν οι κακόβουλη συσκευή δεν καταστρέφει κανένα πακέτο ανεξαρτήτως των *error token* που διαθέτει. Αυτό που θέλουμε να εξετάσουμε κυρίως εδώ είναι το overhead, το οποίο θα δημιουργείται. Όσο αφορά το Goodput θα ισούται με το Τα, ενώ το Dropped θα ισούται με 0.

Αριθμός επαναλήψεων: για αυτό το σενάριο χρειάζεται να τρέξουμε μια φορά τον κάθε αλγόριθμο, αφού πάντα θα καταστρέφει το ίδιο συνολικό αποτέλεσμα. Αυτό είναι απολύτως φυσικό γιατί πάντα θα καταστρέφονται τα ίδια πακέτα.

Εξαιρέσεις: για αυτό το σενάριο εξαιρείται αλγόριθμος ADP(*T,1*), αφού δεν μπορούμε να αυξήσουμε το *f* του. Αν θυμηθούμε ο αλγόριθμος ADP(*T,1*) είναι η τερματική περίπτωση του αλγόριθμου ADP(*T,f*) με *f* >1 και ADP_CHARCHING(*T,f*) με *f* >1, δηλαδή εκτελείται όταν έχουμε στη διάθεση μας το πολύ 1 *error token* ($0 \leq f \leq 1$)

3.3.2 Μη Στοχευμένα Σενάρια

Τέλος ορίσαμε δύο μη στοχευμένα σενάρια έτσι ώστε να δούμε τα αποτελέσματα των αλγορίθμων κάτω από πιο ρεαλιστικές συνθήκες. Για να είμαστε ακριβείς το πώς οι αλγόριθμοι ανταπεξέρχονατι όταν γίνεται εισαγωγή σφαλμάτων με τυχαίο τρόπο.

Σενάριο 3

Περιγραφή Σεναρίου 3: Σε αυτό το σενάριο θέλουμε να εξετάσουμε την περίπτωση όπου η κακόβουλη συσκευή καταστρέφει τυχαία κάποια πακέτα. Για να το πετύχουμε αυτό θέτουμε τη πιθανότητα παρεμβολής να είναι 0<πιθανότητα παρεμβολής <1. Αρχικά θα τρέξουμε τους αλγόριθμους για είσοδο T=2048 και το *f* θα αυξάνεται γραμμικά ανάλογα με τα αποτελέσματα. Ακολούθως θα τρέξουμε το ίδιο σενάριο με T=99999999, έτσι ώστε να δούμε ότι ο αλγόριθμος πως ο αλγόριθμος μας ανταποκρίνεται και για μεγαλύτερο φορτίο. Επίσης οι αλγόριθμοι που χρήζουν επαναφόρτιση θα τρέξουν με την ίδια είσοδο και ρυθμό επαναφόρτισής 1/r με r=0.1, r=0.01, r=0.001 και r=0.0001. Επιλέξαμε αυτούς τους ρυθμούς επαναφόρτισης έτσι ώστε να είναι αρκετά μικροί σε σχέση με το φορτίο.

Στόχος σεναρίου: Να δούμε πως ανταποκρίνεται ο αλγόριθμος μας για τις 2^{o} μετρικές που θέσαμε στο σημείο 3.2.2 Μετρικές Προσομοίωσης, όταν οι κακόβουλη συσκευή δεν καταστρέφει κανένα πακέτο ανεξαρτήτως των *error token* που διαθέτει. Αυτό που θέλουμε να εξετάσουμε κυρίως εδώ είναι το overhead, το οποίο θα δημιουργείται. Όσο αφορά το Goodput θα ισούται με το T.

Αριθμός επαναλήψεων: για αυτό το σενάριο χρειάζεται να τρέξουμε μια φορά τον κάθε αλγόριθμο, αφού πάντα θα καταστρέφει το ίδιο συνολικό αποτέλεσμα. Αυτό είναι απολύτως φυσικό γιατί πάντα θα καταστρέφονται τα ίδια πακέτα.

Εξαιρέσεις**:** για αυτό το σενάριο εξαιρείται αλγόριθμος ADP(*T,1*), αφού δεν μπορούμε να αυξήσουμε το *f* του. Αν θυμηθούμε ο αλγόριθμος ADP(*T,1*) είναι η τερματική περίπτωση του αλγόριθμου ADP(*T,1*) και ADP(*T,f*) με *f* >1, δηλαδή εκτελείται όταν έχουμε στη διάθεση μας το πολύ 1 *error token* ($0 \leq f \leq 1$)

Σενάριο 4

Περιγραφή Σεναρίου 4: Σε αυτό το σενάριο θέλουμε να εξετάσουμε την περίπτωση όπου η κακόβουλη συσκευή καταστρέφει τυχαία κάποια πακέτα με εντελώς τυχαία πιθανότητα παρεμβολής (random – αντί για σταθερό σε όλη την προσομοίωση). Με άλλα λόγια, η κακόβουλη συσκευή έχει την δυνατότητα να αλλάξει το ρυθμό παρεμβολής κατά την διάρκεια της προσομοίωσής. Για να το πετύχουμε αυτό θέτουμε τη πιθανότητα παρεμβολής να είναι 0 ≤ random ≤1. Αρχικά θα τρέξουμε τους αλγόριθμους για είσοδο T=2048 και το *f* θα αυξάνεται γραμμικά ανάλογα με τα αποτελέσματα. Ακολούθως θα τρέξουμε το ίδιο σενάριο με T=99999999, έτσι ώστε να δούμε ότι ο αλγόριθμος πως ο αλγόριθμος μας ανταποκρίνεται και για μεγαλύτερο φορτίο. Επίσης οι αλγόριθμοι που χρήζουν επαναφόρτιση θα τρέξουν με την ίδια είσοδο και ρυθμό επαναφόρτισής 1/r με r=0.1, r=0.01, r=0.001 και r=0.0001. Επιλέξαμε αυτούς τους ρυθμούς επαναφόρτισης έτσι ώστε να είναι αρκετά μικροί σε σχέση με το φορτίο.

Στόχος σεναρίου: Να δούμε πως ανταποκρίνεται ο αλγόριθμος μας για τις $3^{ης}$ μετρικές που θέσαμε στο σημείο 3.2.2 Μετρικές Προσομοίωσης, όταν οι κακόβουλη συσκευή καταστρέφει πακέτα με όχι σταθερό ρυθμό ανεξαρτήτως των *error token* που διαθέτει.

Εξαιρέσεις**:** για αυτό το σενάριο εξαιρείται αλγόριθμος ADP(*T,1*), αφού δεν μπορούμε να αυξήσουμε το *f* του. Αν θυμηθούμε ο αλγόριθμος ADP(*T,1*) είναι η τερματική περίπτωση του αλγόριθμου ADP(*T,f*) με *f* >1 και ADP_CHARGING(*T,f*) με *f* >=1, δηλαδή εκτελείται όταν έχουμε στη διάθεση μας το πολύ 1 *error token* ($0 \leq f \leq 1$).

3.4 Αποτελέσματα πειραματικής αξιολόγησης

Αφού τρέξαμε τα σενάρια με τη βοήθεια του προσομοιωτή δικτύων NS-3, πήραμε τα αποτελέσματα και με την βοήθεια της Microsoft Excel 2013 φτιάξαμε τις γραφικές παραστάσεις που παρουσιάζονται στην παρούσα διπλωματική εργασία. Με αυτό τον τρόπο μπορούμε να κατανοήσουμε καλύτερα την συμπεριφορά των αλγορίθμων.

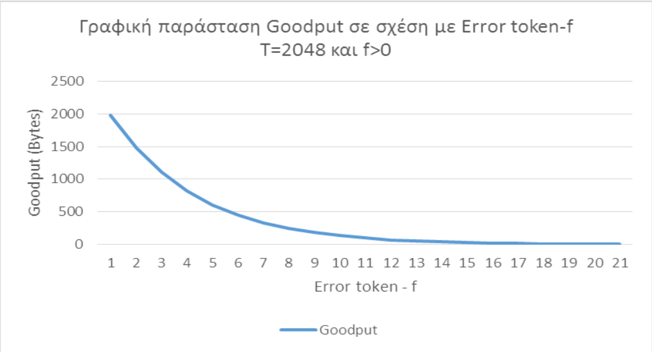
3.4.1 Αποτελέσματα στοχευμένων σεναρίων

Αρχικά θα παρουσιάζουμε και εξηγούμε την συμπεριφορά των δύο πρώτο σεναρίων.

Αποτελέσματα για σενάριο 1

Αποτελέσματα για αλγόριθμο ADP(*T*,*f*) με *f* >1 για T = 2048

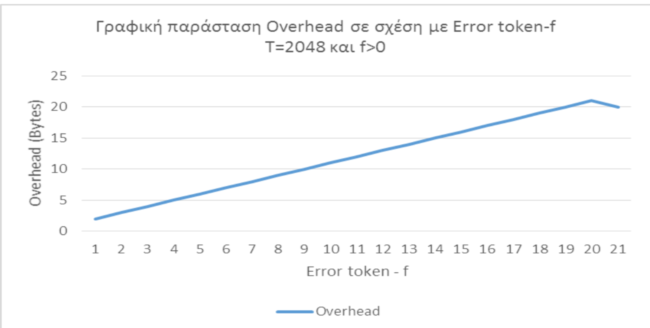
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.1

Στο Σχήματος 3.1 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το ωφέλιμο φορτίο (goodput) μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token* το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα *error token*. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με *f* >= 1). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή.

<u>Overhead</u>

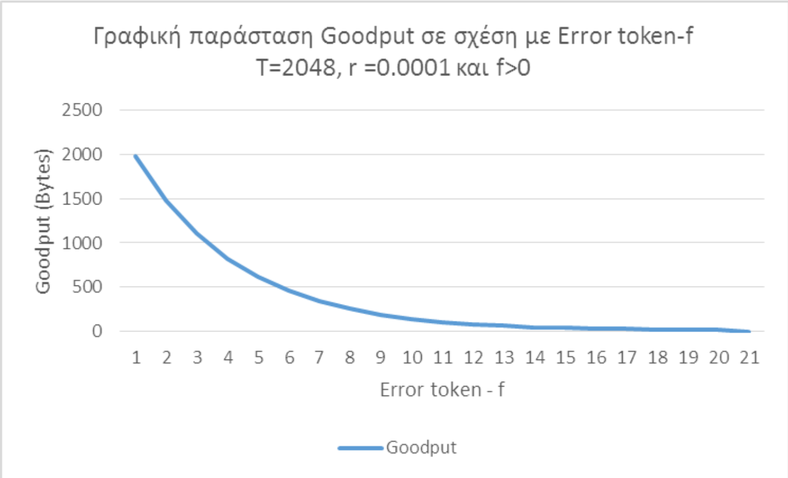


Σχήμα 3.2

Στο Σχήματος 3.2 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα, τότε στέλνουμε ένα επιπλέον πακέτο αφού η κακόβουλη συσκευή μπορεί να καταστρέψει ένα επιπλέων πακέτο.

Αποτελέσματα για αλγόριθμο ADP_ CHARCHING (T,f) με f >1 για T = 2048
r = 0.0001

<u>Ωφέλιμο φορτίο – Goodput</u>

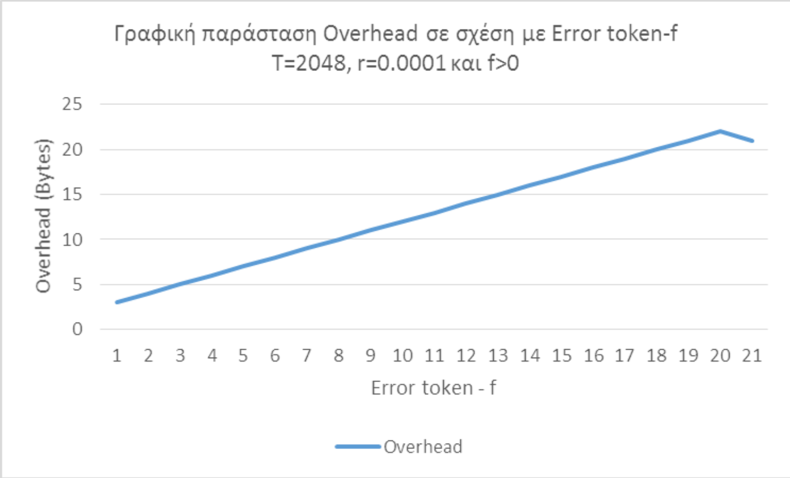


Σχήμα 3.3

Στο Σχήματος 3.3 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα *Error token*. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token* το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα error token. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με *f* >= 1). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή.

<u>Overhead</u>

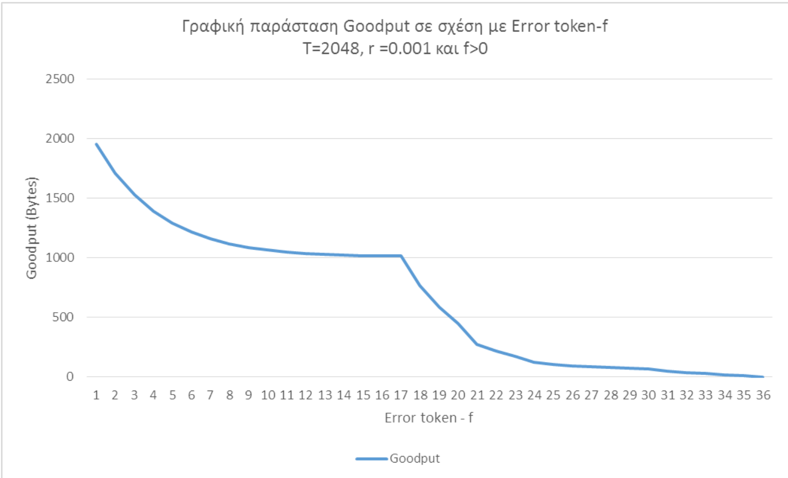
Στο Σχήματος 3.4 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό με γραμμική μορφή. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα, τότε στέλνουμε ένα επιπλέον πακέτο αφού η κακόβουλη συσκευή μπορεί να καταστρέψει ένα επιπλέων πακέτο. Επίσης ο λόγος που παρατηρείτε η μικρή μείωση του overhead όταν τα *Error token* από 20 γίνονται 21 είναι ότι σε αυτό το σημείο δεν χρησιμοποιείται καθόλου ο αλγόριθμος 1 γιατί τα *error token* στην τερματική περίπτωση είναι 2 (δηλαδή *f*>1).



Σχήμα 3.4

r = 0.001

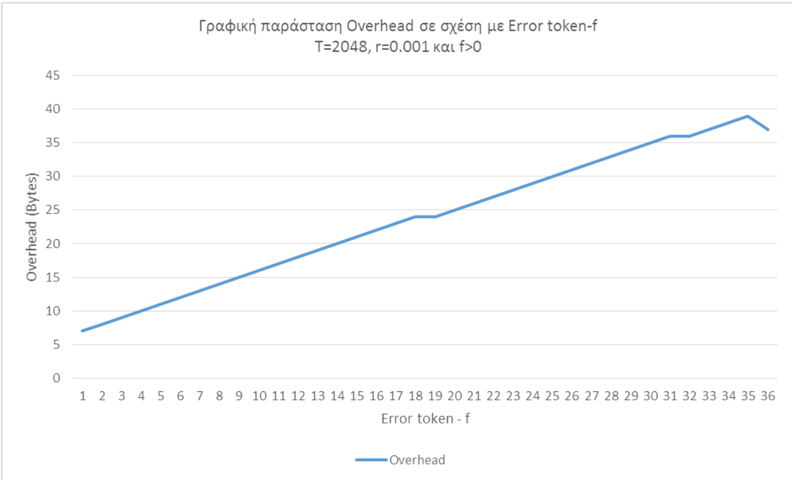
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.5

Στο Σχήματος 3.5 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα *Error token*. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα error token. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με *f* >= 1). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή. Επίσης εύκολα παρατηρούμε ότι στο σημείο όπου τα αρχικά *Error token*ισούνται με 18 ο ρυθμός μείωσης του goodput αλλάζει. Ας μην ξεχνάμε ότι ο αλγόριθμος 3 αποτελεί την επέκταση του αλγόριθμου ADP2 που αποτελείται από δυο “υπό – αλγόριθμους ” (ADP1 και ADP2). Εξετάζοντας ποιος από τους δύο “υπό – αλγόριθμους ” τρέχει ανά πάσα χρονική στιγμή, παρατηρούμε ότι για $f \leq 17$, κατά τη νέα χρονική περίοδο (μετά την πρώτη) τρέχει μόνο ADP1. Αντίθετα για $f \geq 18$ κατά τη νέα χρονική περίοδο τρέχει αρχικά ο ADP2 μέχρι τα διαθέσιμα *error token*– $f \leq 1$ ώστε να καλεστεί ο ADP1.

Overhead



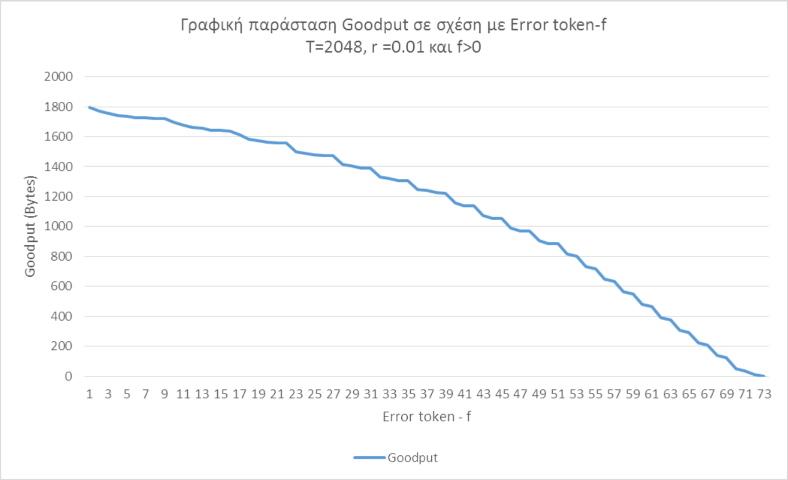
Σχήμα 3.6

Στο Σχήματος 3.6 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό με γραμμική μορφή. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα, τότε στέλνουμε ένα επιπλέον πακέτο αφού η κακόβουλη συσκευή μπορεί να καταστρέψει ένα επιπλέων πακέτο. Τα σημεία στα οποία το overhead παραμένει σταθερό, οφείλονται στο γεγονός ότι αυξάνοντας το μέγιστο αριθμό *error token- f* , κατά τη νέα χρονική περίοδο θα καλεστεί αρχικά διαφορετικός “υπό – αλγόριθμους ” (ADP1 ή ADP2). Έτσι το ο αριθμός των πακέτων που θα σταλούν διαφέρει. Επίσης ο λόγος που παρατηρείτε η μικρή μείωση του overhead όταν τα *Error token* από 35 γίνονται 36 είναι ότι σε αυτό το σημείο δεν χρησιμοποιείται καθόλου ο αλγόριθμος 1 γιατί τα *error token* στην τερματική περίπτωση είναι 2 (δηλαδή $f > 1$).

r = 0.01

Ωφέλιμο φορτίο – Goodput

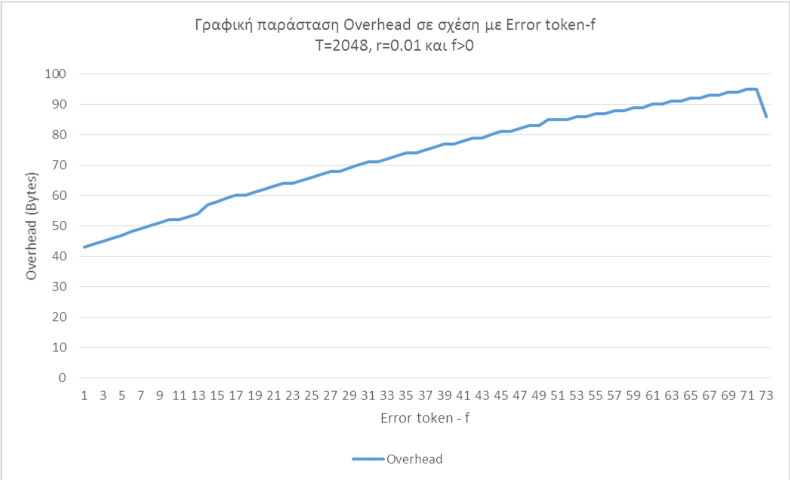
Στο Σχήματος 3.7 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα *Error token*. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο



Σχήμα 3.7

υπάρχουν διαθέσιμα *error token*. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με $f \geq 1$). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή. Επίσης εύκολα παρατηρούμε ότι η γραφική παράσταση είναι κάπως "κυματιστή". Όπως έχουμε ήδη εξηγήσει προηγούμενος αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος μας καλεί τον ADP1 όταν $f \leq 1$ ή ADP2 όταν $f \geq 1$. Επίσης αυξάνοντας το f τότε ο ADP2 καλείτε πιο συχνά.

Overhead

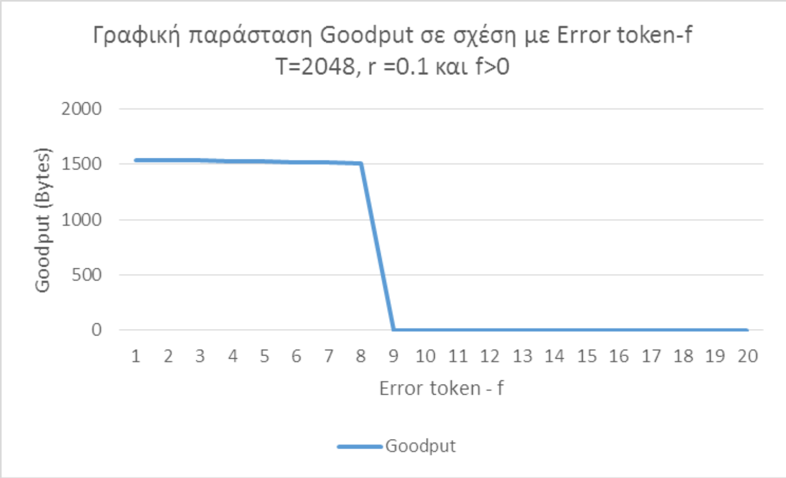


Σχήμα 3.8

Στο Σχήματος 3.8 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το f, το overhead αυξάνεται και αυτό σχετικά γραμμικά. Αυτό οφείλεται στο γεγονός ότι ο ρυθμός επαναφόρτισης μας είναι αρκετά μικρός. Επομένως έχουμε μικρές περιόδους με συχνή επαναφόρτιση. Επίσης ο λόγος που παρατηρείτε η μικρή μείωση του overhead όταν τα *Error token* από 72 γίνονται 73 είναι ότι σε αυτό το σημείο δεν χρησιμοποιείται καθόλου ο ADP1 γιατί τα *error token* στην τερματική περίπτωση είναι 2 (δηλαδή *f*>1).

r = 0.1

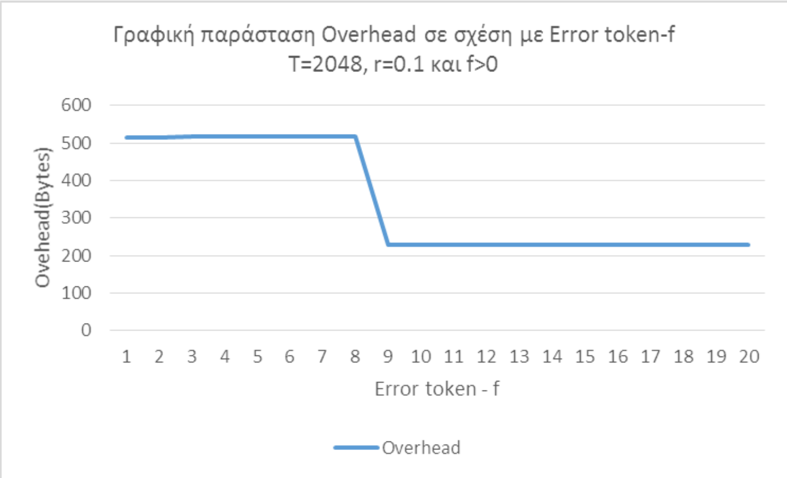
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.9

Στο Σχήματος 3.9 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα *Error token*. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης λόγω του ότι ο ρυθμός επαναφόρτισης είναι αρκετά μικρός για *f* ≥ 9, η κακόβουλη συσκευή διαθέτη αρκετά *error token*ώστε να καταστρέψει ολόκληρο το διαθέσιμο φορτίο. Επίσης από αυτό το σημείο και μετά καλείται μόνο ο ADP2, λόγο του μικρού ρυθμού επαναφόρτισης.

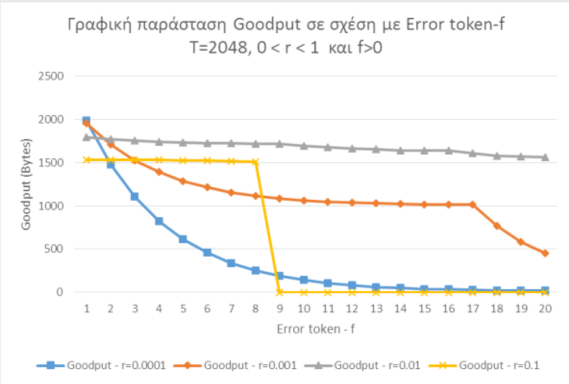
Overhead



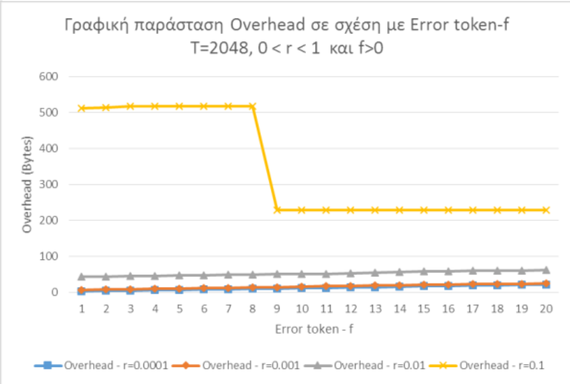
Σχήμα 3.10

Στο Σχήματος 3.10 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό σχετικά γραμμικά με πολύ αργό ρυθμό. Επίσης για *f* ≥ 9 παρατηρούμε μια απότομη πτώση στο overhead. Αυτό εν μέρει είναι συνέπεια του μικρού ρυθμού επαναφόρτισης. Για να γίνουμε πιο συγκεκριμένη ο αλγόριθμος αντιλαμβάνεται ότι για τη συγκεκριμένη χρονική περίοδο ακόμα και να στείλει πακέτα μικρότερου μεγέθους η κακόβουλη συσκευή μπορεί να τα παρεμβάλει. Έτσι στέλνει λιγότερα πακέτα.

6.2.1.1.2.5 Σύγκριση όλων των r



Σχήμα 3.11



Σχήμα 3.12

Με την βοήθεια των πιο πάνω σχημάτων ευκολά βλέπουμε ότι όταν ο ρυθμός επαναφόρτισης είναι 10 (r = 0.01), έχουμε τα καλύτερα αποτελέσματα όσο αφορά το goodput. Επίσης παρατηρούμε ότι δεν έχουμε ραγδαία αύξηση του overhead. Συνεπώς πρέπει ο ρυθμός επαναφόρτισης είναι πολύ σημαντικός έτσι ώστε ο αλγόριθμος μας να είναι αποδοτικός.

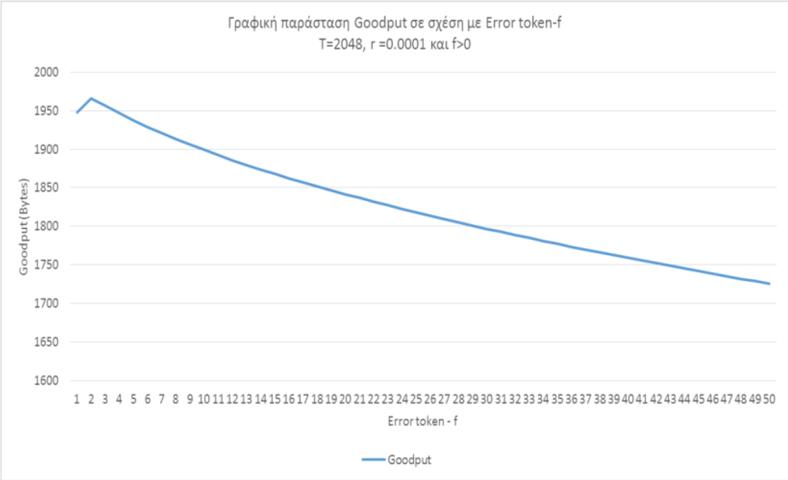
Αποτελέσματα για Uniform για T = 2048

r = 0.0001

Σημείωση: Για το σενάριο αυτό θέσαμε το f = 1…500, έτσι ώστε να ήμαστε σίγουρη για τον τρόπο που ανταποκρίνεται ο αλγόριθμος, ως προς τις 2^{ο} μετρικές. Για σκοπούς παρουσίασης έχουμε δώσει γραφικές με μικρότερες τιμές για το f, έτσι ώστε να γίνει καλύτερη κατανόηση της μορφής της γραφικής (μετά από ένα σημείο οι γραφικές γίνονται εντελώς γραμμικές).

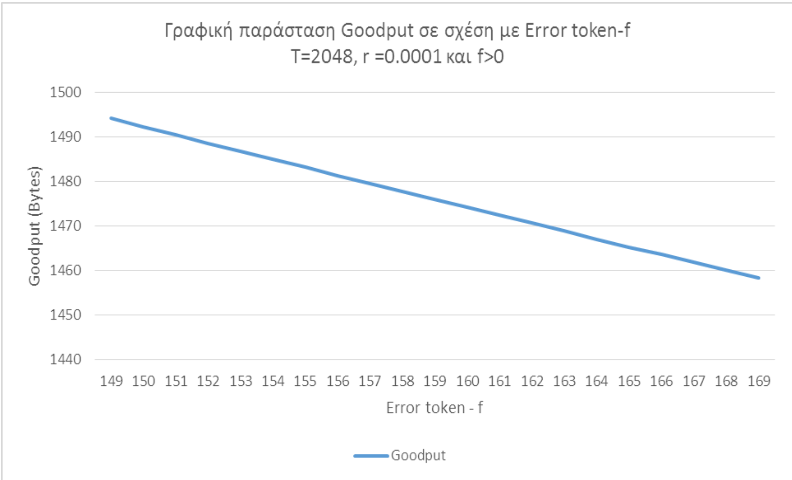
Ωφέλιμο φορτίο – Goodput

Στο Σχήματος 3.13 παρουσιάζουμε τον όγκο των δεδομένων που Στο Σχήματος 3.14



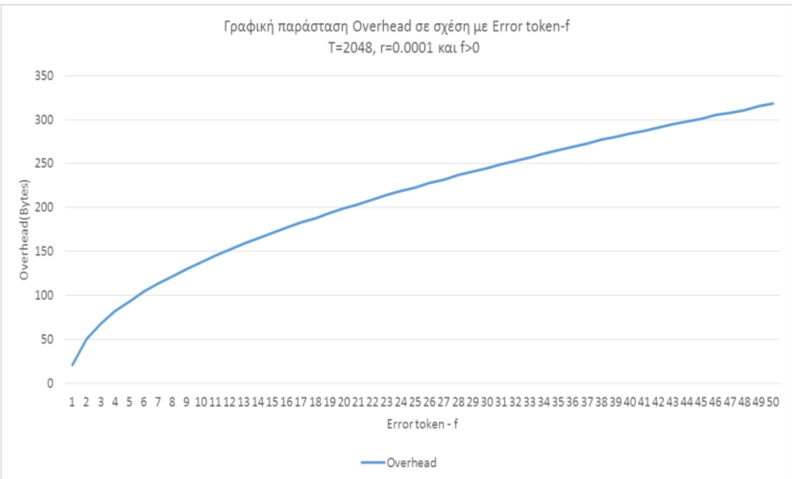
Σχήμα 3.13

παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Για να ήμαστε πιο ακριβείς όταν αυξήσουμε αρκετά το *f*, τότε η μείωση γίνεται γραμμική όπως μπορούμε να δούμε από το σχήμα 3.14.



Σχήμα 3.14

Overhead

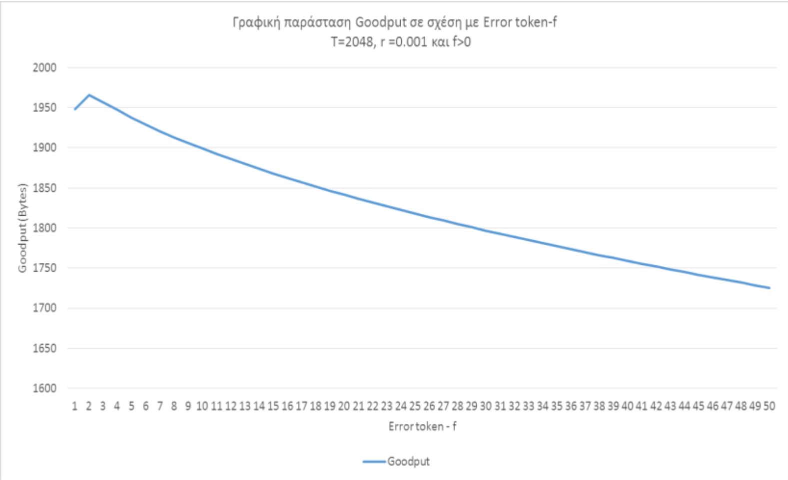


Σχήμα 3.15

Στο Σχήματος 3.15 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό σχεδόν γραμμικά. Αυξάνοντας την τιμή του *f*, το overhead αυξάνετε και αυτό γραμμικά (μετά από ένα σημείο), αυτό μπορούμε να διακρίνουμε και από την πιο πάνω γραφική παράσταση.

r = 0.001

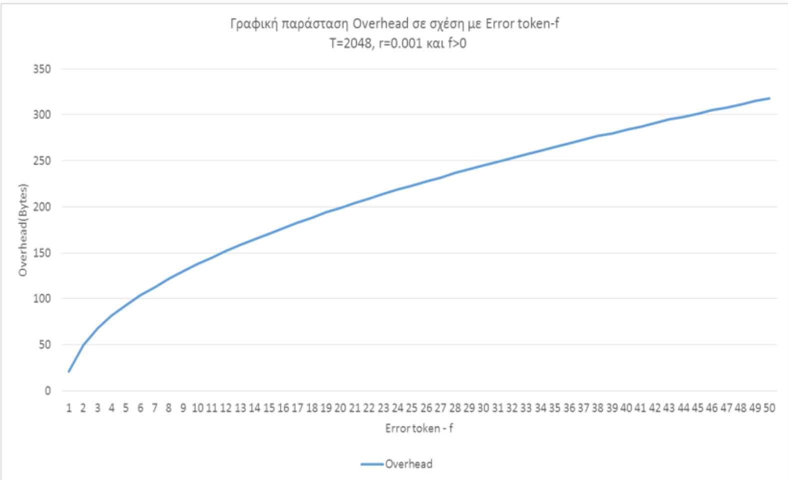
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.16

Στο Σχήματος 3.16 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token* το goodput μειώνεται με πιο αργούς ρυθμούς. Για να ήμαστε πιο ακριβείς όταν αυξήσουμε αρκετά το *f*, τότε η μείωση γίνεται γραμμική.

<u>Overhead</u>

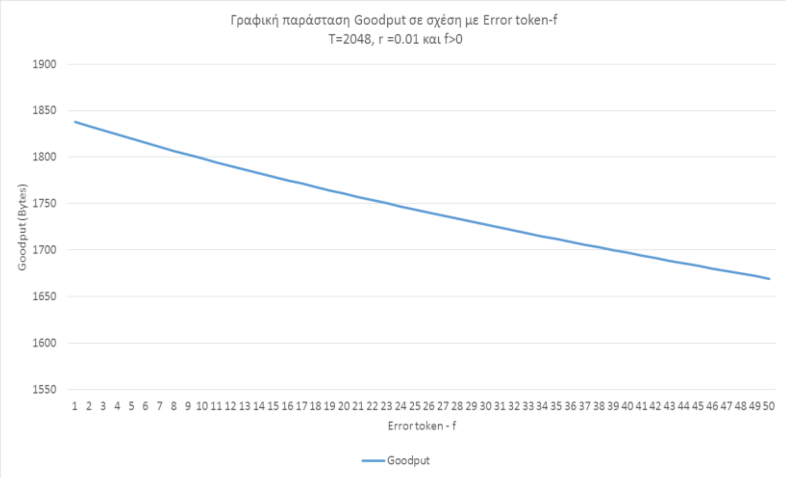


Σχήμα 3.17

Στο Σχήματος 3.17 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό σχεδόν γραμμικά. Αυξάνοντας την τιμή του *f*, το overhead αυξάνετε και αυτό γραμμικά (μετά από ένα σημείο), αυτό μπορούμε να διακρίνουμε και από την πιο πάνω γραφική παράσταση.

r = 0.01

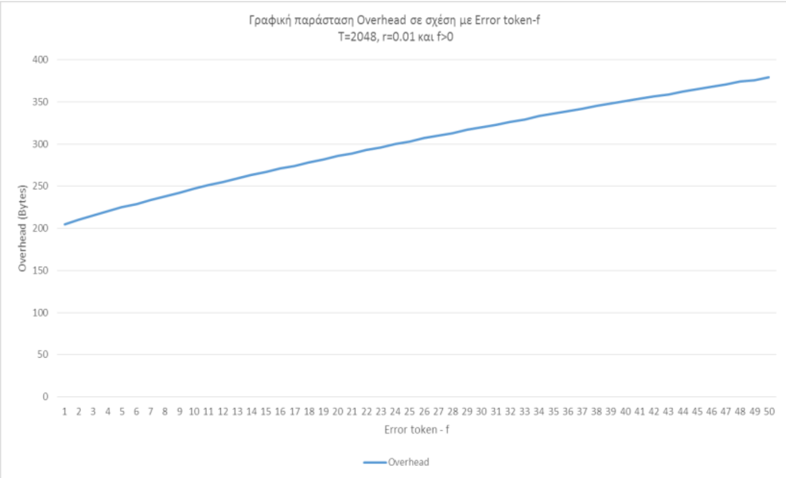
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.18

Στο Σχήματος 3.18 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι γραμμική.

<u>Overhead</u>



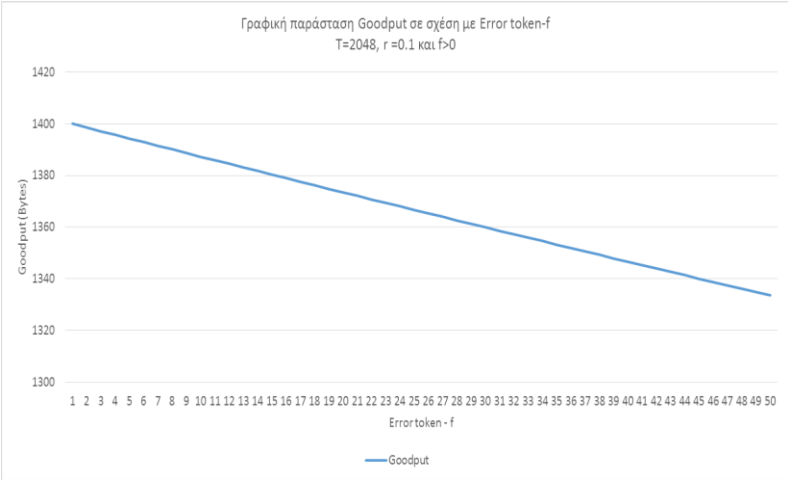
Σχήμα 3.19

Στο Σχήματος 3.19 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά.

r = 0.1

Ωφέλιμο φορτίο – Goodput

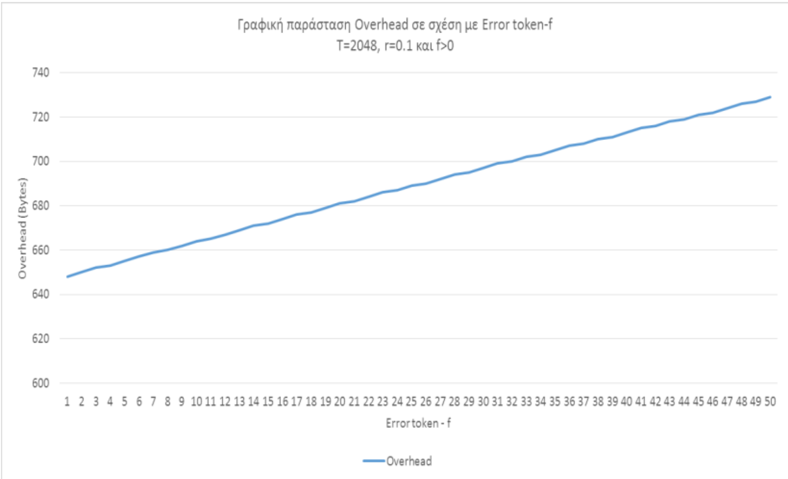
Στο Σχήματος 3.20 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput είναι γραμμική.



Σχήμα 3.20

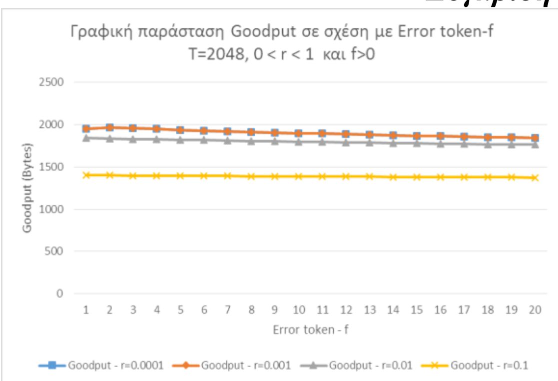
Overhead

Στο Σχήματος 3.21 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά.

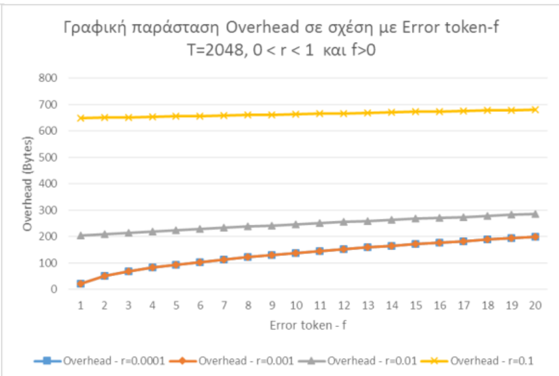


Σχήμα 3.21

Σύγκριση όλων των r



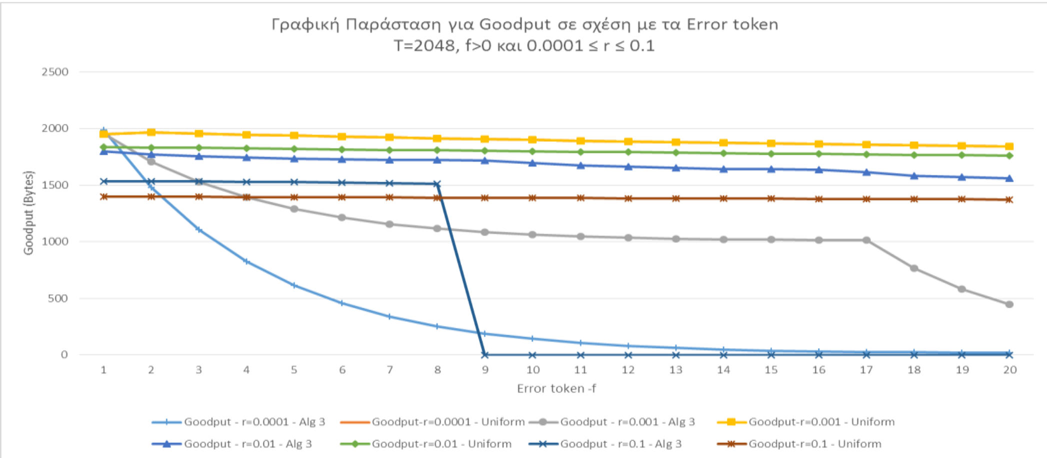
Σχήμα 3.22



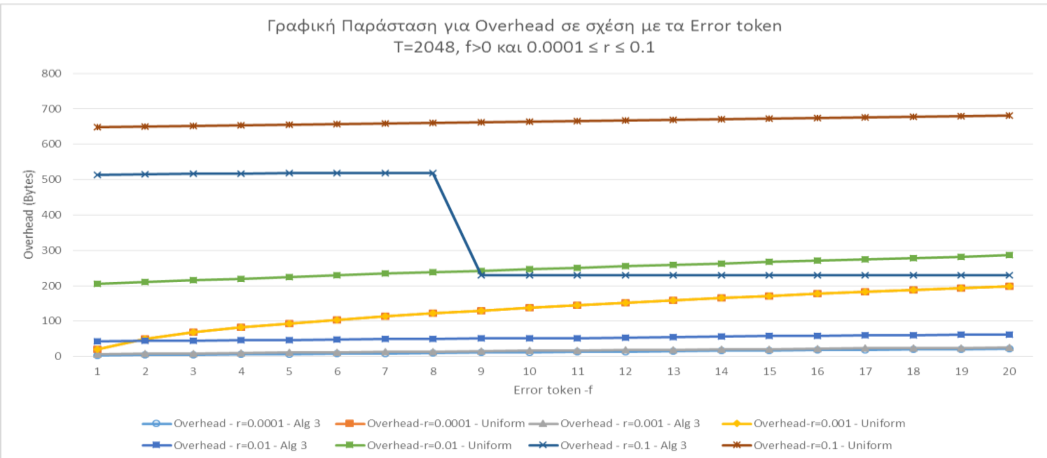
Σχήμα 3.23

Με την βοήθεια των πιο πάνω σχημάτων ευκολά βλέπουμε ότι όταν ο ρυθμός επαναφόρτισης δεν είναι πολύ μικρός, τότε δεν έχουμε σημαντική διαφορά όσο άφορά το goodput και το overhead.

Αλγόριθμος ADP_ CHARCHING (*T*,*f*) με f >1 VS Uni*f*orm – Γραφικές για T = 2048



Σχήμα 3.24



Σχήμα 3.25

Με τη βοήθεια των σχημάτων 3.24 και 3.25 μπορούμε να δούμε πως συμπεριφέρονται οι δύο αλγόριθμοι για τον ίδιο ρυθμό επαναφόρτισης της κακόβουλης συσκευής. Εύκολα παρατηρούμε ότι ο αλγόριθμος Uniform στέλνει περισσότερο χρήσιμο φορτίο, αλλά και περισσότερο επιπλέον φορτίο.

Γενικά Συμπεράσματα για T = 2048

Γραμμική αύξηση του *f* με πιθανότητα καταστροφής πακέτου 1				
	ADP (*T*,*1*)	**ADP (*T*,*f*) με f >1**	**ADP_ CHARCHING (T,f) με f >1**	**Uni*f*orm**
Goodput (bytes)		**Αύξηση** του ***f*** **Μείωση** του **goodput**	**Αύξηση** του ***f*** **Μείωση** του **goodput**	**Αύξηση** του ***f*** **Μείωση** του **goodput**
Overhead (bytes)		**Αύξηση** του ***f*** **Αύξηση** του **overhead**	**Αύξηση** του ***f*** **Αύξηση** του **overhead**	**Αύξηση** του ***f*** **Αύξηση** του **overhead**
Ρυθμό Επαναφόρτισης σε σχέση με το Goodput			*__Αύξηση__ του ***f*** **Μείωση** του **goodput**	**Αύξηση** του ***f*** **Μείωση** του **goodput**
Ρυθμό Επαναφόρτισης σε σχέση με το Overhead			**Αύξηση** του ***f*** **Αύξηση** του **overhead**	**Αύξηση** του ***f*** **Αύξηση** του **overhead**

Όρια Αλγορίθμων – Μέχρι πιο *f* = 2048 το goodput >0					
Ρυθμός επαναφόρτισης	**r = 0**	**r = 0.0001**	**r = 0.001**	**r = 0.01**	**r = 0.1**
Αλγόριθμος ADP (*T*,*f*) με *f* >1	*f* = 20				
	Overhead = 21				
	Goodput = 1				

Αλγόριθμος ADP_ CHARCHING (*T*,*f*) με *f* >1		*f* = 20	*f* = 35	*f* = 72	*f* = 8
		Overhead = 22	Overhead = 39	Overhead = 95	Overhead = 518
		Goodput = 21	Goodput = 12	Goodput = 9	Goodput = 1510
Uni*f*orm		*f* = 2047	*f* = 2047	*f* = 2047	*f* = 2047
		Overhead = 2048	Overhead = 2050	Overhead = 2068	Overhead = 2253
		Goodput = 0.20	Goodput = 0.05	Goodput = 0.48	Goodput = 0.32

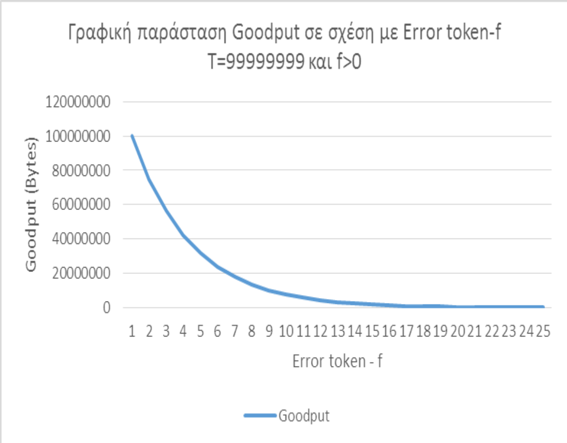
*Αν ο ρυθμός επαναφόρτισης επιλεγεί με προσοχή, μπορεί να μας αύξηση πάρα πολύ το Goodput, χωρίς σημαντική αύξηση του Overhead. Τελικά αυτό ήρθε σε αντίθεση με τις προβλέψεις μας. Όμως αν αναλογιστούμε ότι χωρίζουμε την αρχική μας περίοδο T σε μικρότερες περιόδους 1/r. Έτσι μειώνουμε το μήκος των πακέτων που στέλνονται, άρα καταστρέφεται λιγότερο φορτίο => περισσότερο ωφέλιμο φορτίο, με μικρή αύξηση του Overhead.

<u>Εν κατακλείδι:</u> Ο αλγόριθμος Uni*f*orm, αρχικά φαίνεται να υπερτερεί σε σχέση με τον αλγόριθμο . Όμως το εξαιρετικά μεγάλο Overhead του αλγορίθμου Uni*f*orm έρχεται να μετριάσει αυτή τη διαφορά (αν ο ρυθμός επαναφόρτισης της μπαταρίας επιλεγεί σωστά r=0,01 εδώ). Φυσικά αυτό ισχύει αν τα διαθέσιμα *Error token – f* είναι σχετικά μικρά, όπως μπορούμε να δούμε από τον πίνακα της προηγούμενης σελίδας.

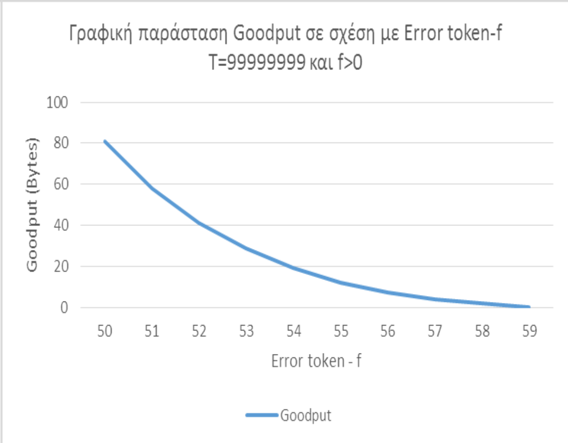
Αποτελέσματα για αλγόριθμο ADP(*T*,*f*) με f >1 για T = 99999999

<u>Ωφέλιμο φορτίο – Goodput</u>

Στο Σχήματος 3.26 & 3.27 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα *Error token*. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό μπορούμε να το αντιληφθούμε καλύτερα με τη βοήθεια του Σχήματος 3.27. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα error token. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με f >= 1). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή. Επίσης όταν το f = 59, ο αλγόριθμος ADP1 δεν χρησιμοποιείται, γιατί καταστρέφεται όλο το φορτίο χωρίς το f να πάρει την τιμή 1.

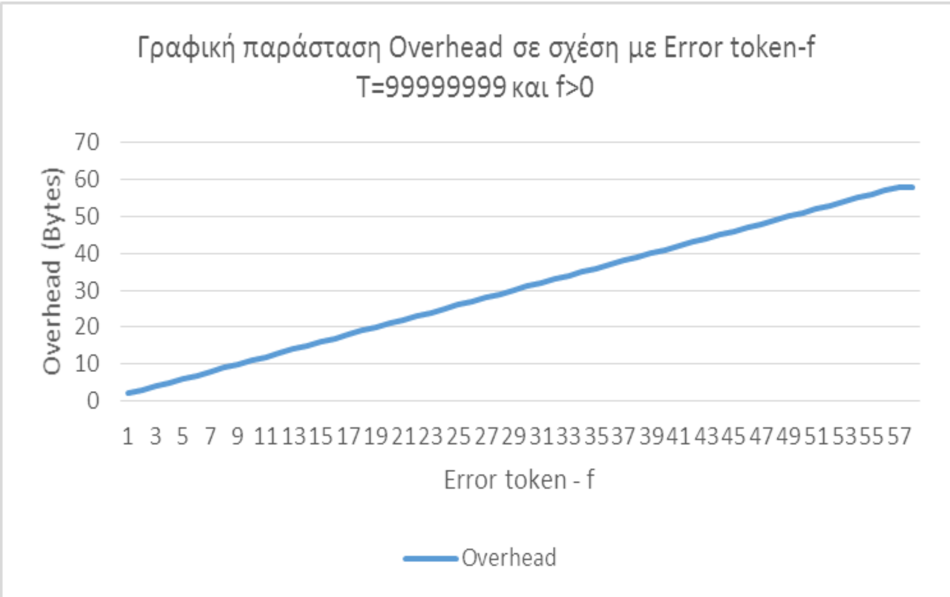


Σχήμα 3.26



Σχήμα 3.27

<u>**Overhead**</u>

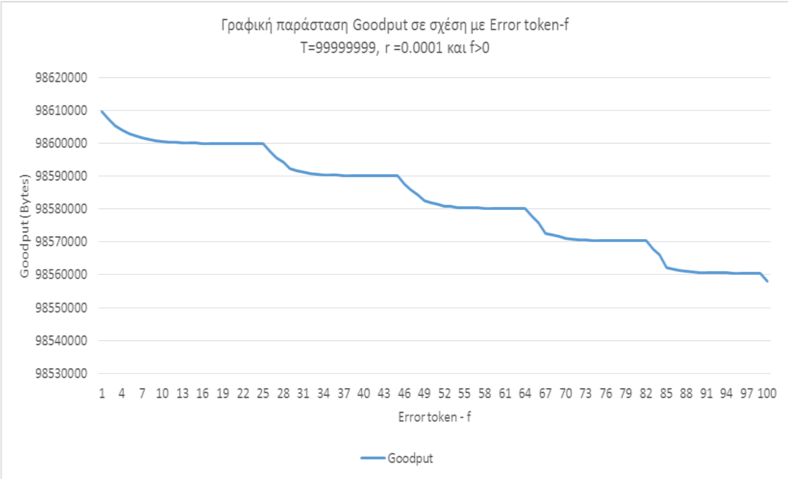


Σχήμα 3.28

Στο Σχήματος 3.28 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το f, το overhead αυξάνεται και αυτό γραμμικά. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα, τότε στέλνουμε ένα επιπλέον πακέτο αφού η κακόβουλη συσκευή μπορεί να καταστρέψει ένα επιπλέων πακέτο.

Αποτελέσματα για αλγόριθμο ADP_ CHARCHING(*T*,*f*) με f >1 για T = 99999999 r = 0.0001

<u>Ωφέλιμο φορτίο – Goodput</u>



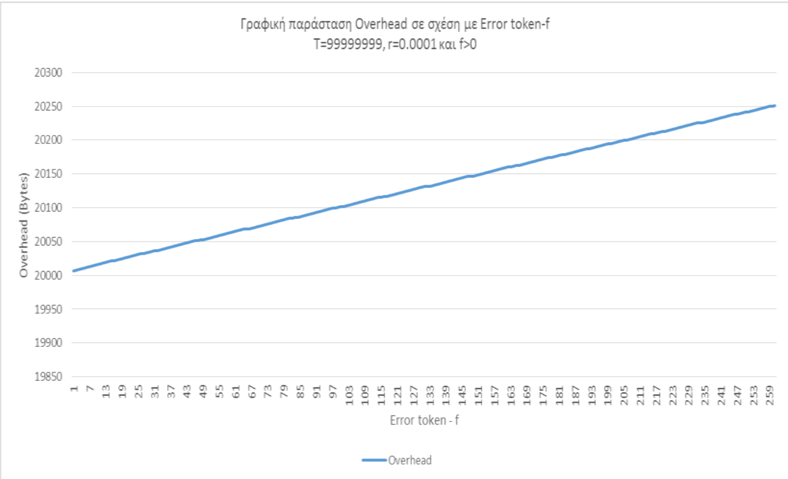
Σχήμα 3.29

Στο Σχήματος 3.29 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα *error token*. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token* το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα error token. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με *f* >= 1). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή. Επίσης εύκολα παρατηρούμε ότι στο σημείο όπου τα αρχικά *Error token* από 25 γίνονται 26 ο ρυθμός μείωσης του goodput αλλάζει. Ας μην ξεχνάμε ότι ο ADP_ CHARCHING (*T*,*f*) με *f* >1 αποτελεί την επέκταση του αλγόριθμου 2 που αποτελείται από δυο “υπό – αλγόριθμους ” (ADP1 και ADP2). Εξετάζοντας ποιος από τους δύο “υπό – αλγόριθμους ” τρέχει ανά πάσα χρονική στιγμή, παρατηρούμε ότι για $f \leq 25$, κατά τη νέα χρονική περίοδο (μετά την πρώτη) τρέχει μόνο ADP1. Αντίθετα για $f \geq 26$ κατά τη νέα χρονική περίοδο τρέχει αρχικά ο ADP2 μέχρι τα διαθέσιμα *error token* – $f \leq 1$

ώστε να καλεστεί ο ADP1. Αυτός είναι και ο λόγος που η γραφική μας παράσταση έχει την πιο πάνω μορφή (σχήμα 3.29). Η μορφή αυτή συνεχίζεται μέχρι τα διαθέσιμα *error token– f* να μπορούν να καταστρέψουν όλο το διαθέσιμο φορτίο.

<u>Overhead</u>

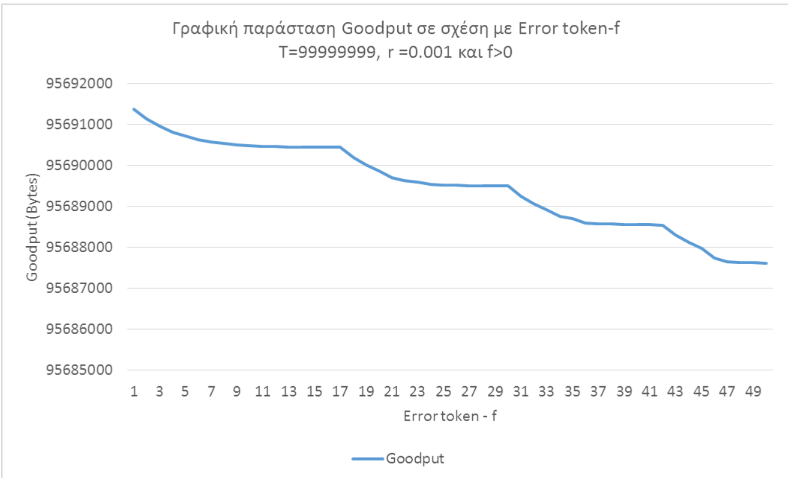
Στο Σχήματος 3.30 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό με γραμμική μορφή. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα, τότε στέλνουμε ένα επιπλέον πακέτο αφού η κακόβουλη συσκευή μπορεί να καταστρέψει ένα επιπλέων πακέτο.



Σχήμα 3.30

r = 0.001

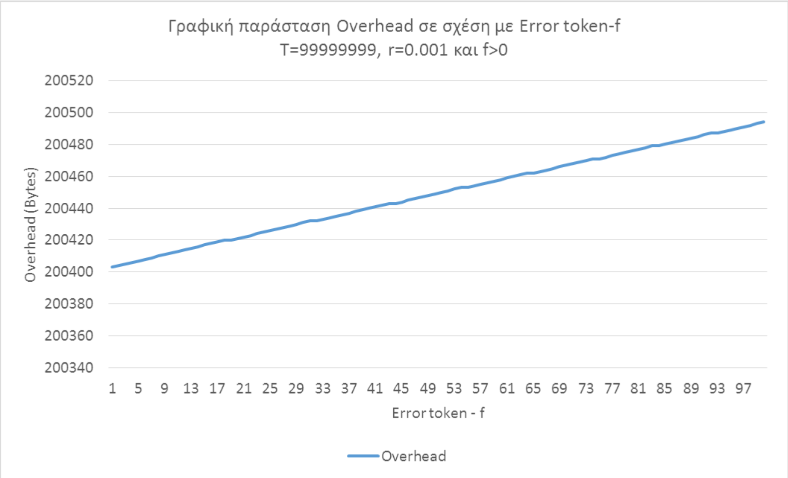
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.31

Στο Σχήματος 3.31 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token* το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα error token. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με $f >= 1$). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή. Επίσης εύκολα παρατηρούμε ότι στο σημείο όπου τα αρχικά *Error token* από 17 γίνονται 18 ο ρυθμός μείωσης του goodput αλλάζει. Ας μην ξεχνάμε ότι ο αλγόριθμος ADP_ CHARCHING (T,f) με $f > 1$ αποτελεί την επέκταση του αλγόριθμου ADP(T,f) με $f > 1$ που αποτελείται από δυο "υπό – αλγόριθμους " (ADP1 και ADP2). Εξετάζοντας ποιος από τους δύο "υπό – αλγόριθμους " τρέχει ανά πάσα χρονική στιγμή, παρατηρούμε ότι για $f \leq 17$, κατά τη νέα χρονική περίοδο (μετά την πρώτη) τρέχει μόνο ADP1. Αντίθετα για $f \geq 18$ κατά τη νέα χρονική περίοδο τρέχει αρχικά ο ADP2 μέχρι τα διαθέσιμα *error token*– $f \leq 1$ ώστε να καλεστεί ο ADP1. Αυτός είναι και ο λόγος που η γραφική μας παράσταση έχει την πιο πάνω μορφή (σχήμα 3.31). Η μορφή αυτή συνεχίζεται μέχρι τα διαθέσιμα *error token*– f να μπορούν να καταστρέψουν όλο το διαθέσιμο φορτίο.

Overhead

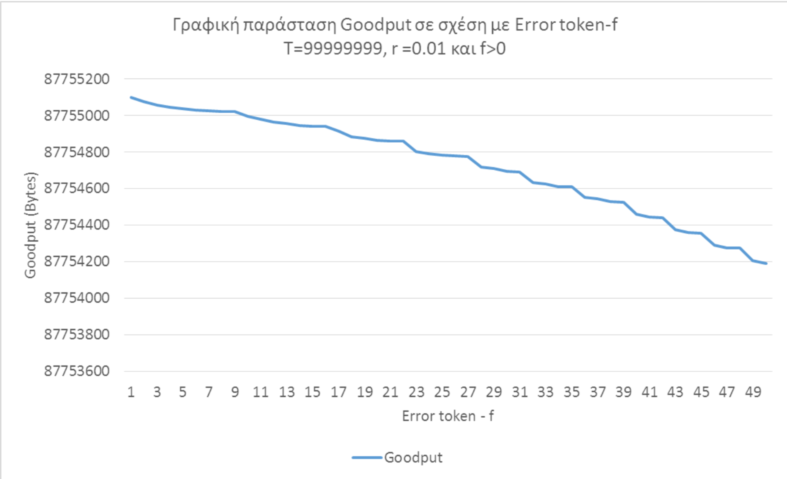


Σχήμα 3.32

Στο Σχήματος 3.32 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το f, το overhead αυξάνεται και αυτό με γραμμική μορφή. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το f κατά μία μονάδα, τότε στέλνουμε ένα επιπλέον πακέτο αφού η κακόβουλη συσκευή μπορεί να καταστρέψει ένα επιπλέων πακέτο.

r = 0.01

Ωφέλιμο φορτίο – Goodput

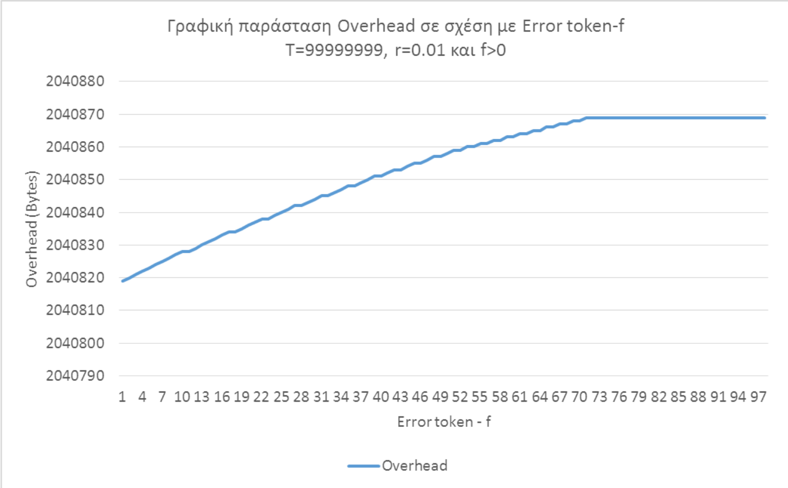


Σχήμα 3.32

Στο Σχήματος 3.32 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος είναι σχεδιασμένος έτσι ώστε να μειώνει το μέγεθος του πακέτου όσο υπάρχουν διαθέσιμα error token. Έτσι τα πρώτα πακέτα που στέλνονται είναι αρκετά μεγαλύτερα από τα τελευταία (με $f>=1$). Για αυτό το λόγο η γραφική μας παράσταση έχει αυτή τη μορφή. Επίσης εύκολα παρατηρούμε ότι στο σημείο όπου τα αρχικά *Error token* από 9 γίνονται 10 ο ρυθμός μείωσης του goodput αλλάζει. Ας μην ξεχνάμε ότι ο αλγόριθμος ADP_CHARCHING(T,f) με $f>1$ αποτελεί την επέκταση του αλγόριθμου ADP (T,f) με $f>1$ που αποτελείται από δυο “υπό – αλγόριθμους ” (ADP1 και ADP2). Εξετάζοντας ποιος από τους δύο “υπό – αλγόριθμους ” τρέχει ανά πάσα χρονική στιγμή, παρατηρούμε ότι για $f \leq 9$, κατά τη νέα χρονική περίοδο (μετά την πρώτη) τρέχει μόνο ADP1. Αντίθετα για $f \geq 10$ κατά τη νέα χρονική περίοδο τρέχει αρχικά ο ADP2 μέχρι τα διαθέσιμα *error token*– $f \leq 1$ ώστε να καλεστεί ο ADP1. Αυτός είναι και ο λόγος που η γραφική μας παράσταση έχει την πιο πάνω μορφή (σχήμα 3.32). Η μορφή αυτή συνεχίζεται μέχρι τα διαθέσιμα *error token* – $f \leq 71$. Ακολούθως η μείωση του goodput γίνεται γραμμική μέχρι $f \leq 98$. Για $f \geq 0$ goodput μηδενίζεται.

Overhead

Στο Σχήματος 3.33 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση.

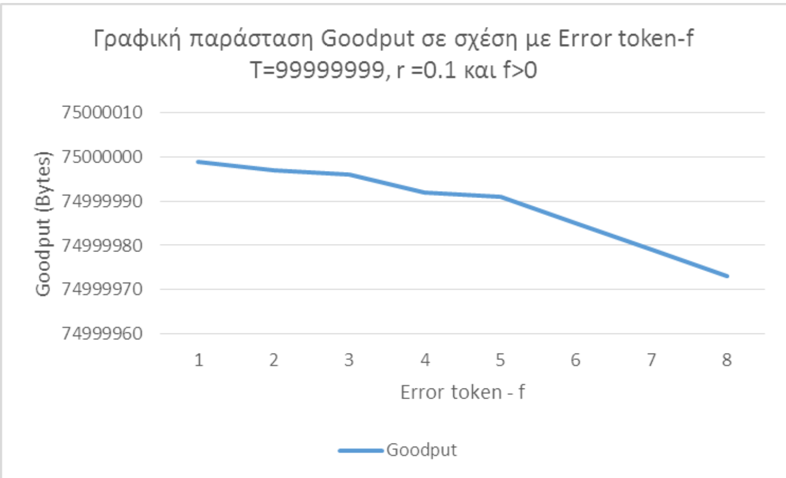


Σχήμα 3.33

Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό με γραμμική μορφή εκτός από ορισμένα σημεία στα οποία παραμένει σταθερό το overhead. Ας μην ξεχνάμε ότι ο αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 αποτελεί την επέκταση του αλγόριθμου ADP(*T*,*f*) με *f* >1 που αποτελείται από δυο "υπό – αλγόριθμους " (ADP1 και ADP2). Εξετάζοντας ποιος από τους δύο "υπό – αλγόριθμους " τρέχει ανά πάσα χρονική στιγμή. Επίσης για *f* = 99 όπου καταστρέφεται ολόκληρο το διαθέσιμο φορτίο, το overhead σταθεροποιείτε σε 1010102.

r = 0.1

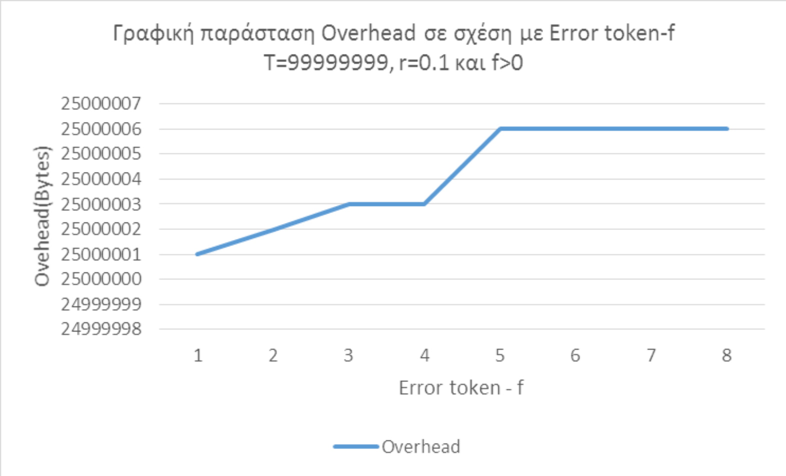
<u>Ωφέλιμο φορτίο – Goodput</u>



3.34

Στο Σχήματος 3.34 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης λόγω του ότι ο ρυθμός επαναφόρτισης είναι αρκετά μικρός για *f* ≥ 9, η κακόβουλη συσκευή διαθέτη αρκετά *error token* ώστε να καταστρέψει ολόκληρο το διαθέσιμο φορτίο. Επίσης από αυτό το σημείο και μετά καλείται μόνο ο ADP2, λόγο του μικρού ρυθμού επαναφόρτισης.

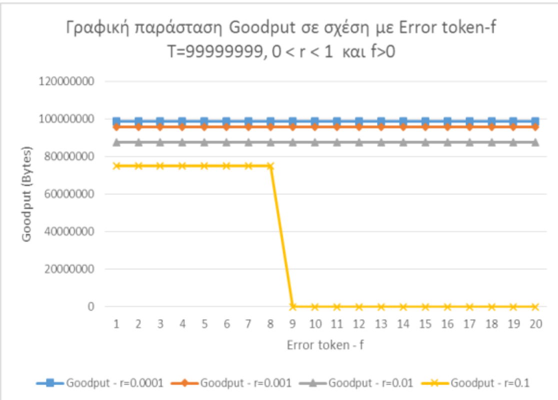
Overhead



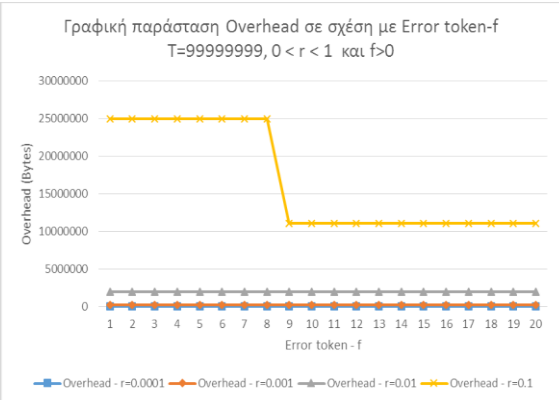
Σχήμα 3.35

Στο Σχήματος 3.35 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό με σχετικά με γραμμική μορφή εκτός από ορισμένα σημεία στα οποία παραμένει σταθερό το overhead. Ας μην ξεχνάμε ότι ο αλγόριθμος ADP_CHARCHING(T,f) με $f>1$ αποτελεί την επέκταση του αλγόριθμου ADP(T,f) με $f>1$ που αποτελείται από δυο "υπό – αλγόριθμους " (ADP1 και ADP2). Εξετάζοντας ποιος από τους δύο "υπό – αλγόριθμους " τρέχει ανά πάσα χρονική στιγμή. Επίσης για $f = 9$ όπου καταστρέφεται ολόκληρο το διαθέσιμο φορτίο, το overhead σταθεροποιείτε σε 11111112.

Σύγκριση όλων των r



Σχήμα 3.36



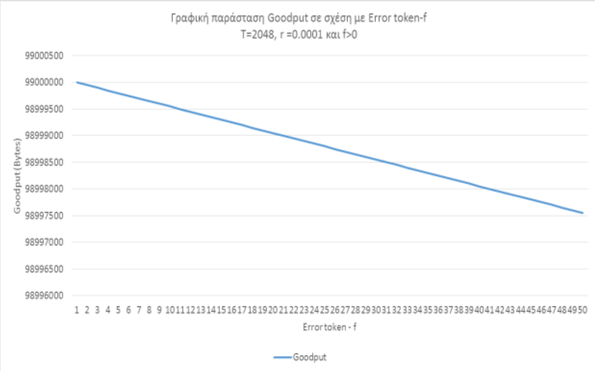
Σχήμα 3.37

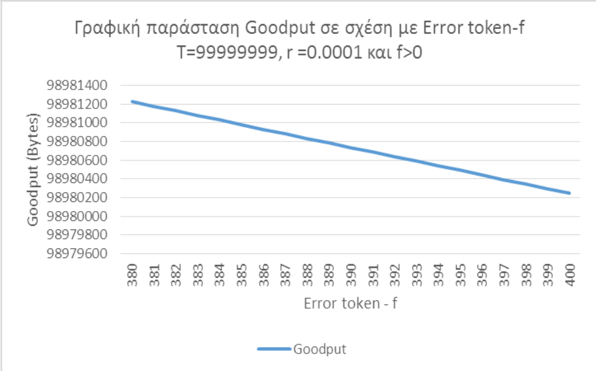
Με την βοήθεια των πιο πάνω σχημάτων ευκολά βλέπουμε ότι για μεγάλα φορτία, πιο αργή ρυθμοί επαναφόρτισης δίνουν καλύτερα αποτελέσματα για τον αλγόριθμο που μελετάμε.

Αποτελέσματα για Uniform για T = 99999999

r = 0.0001

<u>Ωφέλιμο φορτίο – Goodput</u>

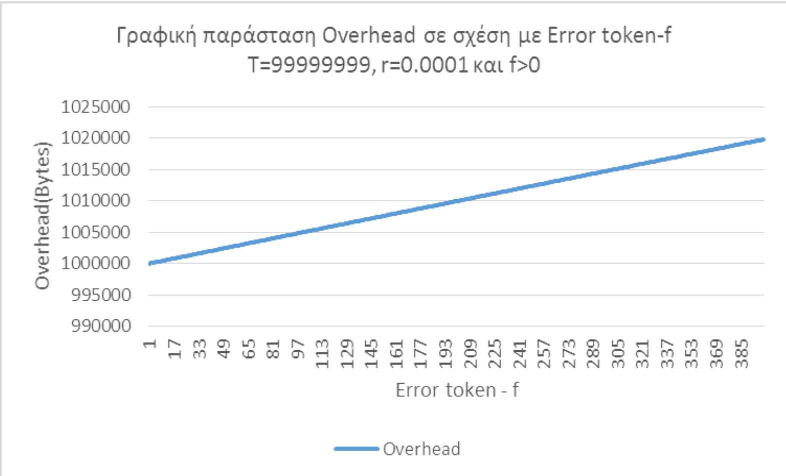




Σχήμα 3.37 Σχήμα 3.38

Στο Σχήματος 3.37 & Σχήμα 3.38 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται γραμμικά.

<u>Overhead</u>



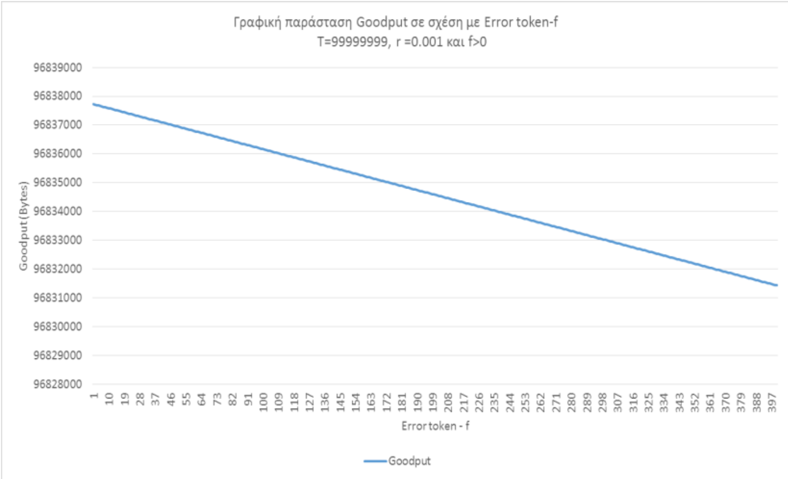
Σχήμα 3.39

Στο Σχήματος 3.39 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά.

r = 0.001

<u>Ωφέλιμο φορτίο – Goodput</u>

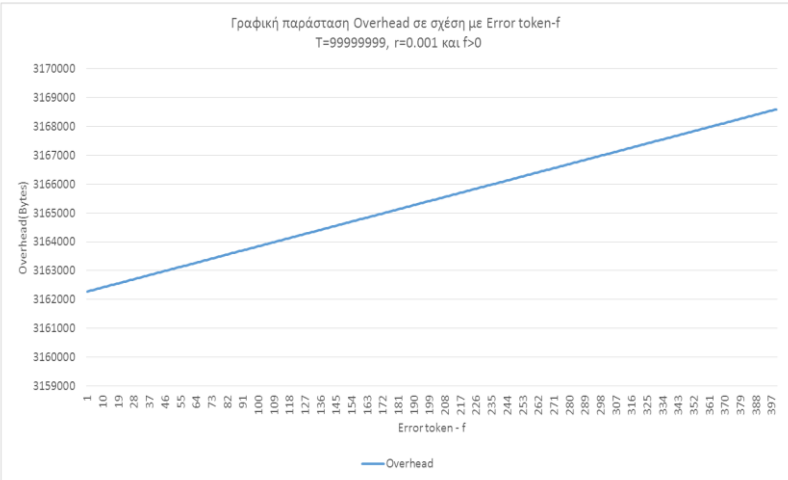
Στο Σχήμα 3.40 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα



Σχήμα 3.40

Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput είναι γραμμική.

<u>Overhead</u>

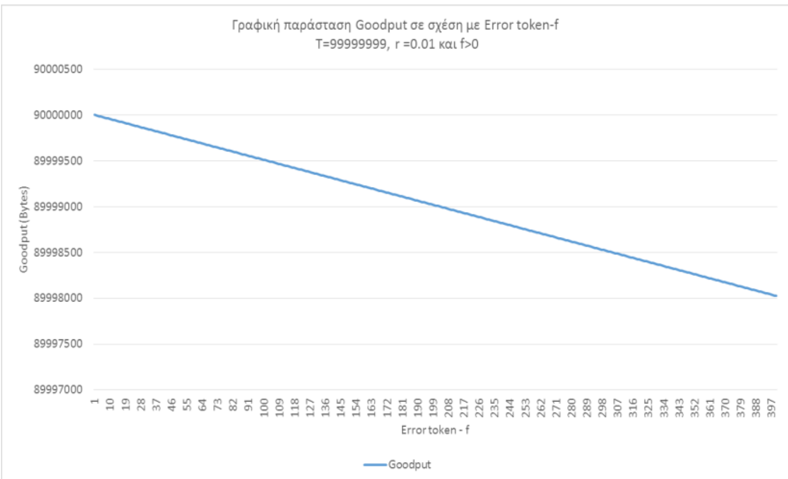


Σχήμα 3.41

Στο Σχήματος 3.41 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά.

r = 0.01

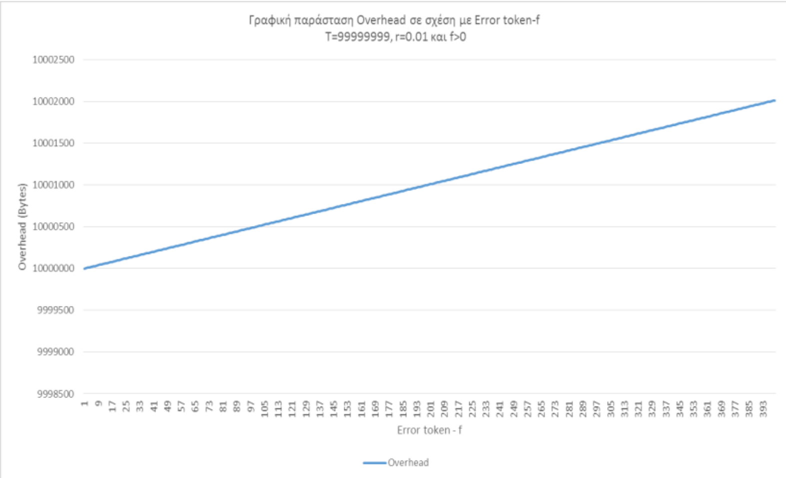
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.42

Στο Σχήμα 3.42 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput είναι γραμμική.

Overhead

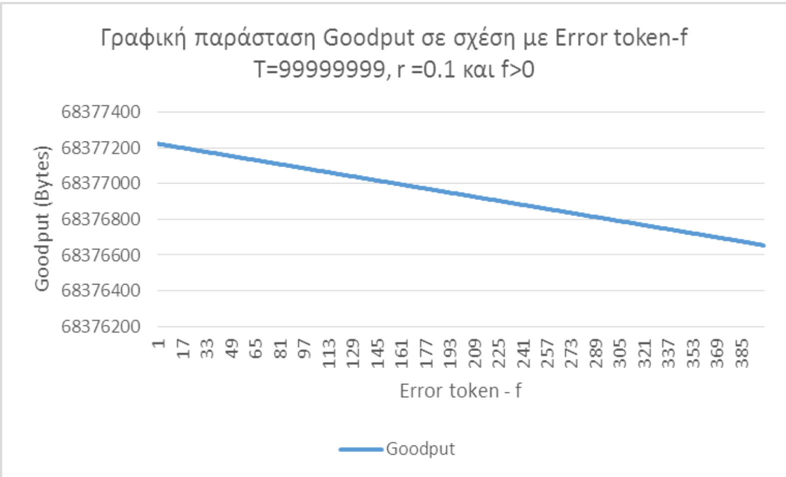


Σχήμα 3.43

Στο Σχήματος 3.43 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά.

r = 0.1

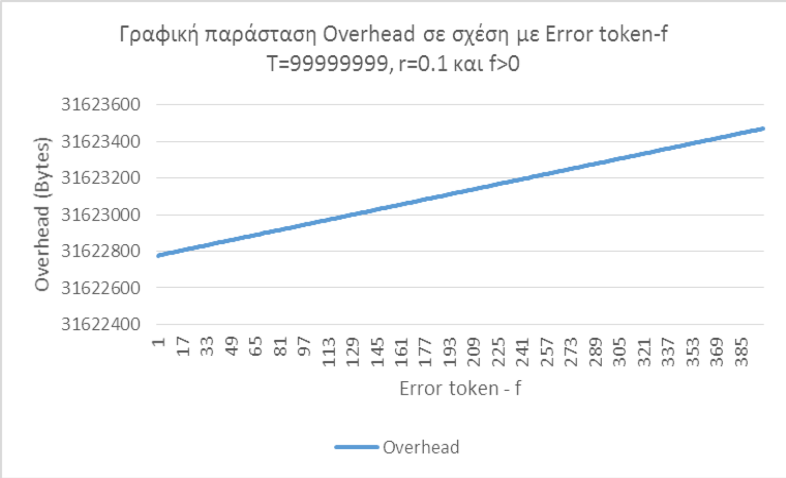
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.44

Στο Σχήμα 3.44 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput είναι γραμμική.

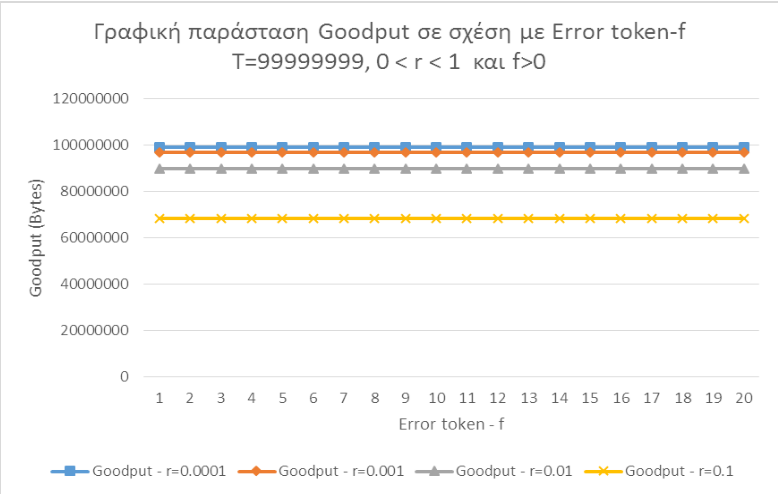
Overhead



Σχήμα 6.45

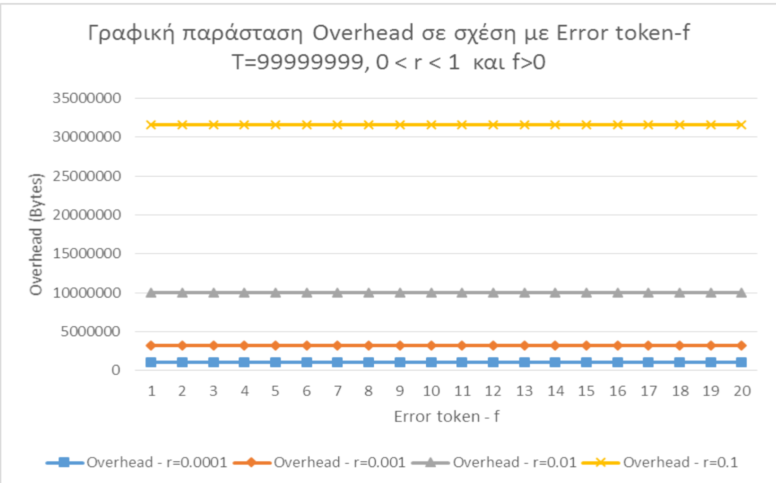
Στο Σχήματος 3.45 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά.

Σύγκριση όλων των r



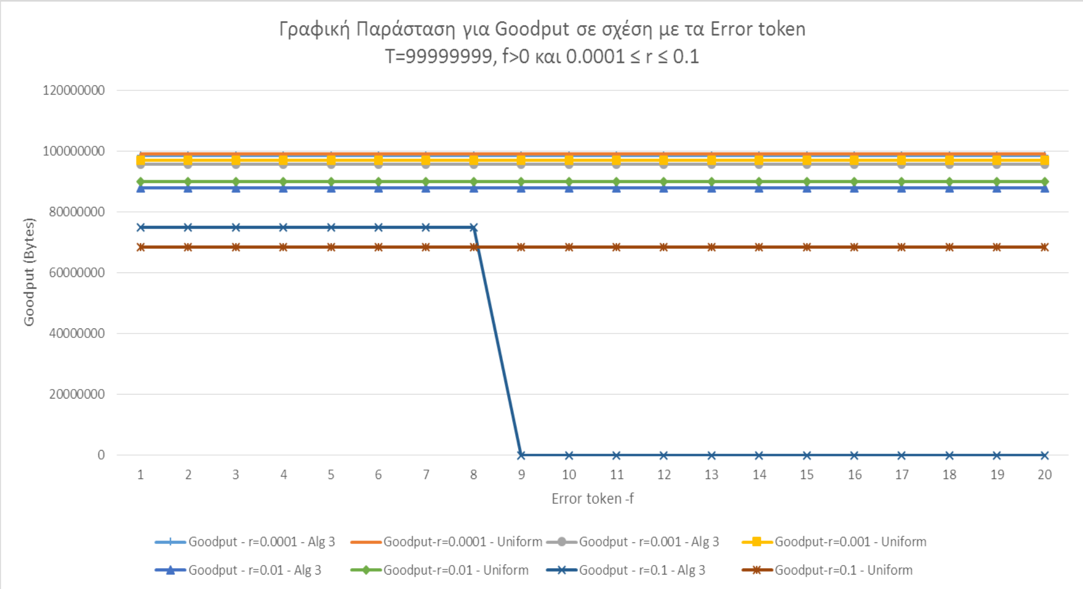
Σχήμα 3.46

Με την βοήθεια των πιο πάνω σχημάτων ευκολά βλέπουμε ότι όταν ο ρυθμός επαναφόρτισης δεν είναι πολύ μικρός, τότε δεν έχουμε σημαντική διαφορά όσο αφορά το goodput και το overhead.

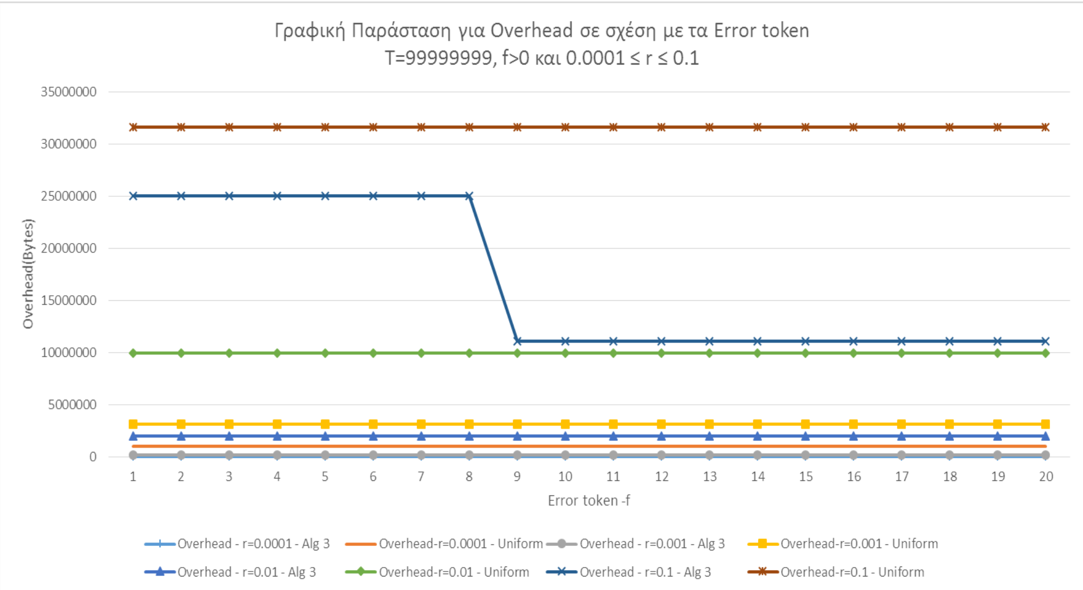


Σχήμα 3.47

Αλγόριθμος ADP_CHARCHING(T,f) με f >1 VS Uniform – Γραφικές για T = 9999999



Σχήμα 3.48



Σχήμα 3.49

Με τη βοήθεια των σχημάτων 3.48 και 3.49 μπορούμε να δούμε πως συμπεριφέρονται οι δύο αλγόριθμοι για τον ίδιο ρυθμό επαναφόρτισης της κακόβουλης συσκευής. Εύκολα παρατηρούμε ότι ο αλγόριθμος Uniform στέλνει περισσότερο χρήσιμο φορτίο, αλλά και περισσότερο επιπλέον φορτίο.

Γενικά Συμπεράσματα για T = 99999999

Γραμμική αύξηση του *f* με πιθανότητα καταστροφής πακέτου 1				
	Αλγόριθμος ADP(*T*,*1*)	Αλγόριθμος ADP(*T*,*f*) με *f* >1	Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1	Uniform
Goodput (bytes)		**Αύξηση** του ***f*** **Μείωση** του **goodput**	**Αύξηση** του ***f*** **Μείωση** του **goodput**	**Αύξηση** του ***f*** **Μείωση** του **goodput**
Overhead (bytes)		**Αύξηση** του ***f*** **Αύξηση** του **overhead**	**Αύξηση** του ***f*** **Αύξηση** του **overhead**	**Αύξηση** του ***f*** **Αύξηση** του **overhead**
Ρυθμό Επαναφόρτισης σε σχέση με το Goodput			****Αύξηση** του ***f*** **Μείωση** του **goodput**	**Αύξηση** του ***f*** **Μείωση** του **goodput**
Ρυθμό Επαναφόρτισης σε σχέση με το Overhead			**Αύξηση** του ***f*** **Αύξηση** του **overhead**	**Αύξηση** του ***f*** **Αύξηση** του **overhead**

Όρια Αλγορίθμων – Μέχρι πιο *f* το goodput >0					
Ρυθμός επαναφόρτισης	**r = 0**	**r = 0.0001**	**r = 0.001**	**r = 0.01**	**r = 0.1**
Αλγόριθμος ADP (*T*,*f*) με *f* >1	***f*** = 57				
	Overhead = 58				
	Goodput = 2				
Αλγόριθμος ADP_ CHARCHING (*T*,*f*) με *f* >1	***f***	9998	998	98	8
	Overhead	25382	200983	204069	25000006
	Goodput	53038551	95249729	87751036	74999973

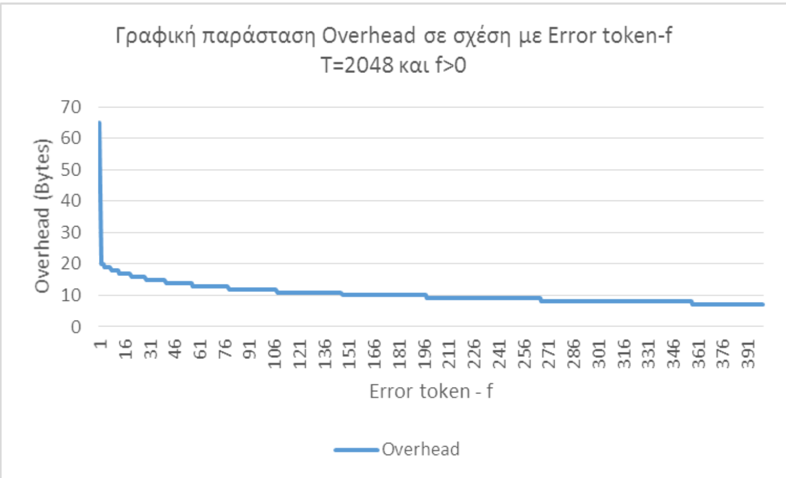
Uniform	***f***	99999999	100000000	99999999	100000000
	Overhead	100009999	100100000	100999999	110000000
	Goodput	0.65	0.16	0.59	0.1

<u>Εν κατακλείδι:</u> Ο αλγόριθμος Uniform και ο αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 για όχι πολύ μικρούς ρυθμούς επαναφόρτισης έχουν σχεδόν το ίδιο goodput, με το Uni*f*orm να υπερτερεί ελάχιστα. Όμως αυτή τη διαφορά έρχεται να την υπερκαλύψει το μεγαλύτερο Overhead που έχει ο Uniform συγκριτικά με τον αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1. Φυσικά αυτό ισχύει αν τα διαθέσιμα *Error token – f* είναι σχετικά μικρά, όπως μπορούμε να δούμε από τον πίνακα της προηγούμενης σελίδας. Από την άλλη όμως παρόλο που ο Uniform έχει πολύ μεγαλύτερο goodput για μεγάλες τιμές του *f*, το Overhead έρχεται σε σημείο να ξεπεράσει το συνολικό φορτίο που θέλουμε να στείλουμε.

Αποτελέσματα για σενάριο 2

Όπως έχουμε ήδη εξηγήσει μέσω αυτού του σεναρίου θέλουμε να δούμε το Overhead που προκαλείτε σε περίπτωση που η κακόβουλη συσκευή δεν καταστρέψει κανένα πακέτο.

Αποτελέσματα για αλγόριθμο ADP (*T*,*f*) με *f* >1 για T = 2048

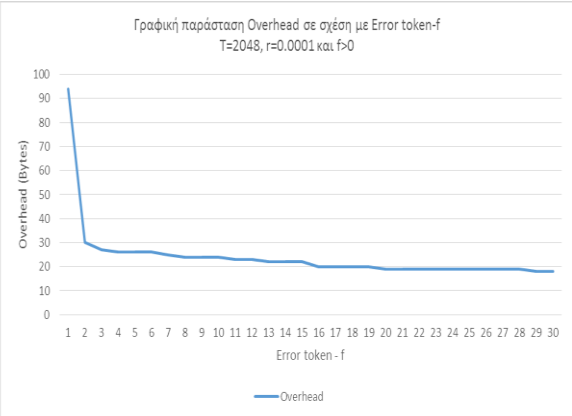


Σχήμα 3.50

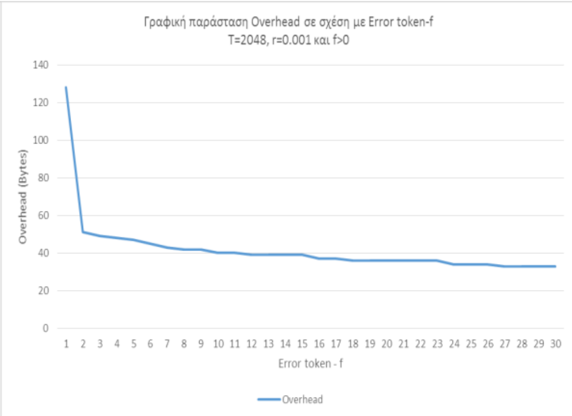
Στο Σχήματος 3.50 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead παραμένει σταθερό για κάποιο εύρος τιμών και μετά μειώνεται ελαφρός. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα και δεδομένου ότι το *f* έχει πάντα τη μέγιστη δυνατή τιμή του (αφού δεν καταστρέφεται κανένα πακέτο), τότε μπαίνει πιο γρήγορα στην τερματική περίπτωση όπου τα *error token*ξεπερνούν το φορτίο. Συνεπώς στέλνουν όλο το εναπομείναντα φορτίο => αποστολή λιγότερων πακέτων => λιγότερο Overhead. Επίσης η αρχική τιμή του Overhead είναι αρκετά μεγαλύτερη για $f = 1$. Αυτό οφείλεται στο γεγονός ότι μόνο σε αυτή την περίπτωση τρέχει ο αλγόριθμος ADP1.

Αποτελέσματα για αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 για T = 2048

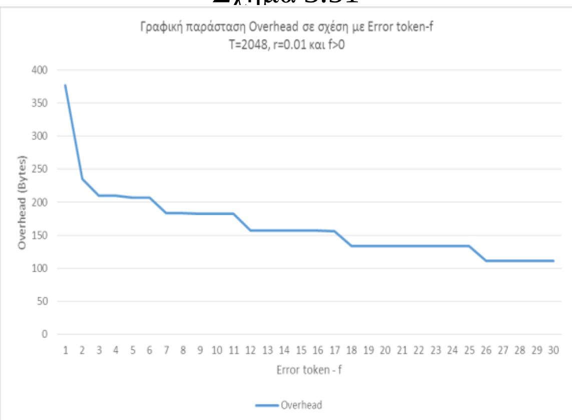
$0.1 \leq r \leq 0.0001$



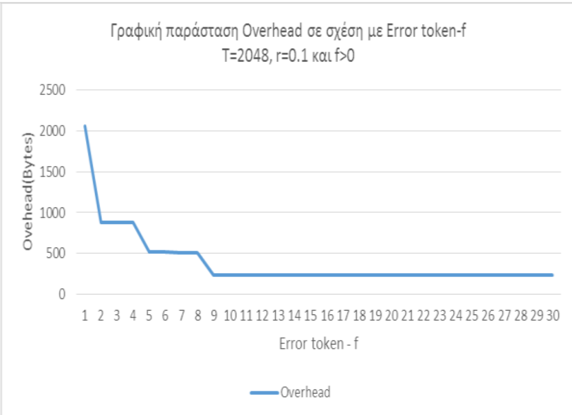
Σχήμα 3.51



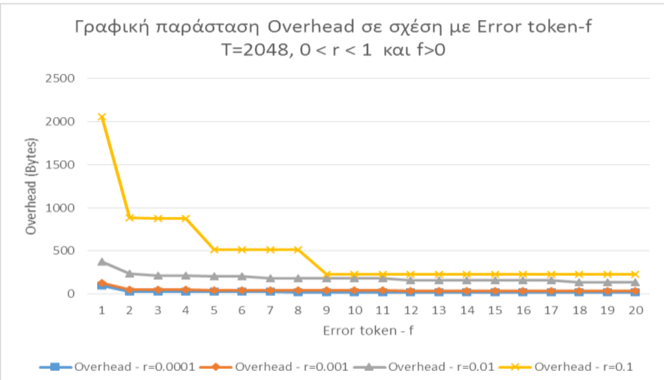
Σχήμα 3.52



Σχήμα 3.53



Σχήμα 3.54

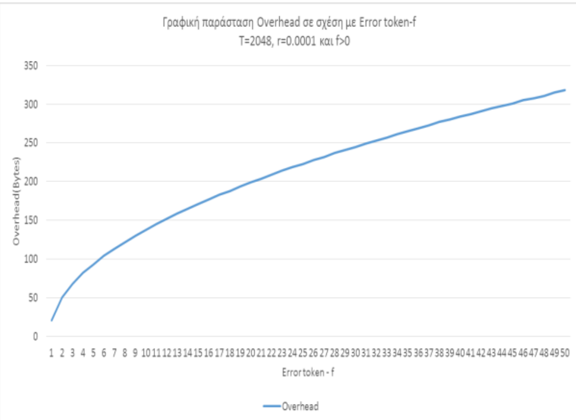


Σχήμα 3.55

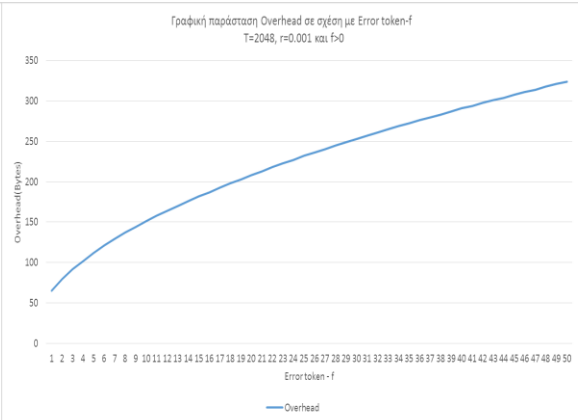
Στα Σχήματα 3.51, 3.52, 3.53, 3.54 και 3.55 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead παραμένει σταθερό για κάποιο εύρος τιμών και μετά μειώνεται ελαφρός. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα και δεδομένου ότι το *f* έχει πάντα τη μέγιστη δυνατή τιμή του (αφού δεν καταστρέφεται κανένα πακέτο), τότε μπαίνει πιο γρήγορα στην τερματική περίπτωση όπου τα *error token* ξεπερνούν το φορτίο. Συνεπώς στέλνουν όλο το εναπομείναντα φορτίο => αποστολή λιγότερων πακέτων => λιγότερο Overhead. Επίσης η αρχική τιμή του Overhead είναι αρκετά μεγαλύτερη για *f* = 1. Αυτό οφείλεται στο γεγονός ότι μόνο σε αυτή την περίπτωση τρέχει ο αλγόριθμος ADP1. Επίσης παρατηρούμε ότι η συμπεριφορά που περιγράψαμε πιο πάνω συμβαίνει ανεξαρτήτως του χρόνου επαναφόρτισης. Όμως όσο μικρότερος είναι ο χρόνος επαναφόρτισης, παρατηρούμε πιο γρήγορη σταθεροποίηση του overhead, όμως με μεγαλύτερες τιμές.

6.4.1.2.3 Αποτελέσματα για Uniform για T = 2048

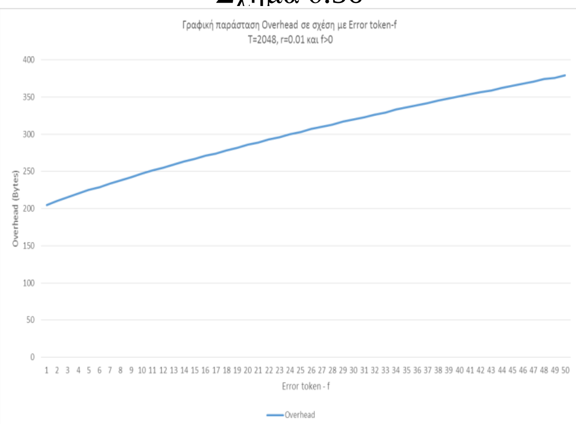
6.4.1.2.3.1 0.1 ≤ r ≤ 0.0001



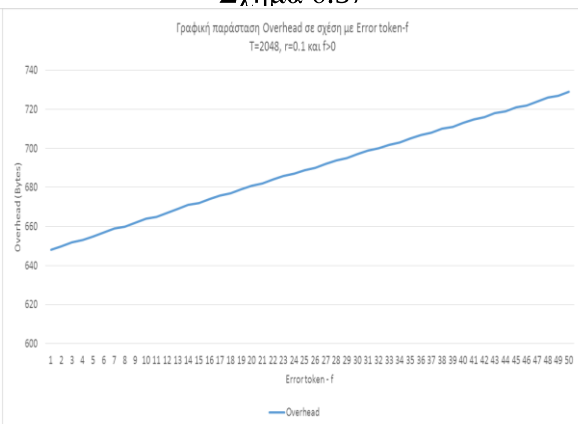
Σχήμα 6.56



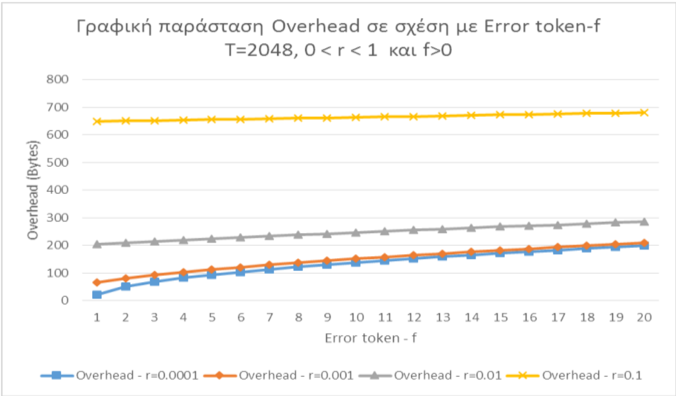
Σχήμα 6.57



Σχήμα 6.58



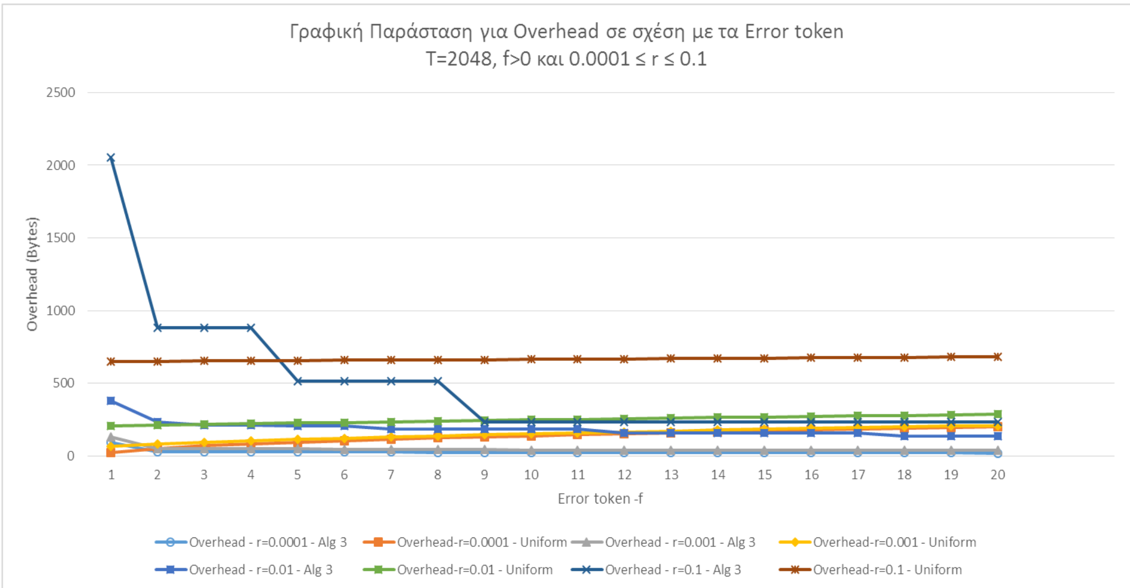
Σχήμα 6.59



Σχήμα 6.60

Στα Σχήματα 3.56, 3.57, 3.58, 3.59 και 3.60 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται αρχικά εκθετικά για μεγαλύτερες τιμές χρόνου επαναφόρτισης. Όμως στην συνέχεια αυξάνεται σχεδόν γραμμικά. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος κατά τον υπολογισμό του μήκους του πακέτου, ο αλγόριθμος λαμβάνει υπόψη όλες τις παραμέτρους. Επίσης αυτός είναι και ο λόγος που το overhead αυξάνετε.

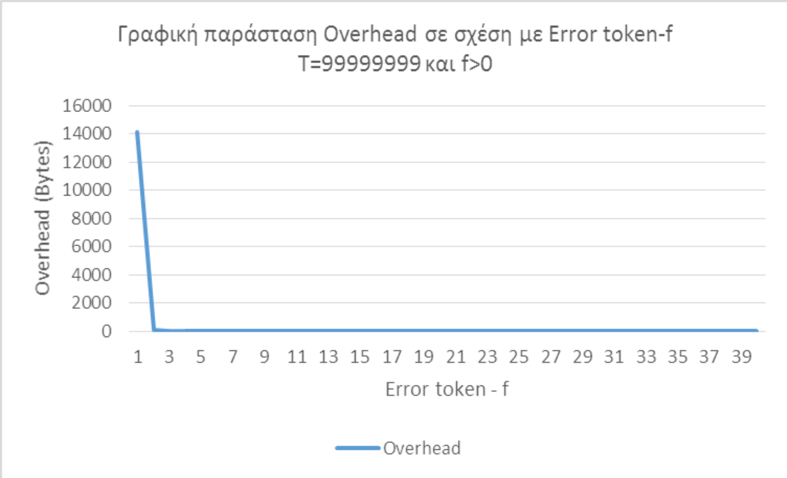
Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 VS Uniform – Γραφικές για T = 2048



Σχήμα 3.61

Με τη βοήθεια του σχήματος 3.61 μπορούμε να δούμε ότι για αυτό το σενάριο ο αλγόριθμος 3 υπερτερεί.

Αποτελέσματα για αλγόριθμο ADP(*T*,*f*) με *f* >1 για T = 99999999

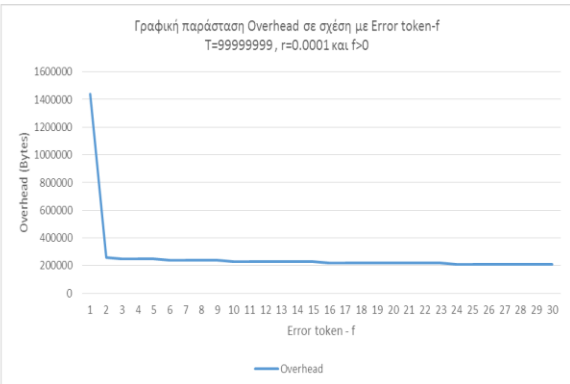


Σχήμα 3.62

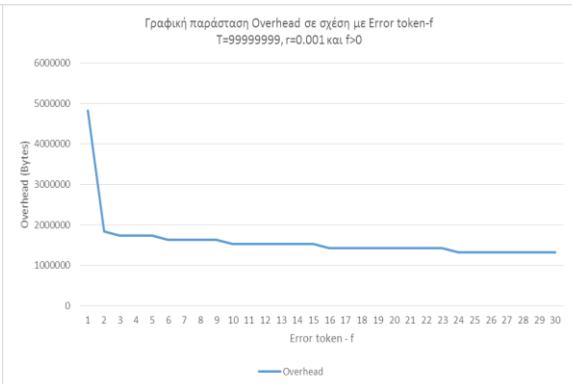
Στο Σχήματος 3.62 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead μειώνεται απότομα όταν το *f*=1, ενώ στην συνέχεια με πολύ αργό ρυθμό. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα και δεδομένου ότι το *f* έχει πάντα τη μέγιστη δυνατή τιμή του (αφού δεν καταστρέφεται κανένα πακέτο), τότε μπαίνει πιο γρήγορα στην τερματική περίπτωση όπου τα *error token*ξεπερνούν το φορτίο. Συνεπώς στέλνουν όλο το εναπομείναντα φορτίο => αποστολή λιγότερων πακέτων => λιγότερο Overhead. Επίσης η αρχική απότομη πτώση οφείλεται στο γεγονός ότι μόνο σε αυτή την περίπτωση τρέχει μόνο ο αλγόριθμος ADP1.

Αποτελέσματα για αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 για T = 99999999

0.1 ≤ r ≤ 0.0001

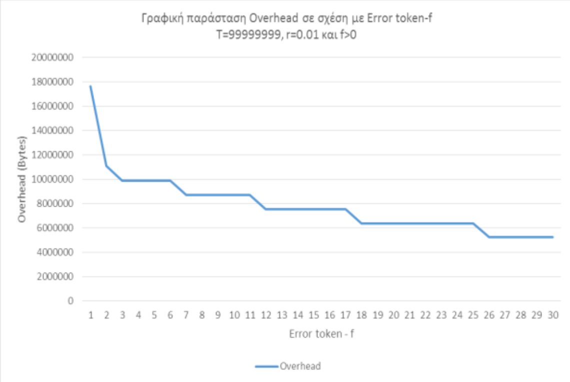


Σχήμα 3.63

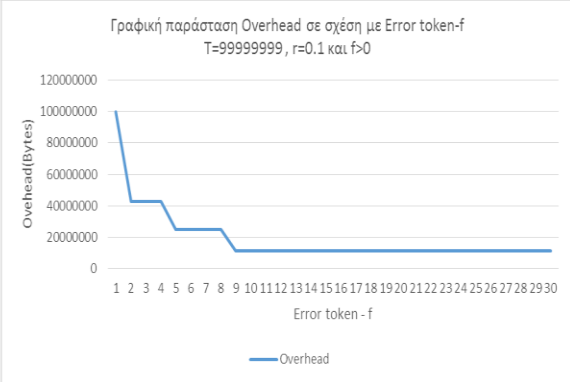


Σχήμα 3.64

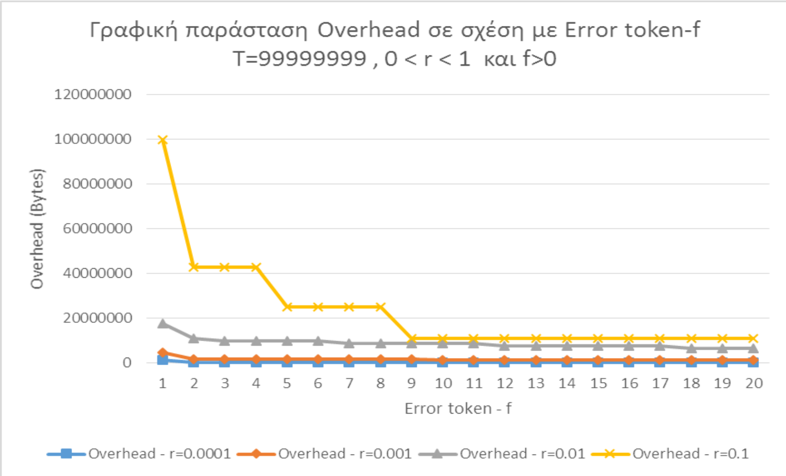
Στα Σχήματα 3.63, 3.64, 3.65, 3.66 και 3.67 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead παραμένει σταθερό για κάποιο εύρος τιμών και μετά μειώνεται ελαφρός. Αυτό είναι απολύτως λογικό γιατί κάθε φορά που αυξάνουμε το *f* κατά μία μονάδα και δεδομένου ότι το *f* έχει πάντα τη μέγιστη δυνατή τιμή του (αφού δεν καταστρέφεται κανένα πακέτο), τότε μπαίνει πιο γρήγορα στην τερματική περίπτωση όπου τα *error token* ξεπερνούν το φορτίο. Συνεπώς στέλνουν όλο το εναπομείναντα φορτίο => αποστολή λιγότερων πακέτων => λιγότερο Overhead. Επίσης η αρχική τιμή του Overhead είναι αρκετά μεγαλύτερη για *f* = 1. Αυτό οφείλεται στο γεγονός ότι μόνο σε αυτή την περίπτωση τρέχει ο αλγόριθμος ADP1. Επίσης παρατηρούμε ότι η συμπεριφορά που περιγράψαμε πιο πάνω συμβαίνει ανεξαρτήτως του χρόνου επαναφόρτισης. Όμως όσο μικρότερος είναι ο χρόνος επαναφόρτισης, παρατηρούμε πιο γρήγορη σταθεροποίηση του overhead, όμως με μεγαλύτερες τιμές.



Σχήμα 3.65



Σχήμα 3.66

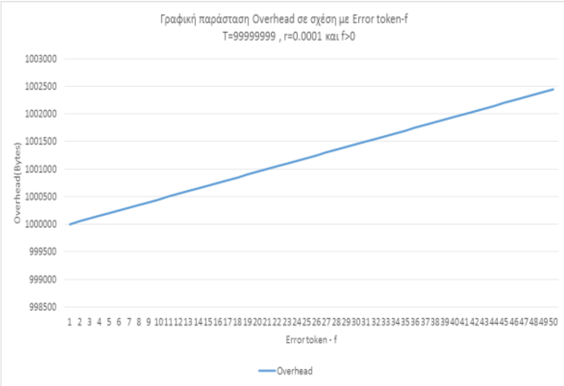


Σχήμα 3.67

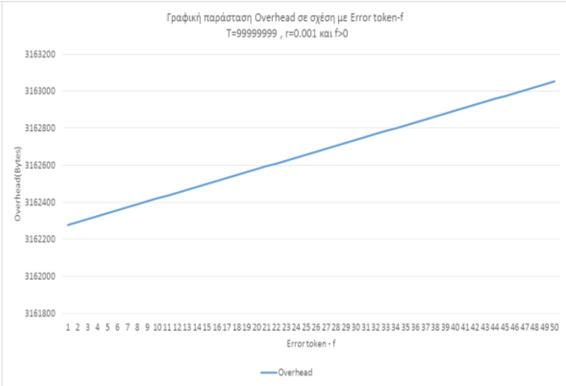
Αποτελέσματα για Uniform για T = 99999999

0.1 ≤ r ≤ 0.0001

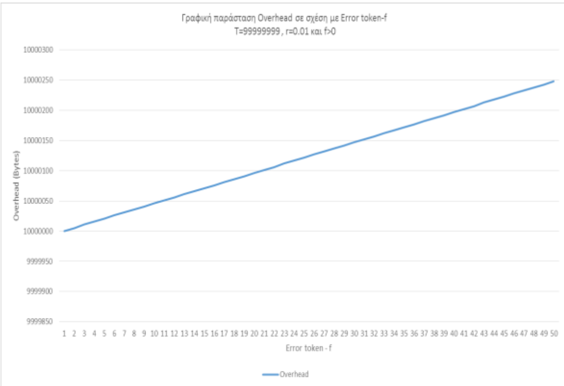
Στα Σχήματα 3.68, 3.69, 3.70, 3.71 και 3.72 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση. Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead και αυτό με γραμμικό ρυθμό. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος κατά τον υπολογισμό του μήκους του πακέτου, ο αλγόριθμος λαμβάνει υπόψη όλες τις παραμέτρους. Επίσης αυτός είναι και ο λόγος που το overhead αυξάνετε.



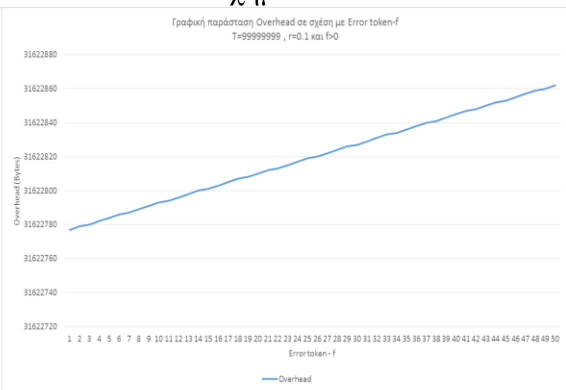
Σχήμα 3.68



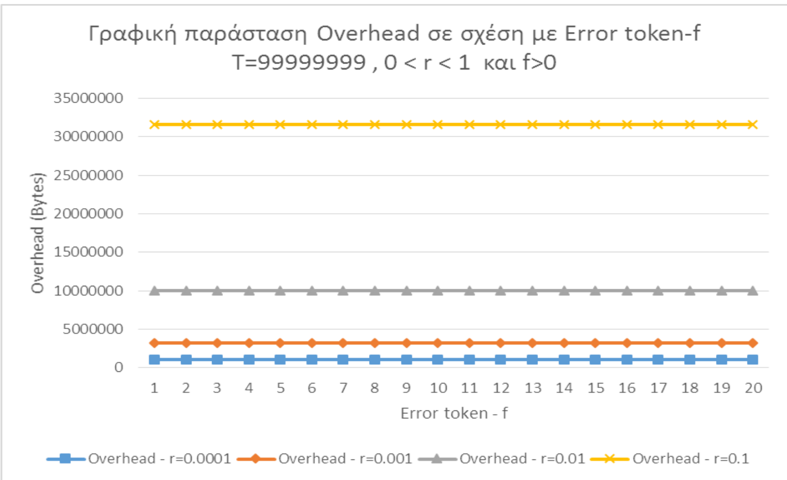
Σχήμα 3.69



Σχήμα 3.70

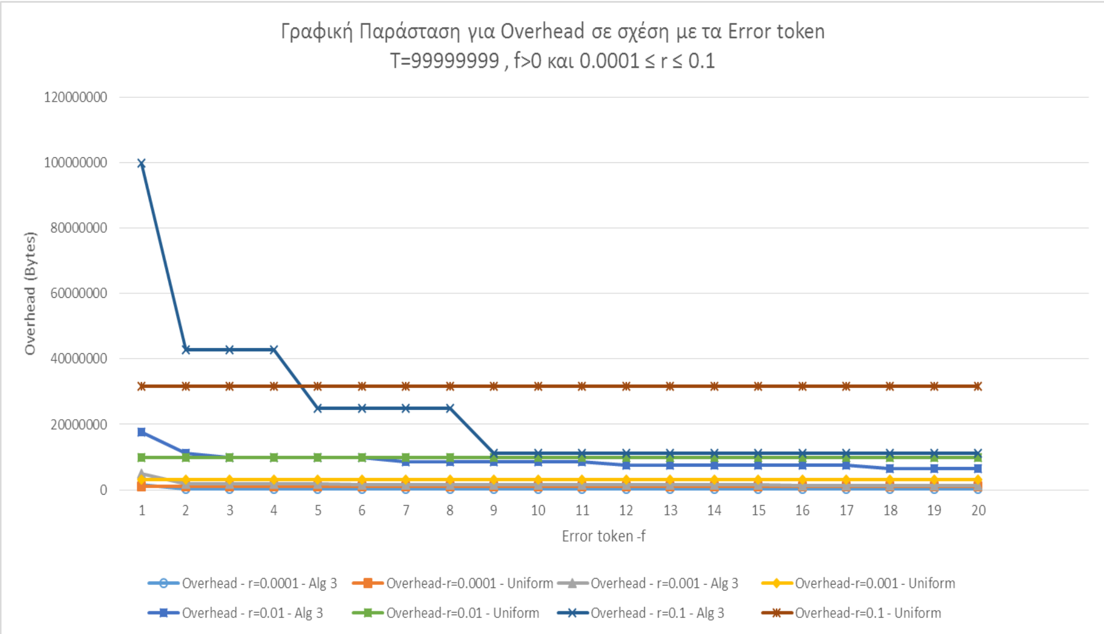


Σχήμα 3.71



Σχήμα 3.72

Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 VS Uniform – Γραφικές για T = 99999999



Σχήμα 3.73

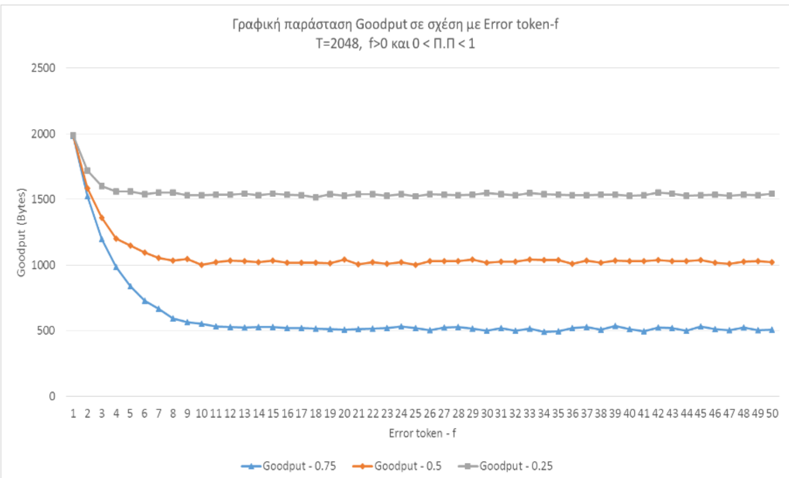
Με τη βοήθεια του σχήματος 3.73 μπορούμε να δούμε ότι για αυτό το σενάριο ο αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 υπερτερεί, όσο αφορά το Overhead. Επίσης μπορούμε ξεκάθαρα να δούμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή με T=2048, απλά με μεγαλύτερες τιμές για το Overhead.

6.4.2 Αποτελέσματα μη στοχευμένων σεναρίων

Αποτελέσματα για σενάριο 3

Αποτελέσματα για αλγόριθμο ADP_(*T*,*f*) με *f* >1 για T = 2048

Ωφέλιμο φορτίο – Goodput

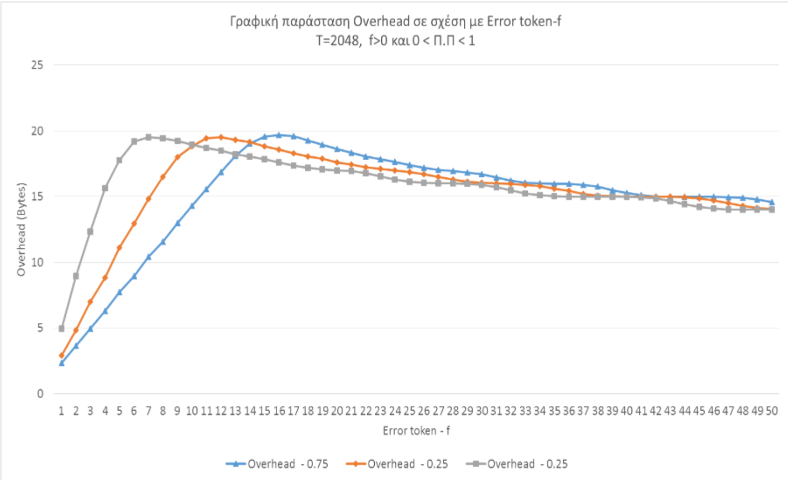


Σχήμα 3.74

Στο Σχήματος 3.74 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές

Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput παραμένει σχεδόν σταθερό. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f* (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

Overhead



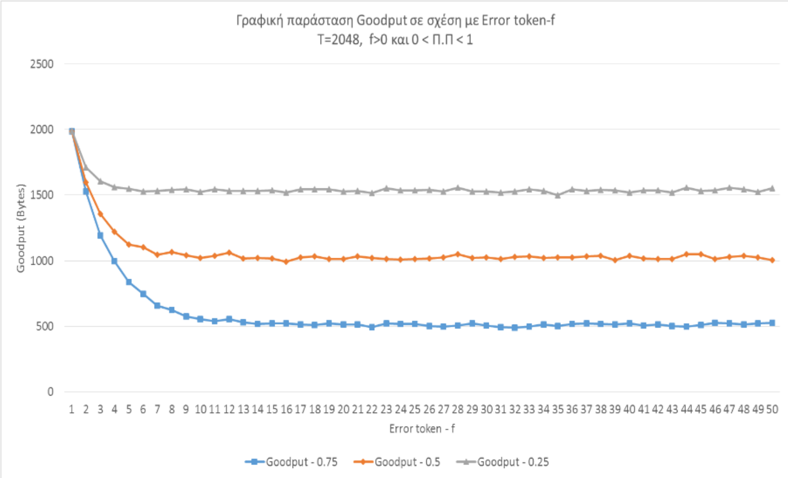
Σχήμα 3.75

Στο Σχήματος 3.75 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται με γρήγορους ρυθμούς. Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Στη συνέχεια παρατηρούμε ότι το overhead μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

Αποτελέσματα για αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 για T = 2048

r = 0.0001

<u>Ωφέλιμο φορτίο – Goodput</u>

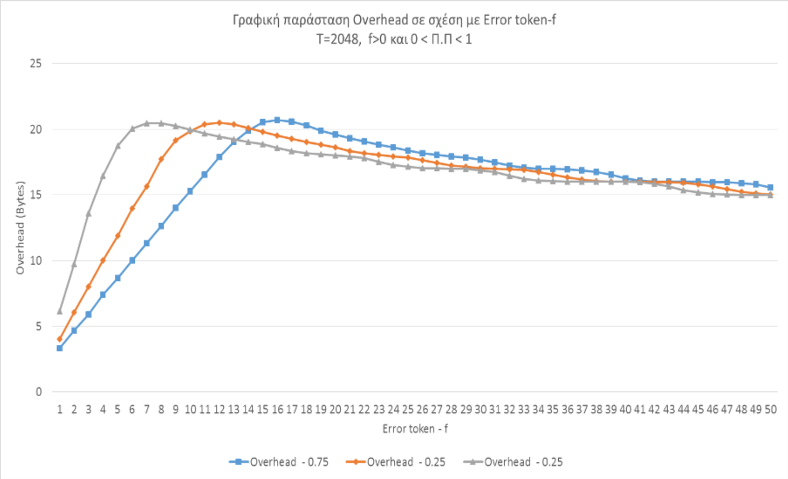


Σχήμα 3.76

Στο Σχήματος 3.76 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput παραμένει σχεδόν σταθερό. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f* (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

<u>Overhead</u>

Στο Σχήματος 3.77 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται με γρήγορους ρυθμούς. Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι

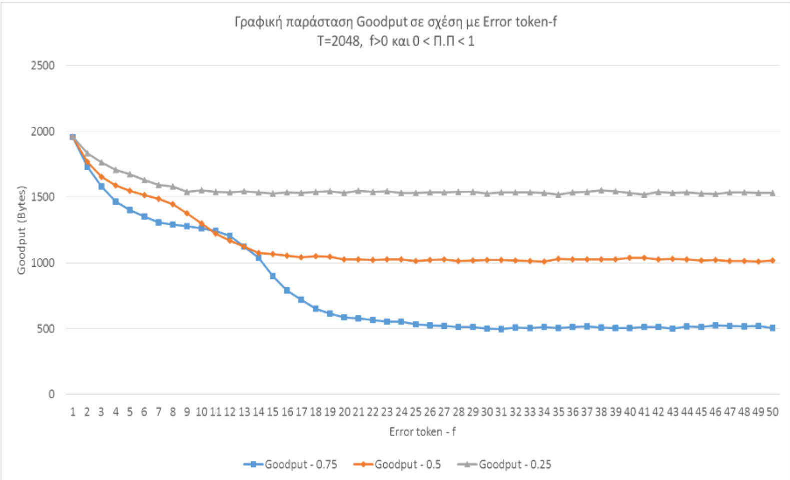


Σχήμα 3.77

χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Στη συνέχεια παρατηρούμε ότι το overhead μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

r = 0.001

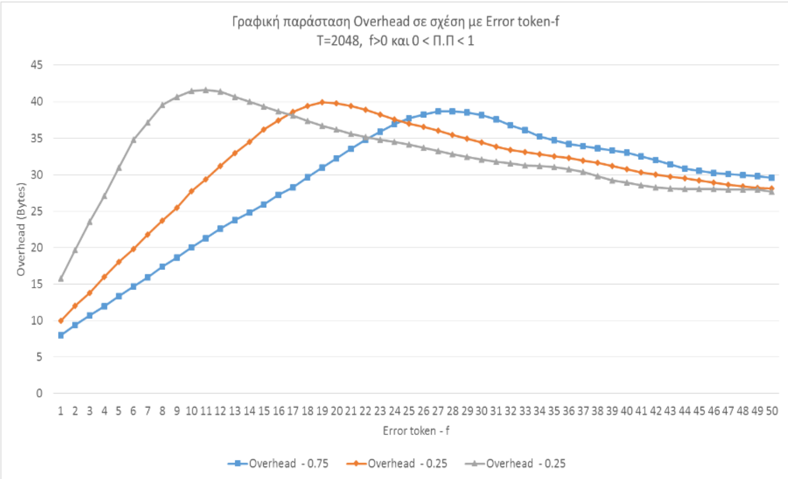
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.78

Στο Σχήματος 3.78 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Επίσης σε αυτή για αυτή τη χρονική διάρκεια παρατηρούμε και ένα είδος κυματισμού. Όπως έχουμε ήδη εξηγήσει σε προηγούμενα σενάρια αυτό οφείλεται στο διαχωρισμό των χρονικών περιόδων (time slot). Στη συνέχεια παρατηρούμε ότι το goodput παραμένει σχεδόν σταθερό. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f* (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

<u>Overhead</u>

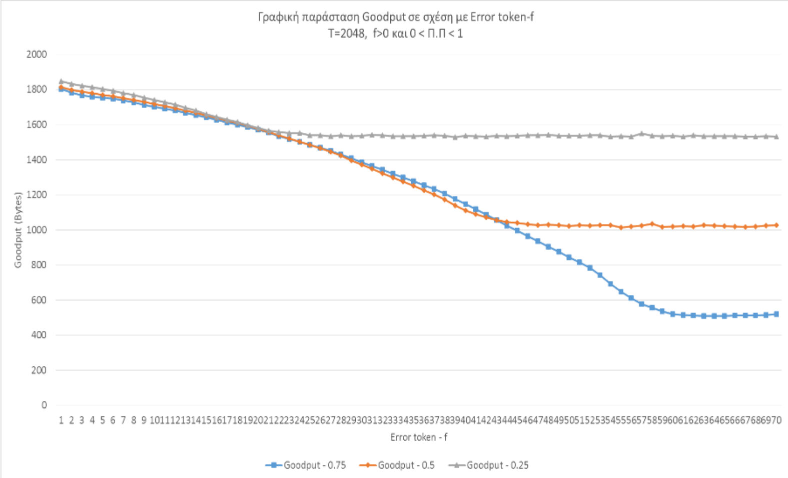


Σχήμα 3.79

Στο Σχήματος 3.79 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται με γρήγορους ρυθμούς. Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Στη συνέχεια παρατηρούμε ότι το overhead μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

r = 0.01

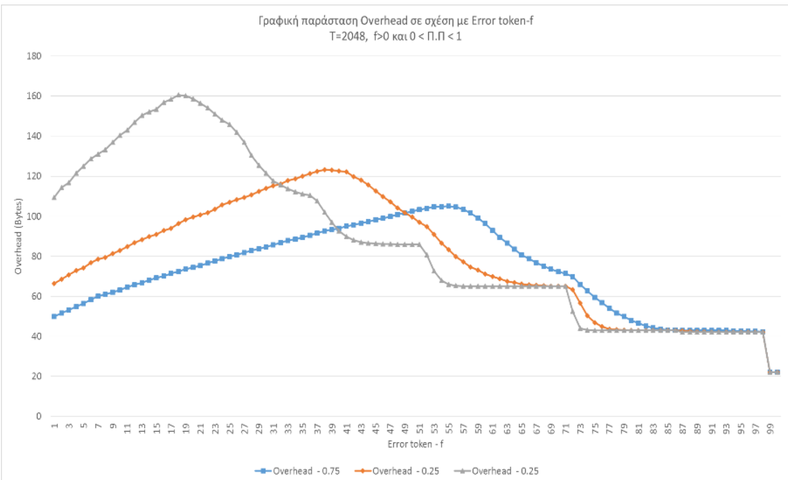
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.80

Στο Σχήματος 3.80 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται και ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput παραμένει σχεδόν σταθερό. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f* (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

Overhead



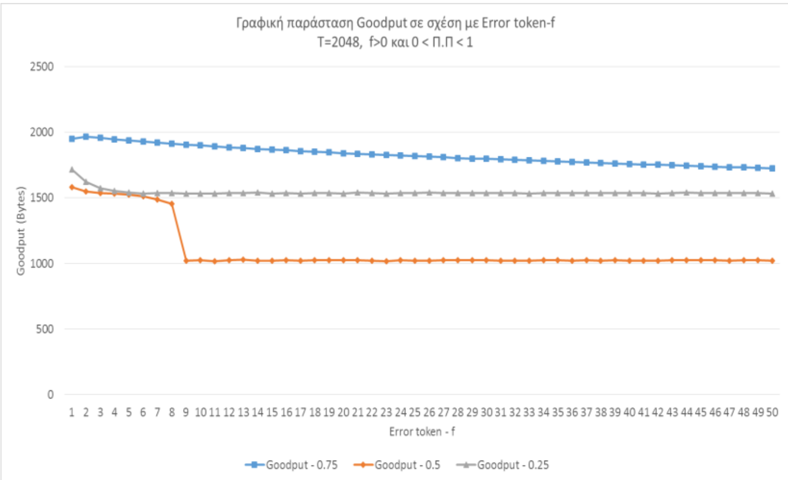
Σχήμα 3.81

Στο Σχήματος 3.81 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες

Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται με γρήγορους ρυθμούς. Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Στη συνέχεια παρατηρούμε ότι το overhead μειώνεται αρχικά με γρήγορους ρυθμούς ενώ στη συνέχεια με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

r = 0.1

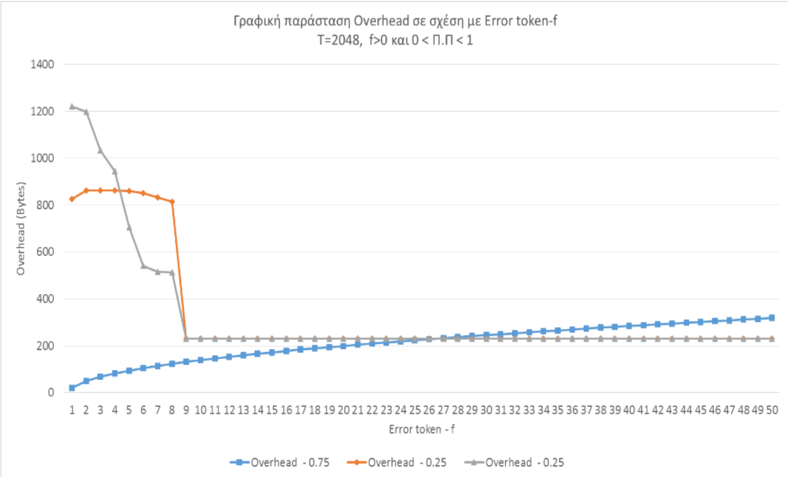
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.82

Στο Σχήματος 3.82 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι όταν η Π.Π είναι πιο μεγάλη το goodput μειώνεται με πολύ αργούς ρυθμούς. Ενώ στις άλλες δύο περιπτώσεις μειώνεται αρκετά και μετά παραμένει σχεδόν σταθερό. Αυτό οφείλεται και στο γεγονός ότι τα time slot που χωρίζουμε τον αλγόριθμο μας είναι πολύ μικρά. Από ένα σημείο και μετά το goodput σταθεροποιείται σχετικά, αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του f (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

Overhead



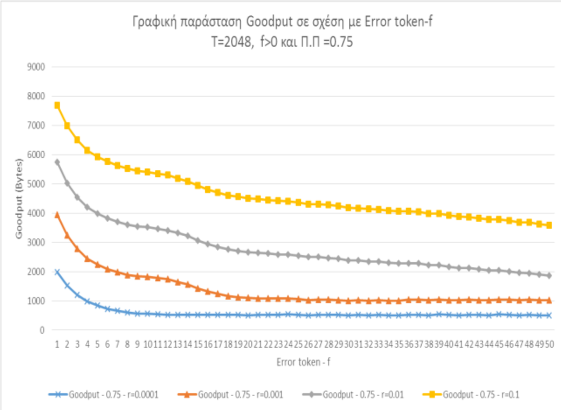
Σχήμα 3.83

Στο Σχήματος 3.83 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Για μεγάλες Π.Π το overhead αυξάνεται αρχικά γρήγορα και στη συνέχεια πιο αργά. Όμως για μικρές Π.Π το overhead αρχικά μειώνεται γρήγορα (όταν χρησιμοποιείται ο ADP1), ενώ στην συνέχεια με σταθεροποιείται. Αυτό οφείλεται στο γεγονός ότι τα time slot είναι πολύ μικρά σε συνδυασμό με το ότι ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token, που είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης φτάνουμε γρήγορα στην τερματική περίπτωση όπου το f είναι μεγαλύτερο από το εναπομείναντα φορτίο (για κάθε time slot) και έτσι στέλνεται ένα μεγαλύτερο πακέτο. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του f (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

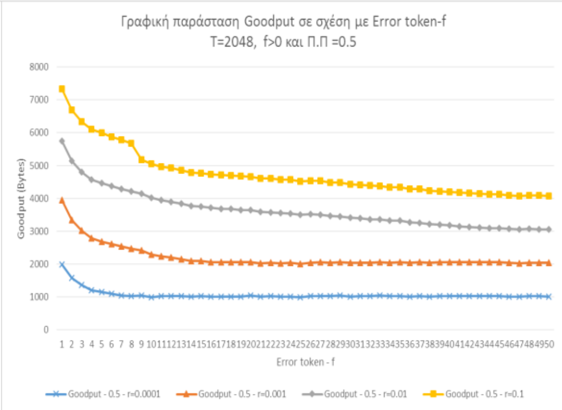
Σύγκριση όλων των r

Ωφέλιμο φορτίο – Goodput

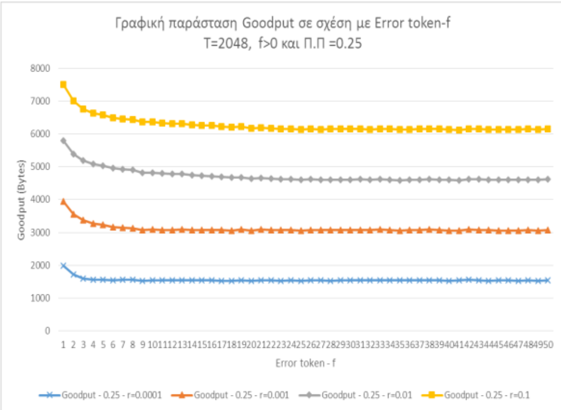
Με την βοήθεια των σχημάτων (3.84, 3.85, 3.86 και 3.87) που μπορούμε να δούμε στην επόμενη σελίδα ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο λιγότερο ωφέλιμο φορτίο έχουμε.



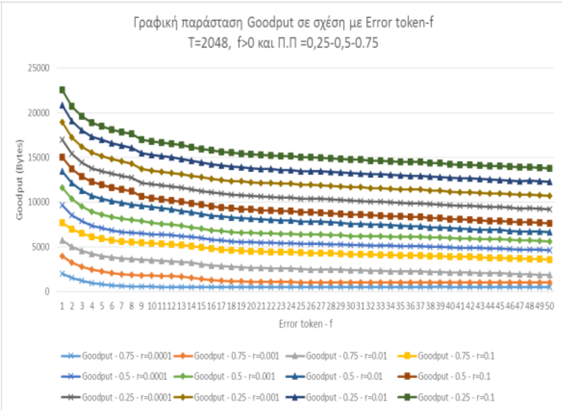
Σχήμα 3.84



Σχήμα 3.85



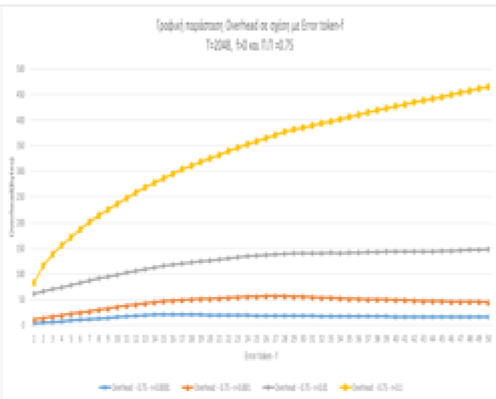
Σχήμα 3.86



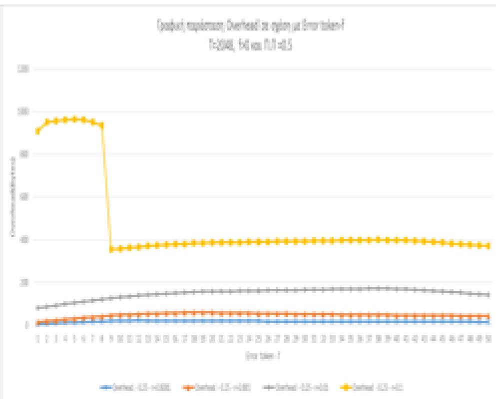
Σχήμα 3.87

Overhead

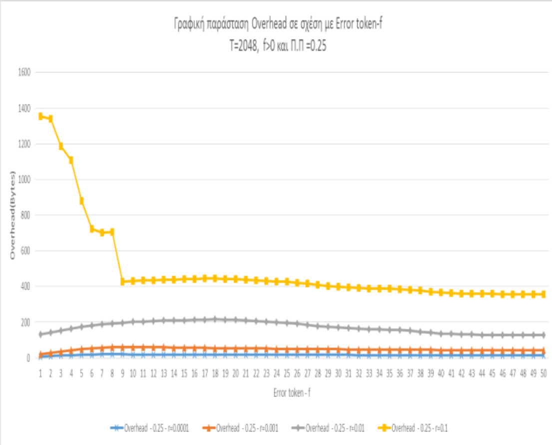
Με την βοήθεια των σχημάτων (3.88, 3.89, 3.90 και 3.91) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο μεγαλύτερο είναι το Overhead.



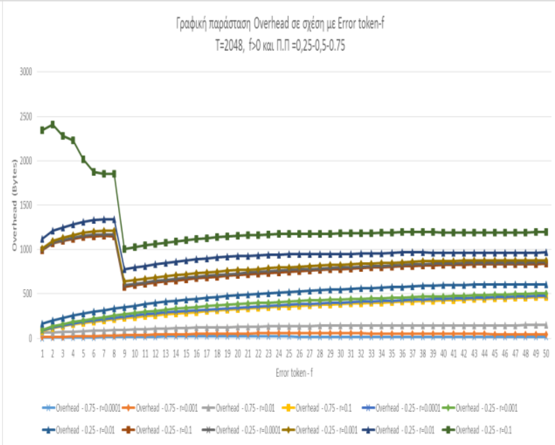
Σχήμα 3.88



Σχήμα 3.89



Σχήμα 3.90

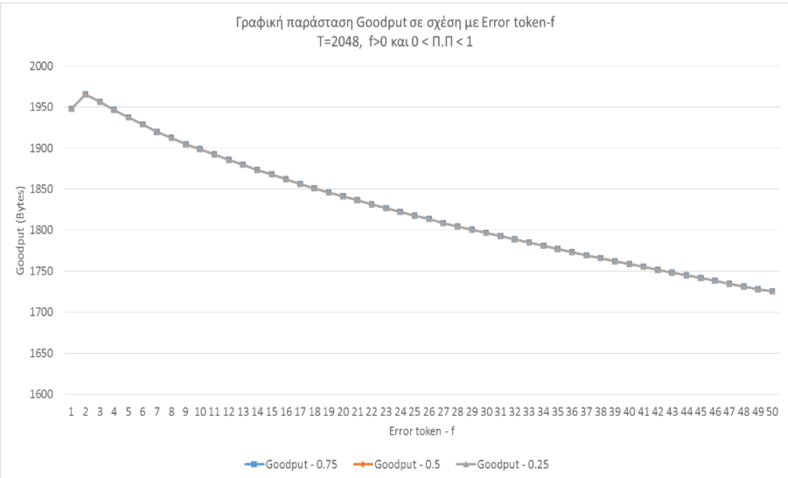


Σχήμα 3.91

Αποτελέσματα για Uniform για T = 2048

r = 0.0001

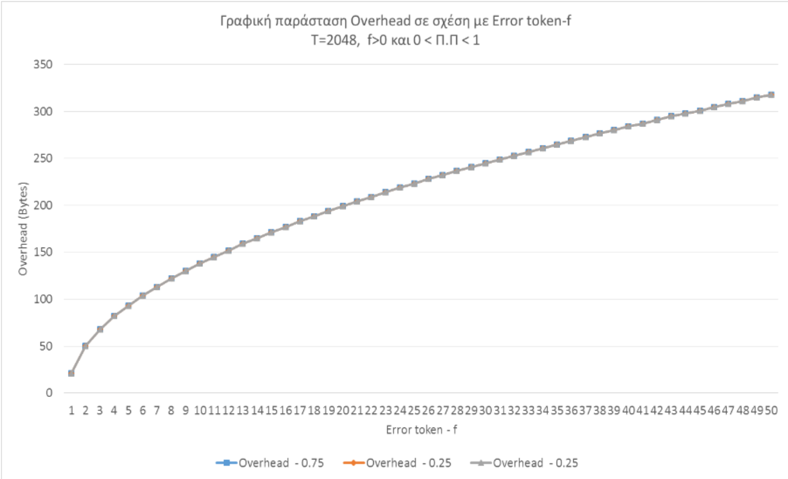
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.92

Στο Σχήματος 3.92 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token* το goodput μειώνεται με πιο αργούς ρυθμούς. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το goodput. Αν όμως ελέγξουμε τους πίνακες με τις τιμές για τα ανάλογα *f*, τότε θα παρατηρήσουμε ότι για σχετικά μεγάλες τιμές του *f* έχουμε μια μικρή διαφορά για κάθε Π.Π, η οποία δεν ξεπερνά το 0,5.

Overhead

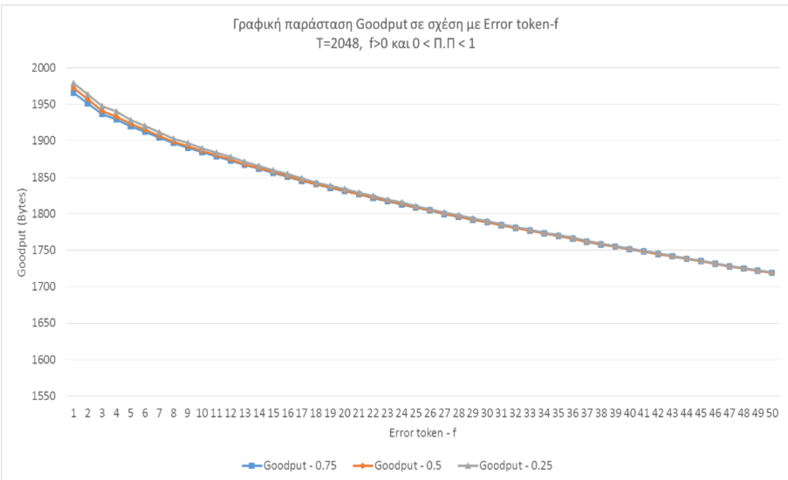


Σχήμα 3.93

Στο Σχήματος 3.93 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό αρχικά γρήγορα ενώ στην συνέχεια τείνει να γίνει γραμμική η αύξηση. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

r = 0.001

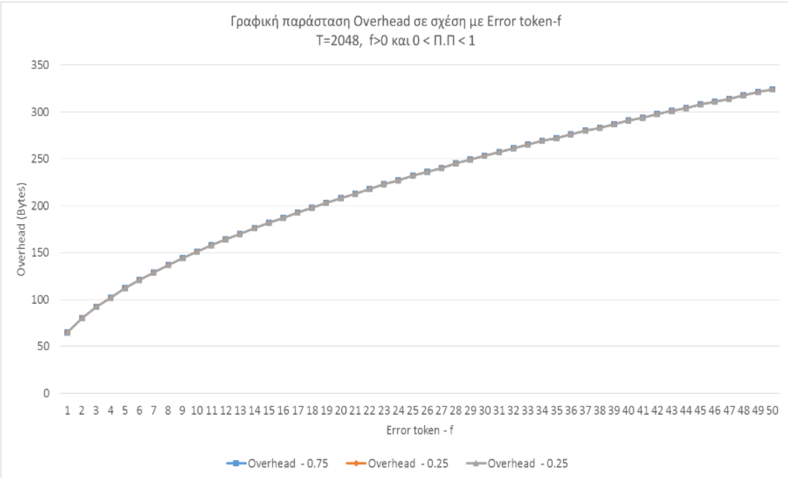
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.94

Στο Σχήματος 3.94 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Επίσης εύκολα παρατηρούμε ότι για αυτό το ρυθμό επαναφόρτισης είναι λίγο εμφανείς στην αρχή η διαφορά στο goodput. Όμως στη συνέχεια η διαφορά αυτή γίνεται πολύ μικρή, για να την αντιληφθεί κάποιος πρέπει να ελέγξει τους πίνακες. Αυτό είναι λογικό αφού όταν μεγαλώσουμε το *f* στέλνονται πακέτα μικρότερου μεγέθους. Επίσης λόγω του ότι η Π.Π δεν είναι 100% => δεν χρησιμοποιούνται όλα τα *error token*όταν αυξήσουμε το *f*.

<u>**Overhead**</u>



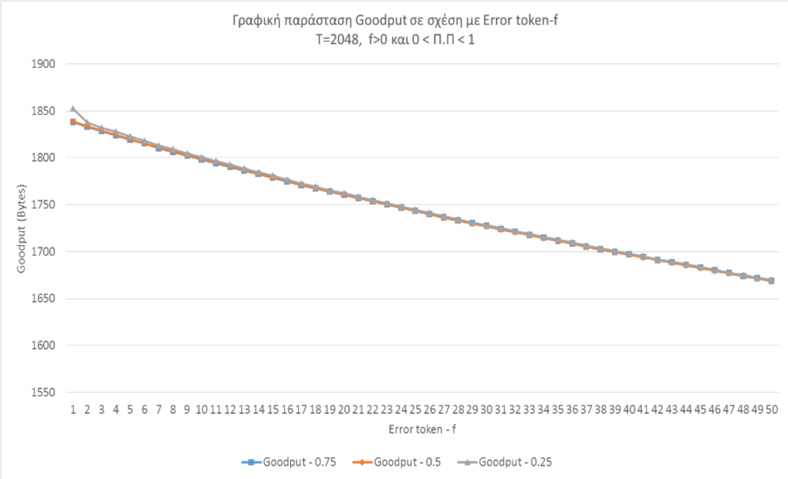
Σχήμα 3.95

Στο Σχήματος 3.95 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό αρχικά γρήγορα ενώ στην συνέχεια τείνει να γίνει γραμμική η αύξηση. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

r = 0.01

<u>**Ωφέλιμο φορτίο – Goodput**</u>

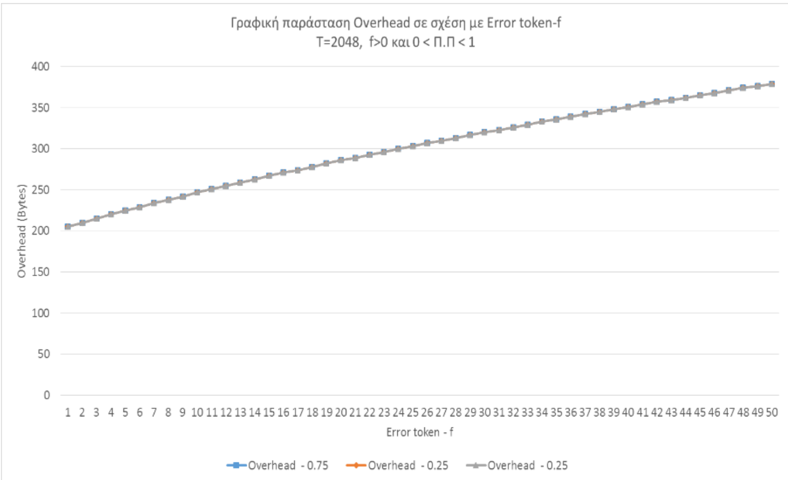
Στο Σχήματος 3.96 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με αργούς ρυθμούς (γραμμική μείωση). Επίσης εύκολα παρατηρούμε ότι για αυτό το ρυθμό επαναφόρτισης είναι λίγο εμφανείς στην αρχή η διαφορά στο goodput. Όμως στη συνέχεια η διαφορά αυτή γίνεται πολύ μικρή, για να την αντιληφθεί κάποιος πρέπει να ελέγξει τους πίνακες.



Σχήμα 3.96

Αυτό είναι λογικό αφού όταν μεγαλώσουμε το *f* στέλνονται πακέτα μικρότερου μεγέθους. Επίσης λόγω του ότι η Π.Π δεν είναι 100% => δεν χρησιμοποιούνται όλα τα *error token* όταν αυξήσουμε το *f*.

Overhead

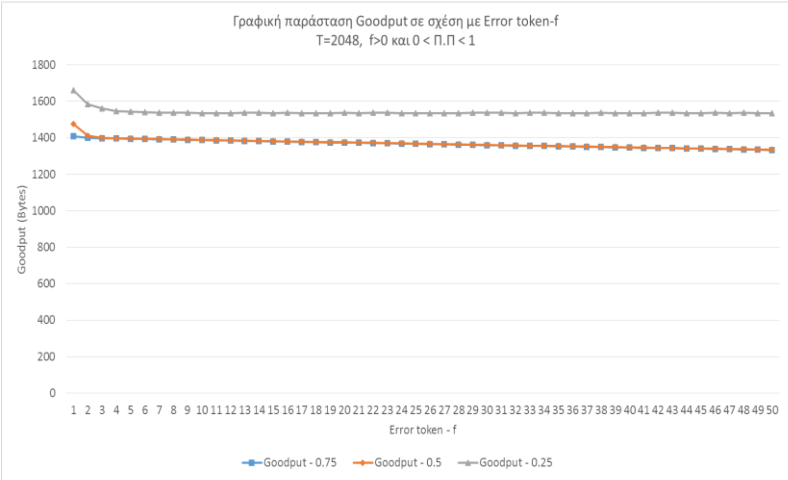


Σχήμα 3.97

Στο Σχήματος 3.97 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

r = 0.1

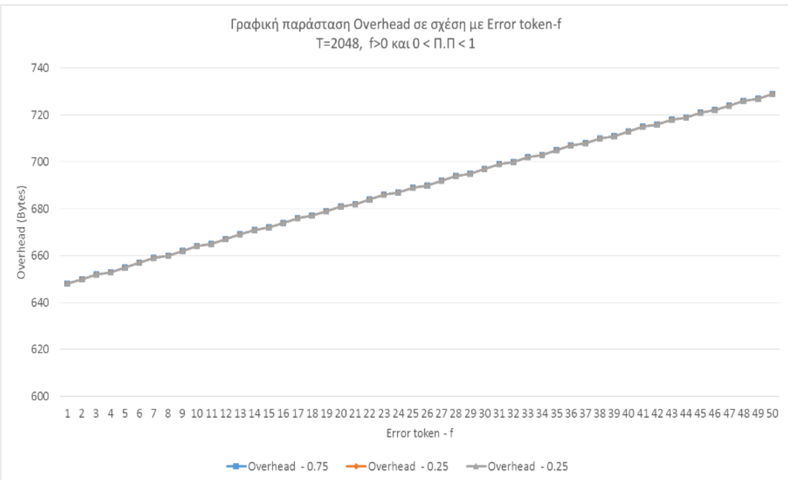
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.98

Στο Σχήματος 3.98 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με αργούς ρυθμούς (γραμμική μείωση). Επίσης εύκολα παρατηρούμε ότι για αυτό το ρυθμό επαναφόρτισης είναι εμφανές ότι για μικρή Π.Π έχουμε πολύ καλύτερα αποτελέσματα. Αυτό οφείλεται στο γεγονός ότι ο ρυθμός επαναφόρτισης είναι πάρα πολύ μικρός, έχει ως φυσικό επακόλουθο το μέγεθος των πακέτων είναι αρκετά μικρό. Άρα όταν η Π.Π είναι μικρή αναμένουμε ότι θα είναι εμφανείς η διαφορά στην καταστροφή των πακέτων μιας και αυτά είναι πολύ μικρά. Επίσης λόγω του ότι η Π.Π δεν είναι 100% => δεν χρησιμοποιούνται όλα τα *error token*όταν αυξήσουμε το *f*.

Overhead



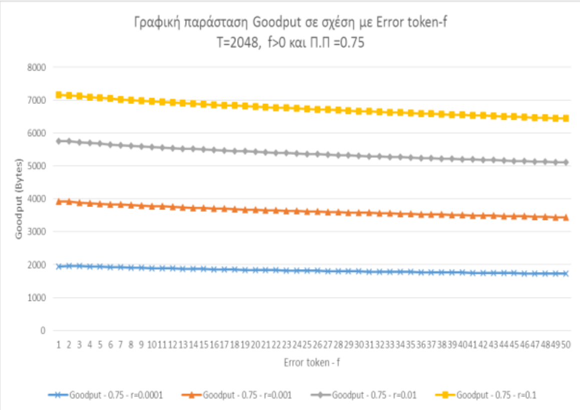
Σχήμα 3.99

Στο Σχήματος 3.99 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες

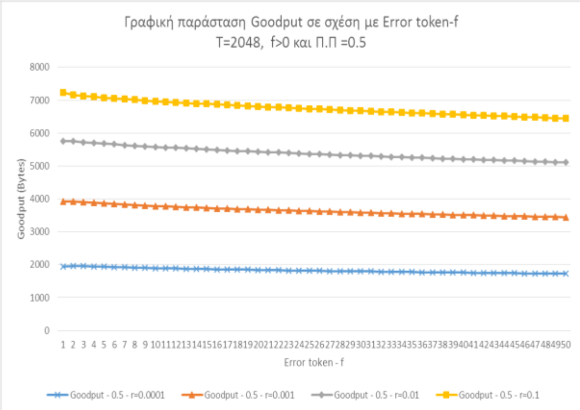
Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*form* στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

Σύγκριση όλων των r

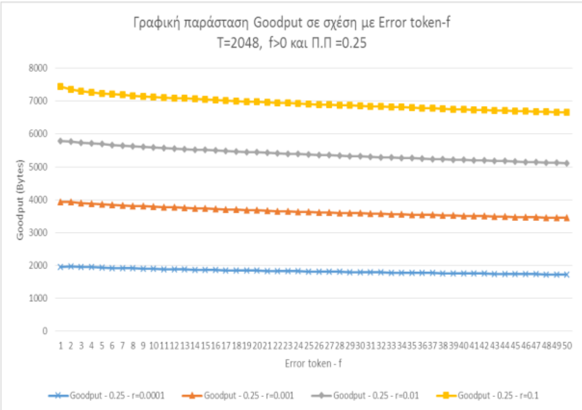
<u>Ωφέλιμο φορτίο – Goodput</u>



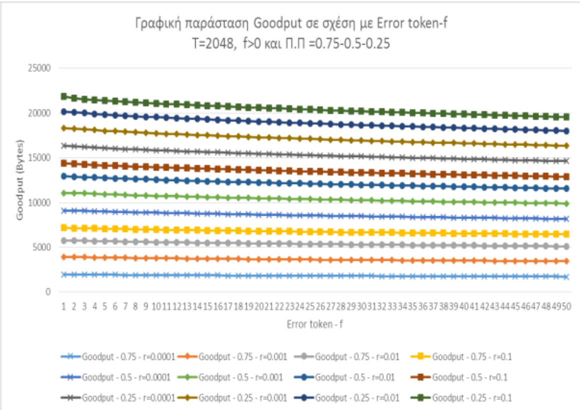
Σχήμα 3.100



Σχήμα 3.101



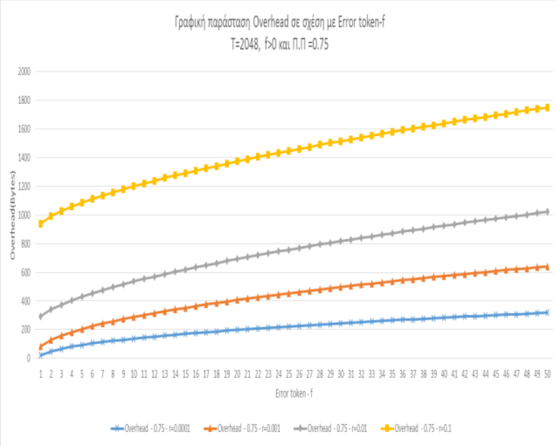
Σχήμα 3.102



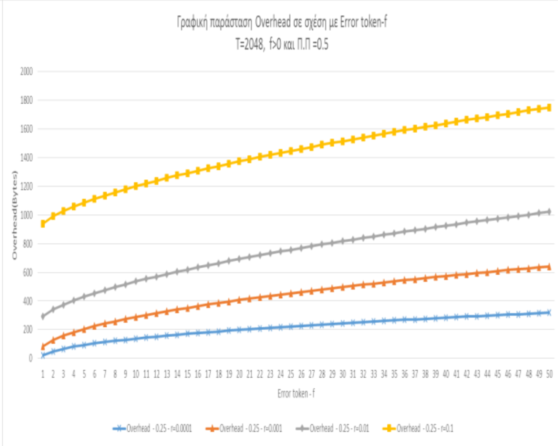
Σχήμα 3.103

Με την βοήθεια των πιο πάνω σχημάτων (3.100, 3.101, 3.102 και 3.103) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο λιγότερο ωφέλιμο φορτίο έχουμε.

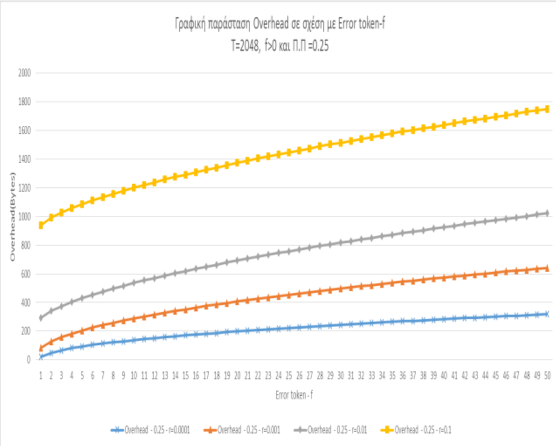
Overhead



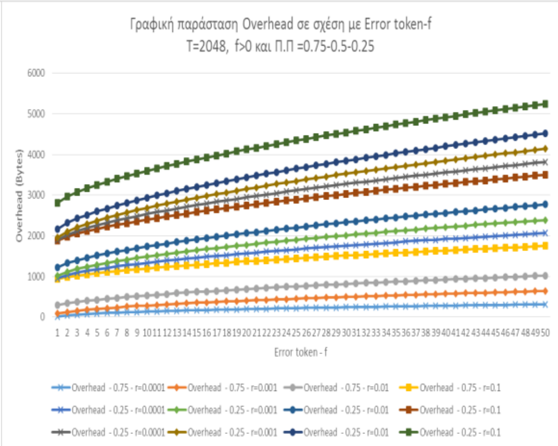
Σχήμα 3.104



Σχήμα 3.105



Σχήμα 3.106

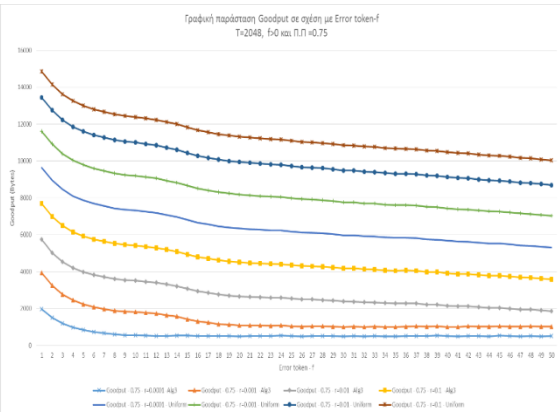


Σχήμα 3.107

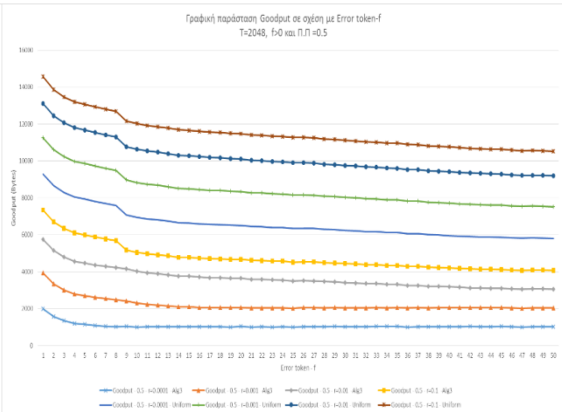
Με την βοήθεια των πιο πάνω σχημάτων (3.104, 3.105, 3.106 και 3.107) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο μεγαλύτερο είναι το Overhead.

Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 VS Uniform – Γραφικές για T = 2048

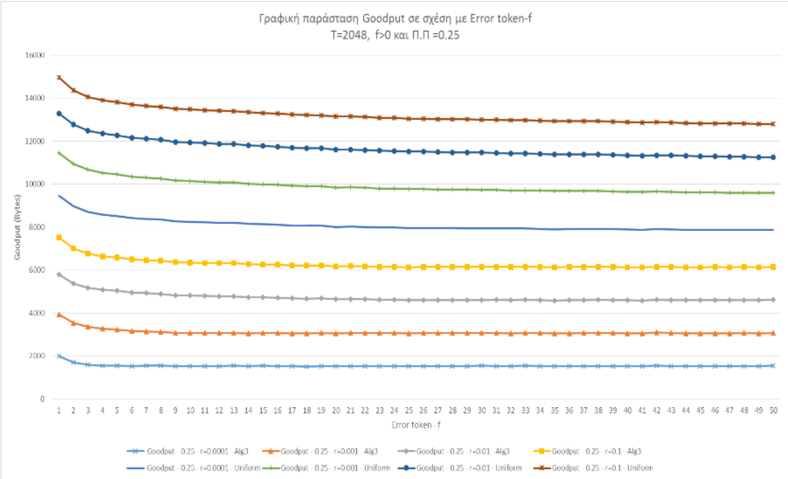
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.108

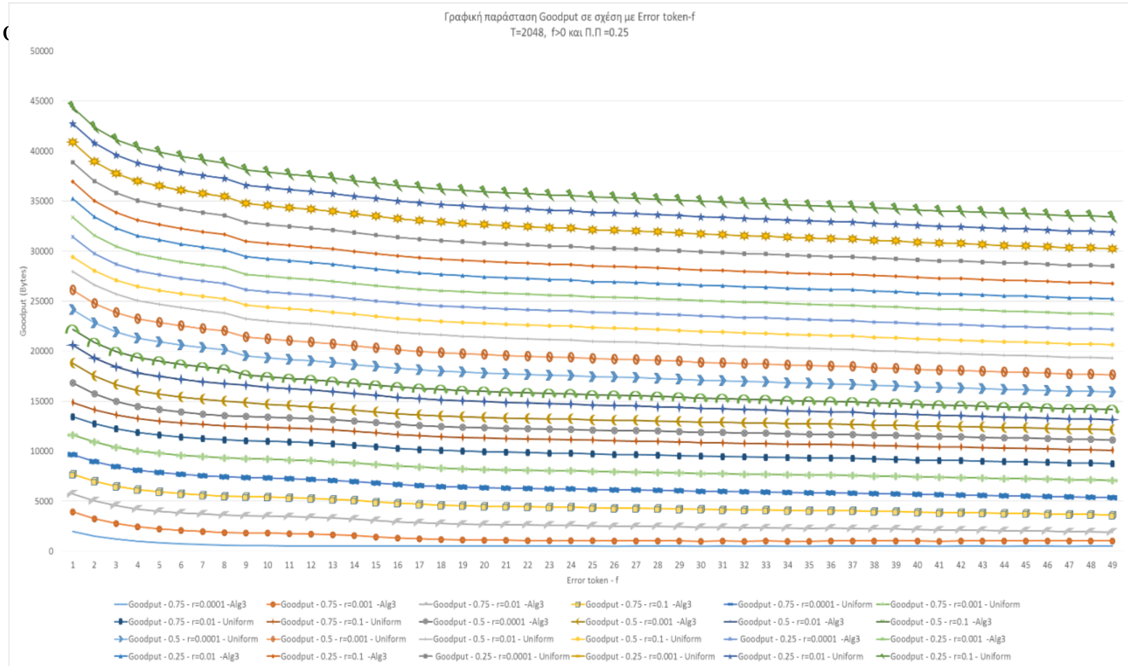


Σχήμα 3.109



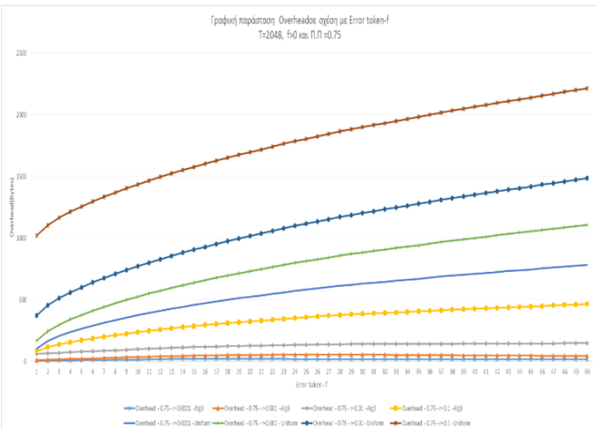
Σχήμα 3.110

Με την βοήθεια των πιο πάνω σχημάτων (3.108, 3.109, 3.110 και 3.111) εύκολα βλέπουμε όσο μεγαλύτερη είναι η Π.Π σε συνδυασμό με το ρυθμό επαναφόρτισης τόσο λιγότερο ωφέλιμο

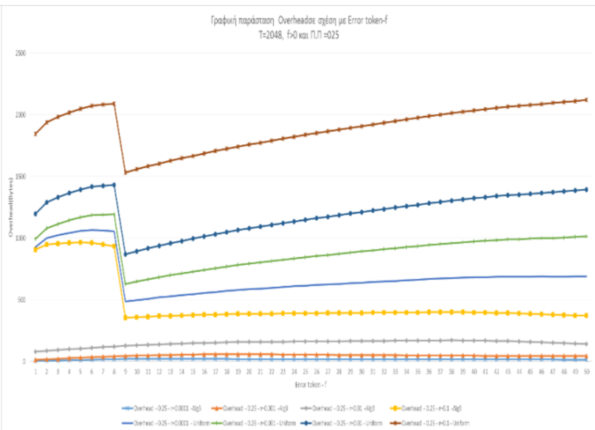


Σχήμα 3.111

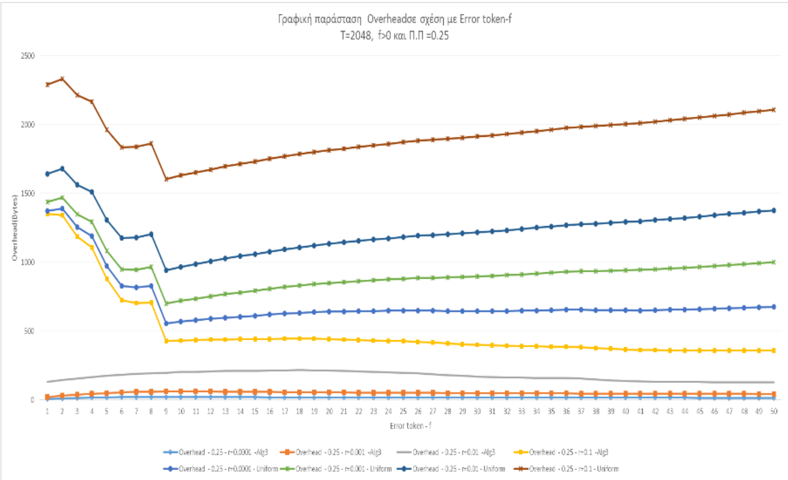
Overhead



Σχήμα 3.112

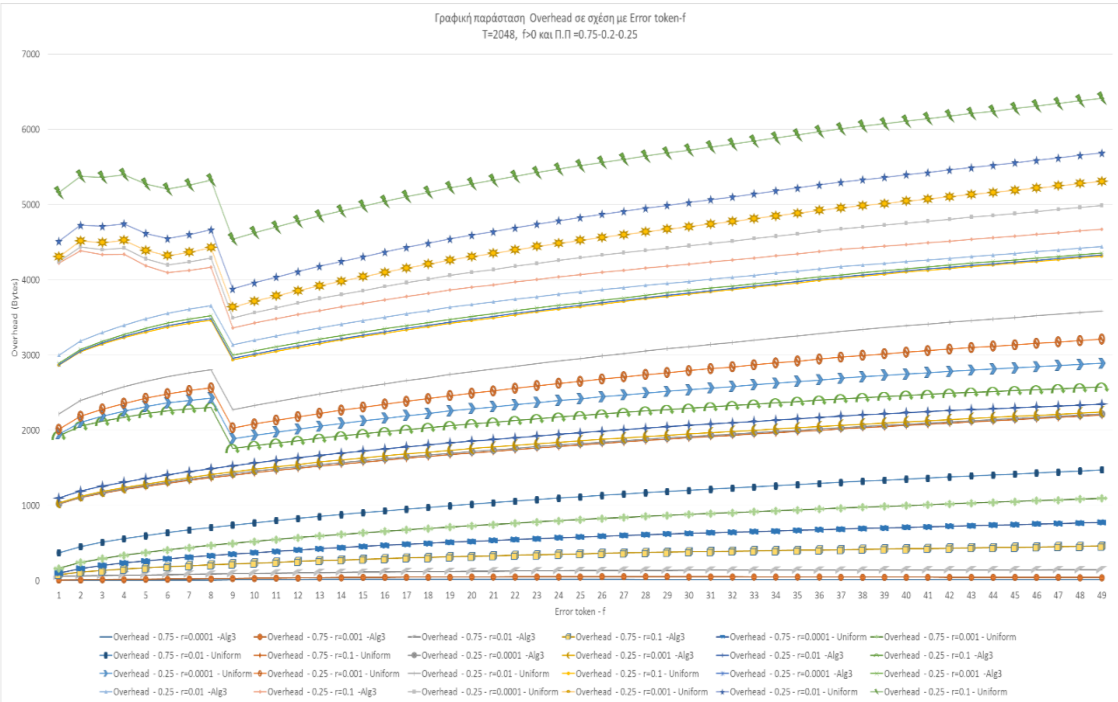


Σχήμα 3.113



Σχήμα 3.114

Με την βοήθεια των πιο πάνω σχημάτων (3.112, 3.113, 3.114 και 3.115) εύκολα βλέπουμε όσο μεγαλύτερη είναι η Π.Π σε συνδυασμό με το ρυθμό επαναφόρτισης τόσο μεγαλύτερο overhead έχουμε. Αυτό είναι λογικό γιατί δεν υπάρχουν περιπτώσεις που δεν χρησιμοποιούμε όλα τα *error token*γρήγορα => περισσότερα πακέτα μικρότερου μήκους.

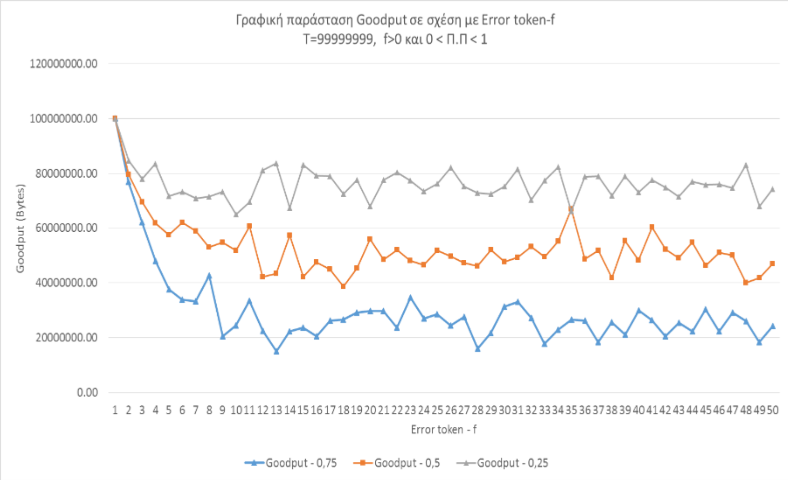


Σχήμα 3.115

Αποτελέσματα για αλγόριθμο ADP(*T*,*f*) με *f* >1 για T = 99999999

<u>Ωφέλιμο φορτίο – Goodput</u>

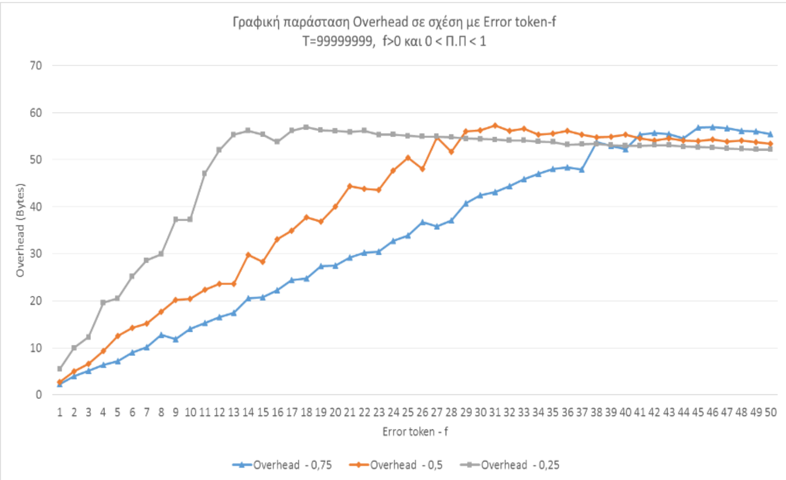
Στο Σχήματος 3.116 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς



Σχήμα 3.116

αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput αυξομειώνεται. Πράγμα το οποίο γινόταν και για T=2048, απλά εδώ μπορούμε να το διακρίνουμε αμέσως. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f* (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

Overhead

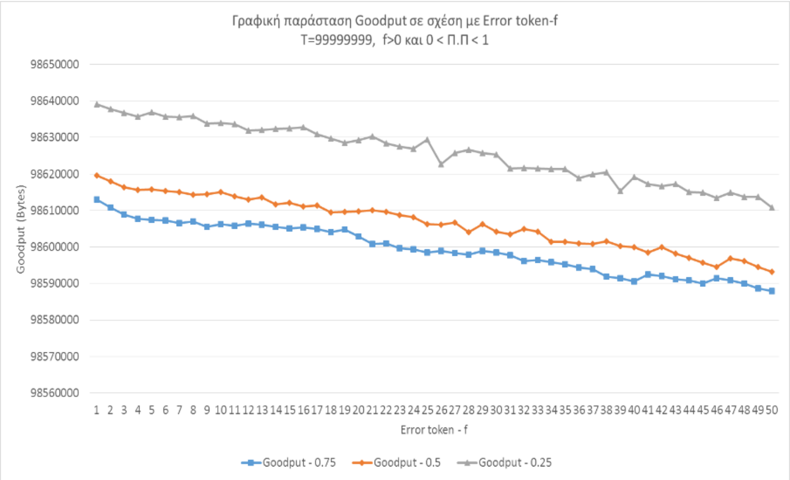


Σχήμα 3.117

Στο Σχήματος 3.117 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται με γρήγορους ρυθμούς. Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Στη συνέχεια παρατηρούμε ότι το overhead μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

Αποτελέσματα για αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 για T = 99999999 r = 0.0001

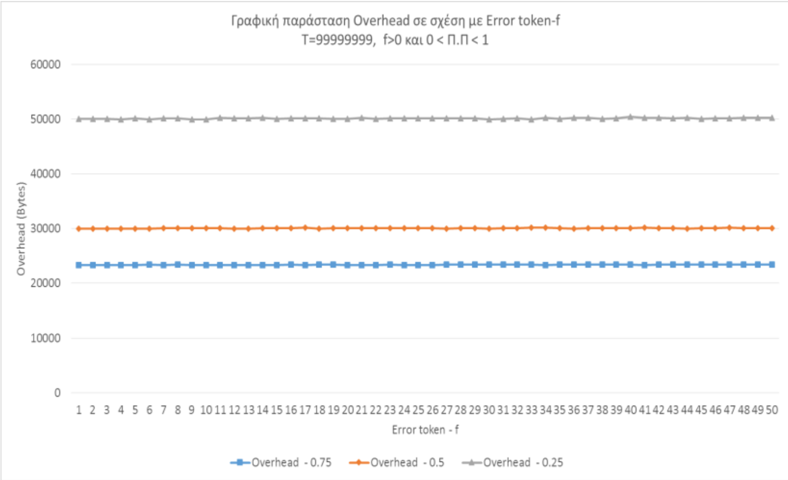
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.118

Στο Σχήματος 3.118 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f* (το έχουμε ελέγξει μέχρι $0 < f \leq 20000$)

Overhead



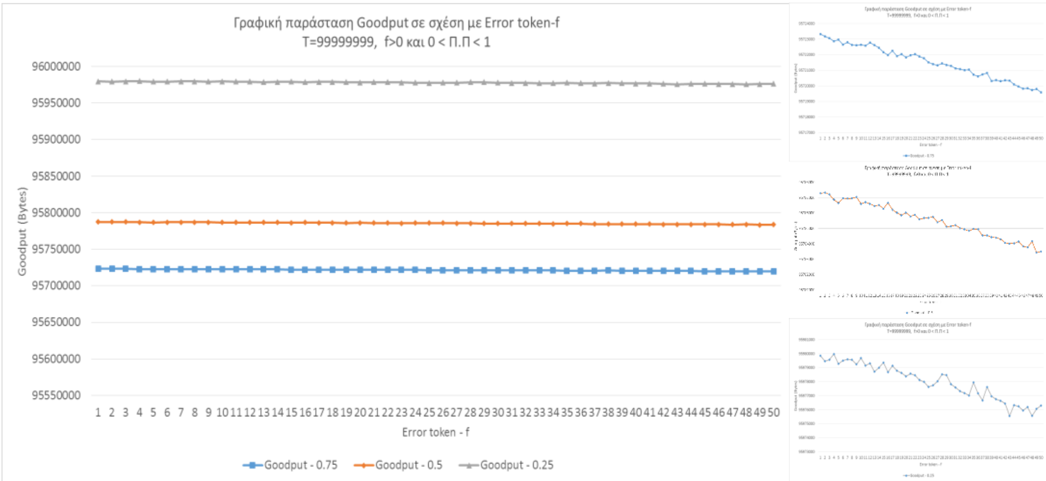
Σχήμα 3.119

Στο Σχήματος 3.119 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξομειώνεται γραμμικά για μικρά *f* (δεν φαίνεται από τη γραφική), ενώ στην συνέχεια αυξομειώνεται. Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Οι αυξομειώσεις οφείλονται στο γεγονός ότι τα πακέτα καταστρέφονται τυχαία σε διαφορετικά χρονικά διάστημα. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

r = 0.001

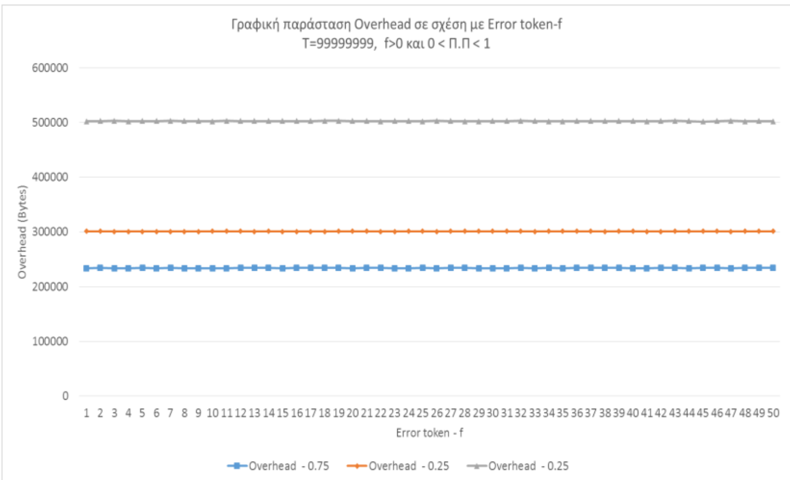
Ωφέλιμο φορτίο – Goodput

Στο Σχήματος 3.120 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά σχεδόν γραμμικά με μικρές αυξομειώσεις (λιγότερες από την προηγούμενη περίπτωση). Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput παραμένει μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα *error token* ή τα χρησιμοποιεί σε διαφορετικές χρονικές στιγμές. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%.



Σχήμα 3.120

<u>**Overhead**</u>

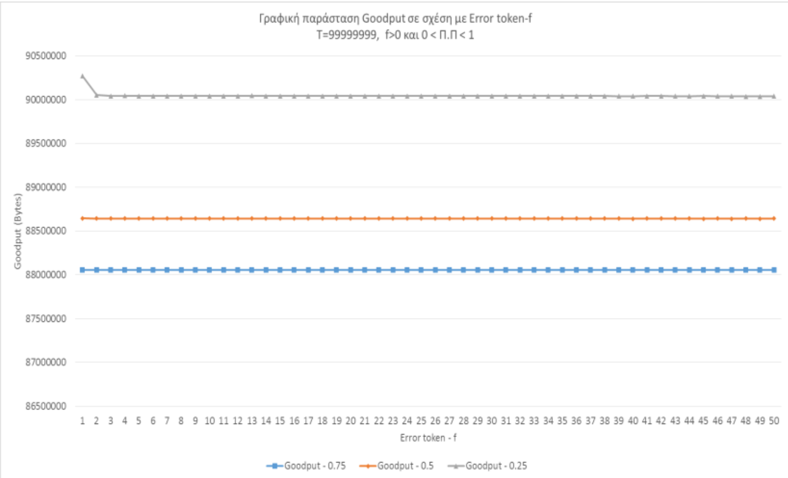


Σχήμα 3.121

Στο Σχήματος 6.121 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξομειώνεται γραμμικά για μικρά *f* (δεν φαίνεται από τη γραφική). Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Οι αυξομειώσεις οφείλονται στο γεγονός ότι τα πακέτα καταστρέφονται τυχαία σε διαφορετικά χρονικά διάστημα. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Στη συνέχεια όμως όταν αυξήσουμε αρκετά το *f*, αυτό τείνει προς τις ίδιες τιμές.

r = 0.01

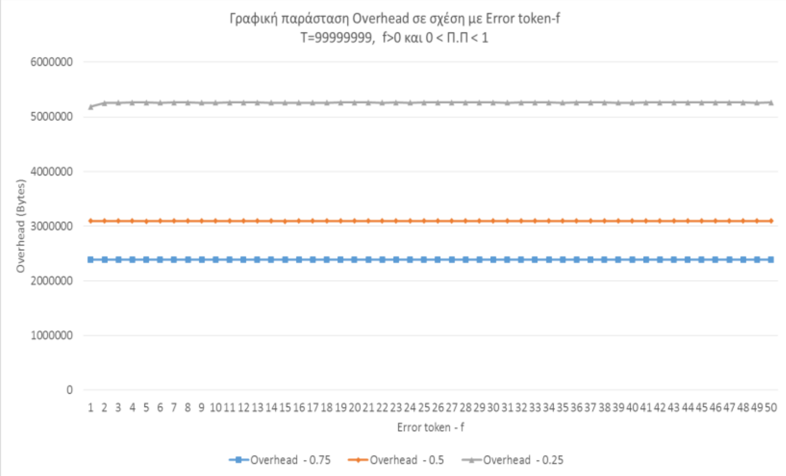
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.122

Στο Σχήματος 3.122 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή. Όμως όσο μεγαλύτερη είναι η Π.Π τότε το goodput μειώνεται περισσότερο. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά σχεδόν γραμμικά με μικρές αυξομειώσεις (λιγότερες από την προηγούμενη περίπτωση). Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput μειώνεται με πιο αργούς ρυθμούς. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα *error token* ή τα χρησιμοποιεί σε διαφορετικές χρονικές στιγμές. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%.

Overhead



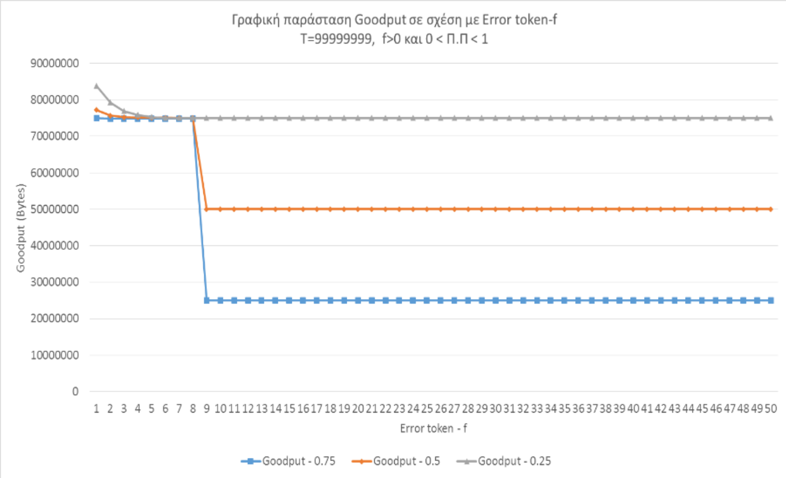
Σχήμα 3.123

Στο Σχήματος 3.123 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα διακρίνουμε ότι η γραφικές παραστάσεις έχουν την ίδια μορφή και ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξομειώνεται γραμμικά για μικρά *f* (δεν φαίνεται από τη γραφική). Αυτό είναι απολύτως λογικό, γιατί όπως έχουμε εξηγήσει και προηγουμένως αυτό οφείλεται στο γεγονός ότι χρησιμοποιείται και ο αλγόριθμος ADP1 πιο συχνά, ενώ στη συνέχεια χρησιμοποιείται λιγότερο μέχρι και καθόλου. Αυτό οφείλεται στο γεγονός ότι όταν αυξάνουμε το *f*, ο αλγόριθμος μπαίνει πιο γρήγορα στην περίπτωση όπου το *f* είναι μεγαλύτερο ή ίσο με το εναπομείναν φορτίο=> στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Οι αυξομειώσεις οφείλονται στο γεγονός ότι τα πακέτα καταστρέφονται τυχαία σε διαφορετικά χρονικά διάστημα. Τέλος παρατηρούμε ότι αρχικά το overhead είναι μεγαλύτερο όταν η Π.Π. είναι μικρότερη. Αυτό είναι και το αναμενόμενο αφού το *f* μειώνεται με πιο αργούς ρυθμούς. Επίσης για μικρότερες τιμές παρεμβολής αυτή η αυξομείωση είναι μικρότερης κλίμακας.

r = 0.1

Ωφέλιμο φορτίο – Goodput

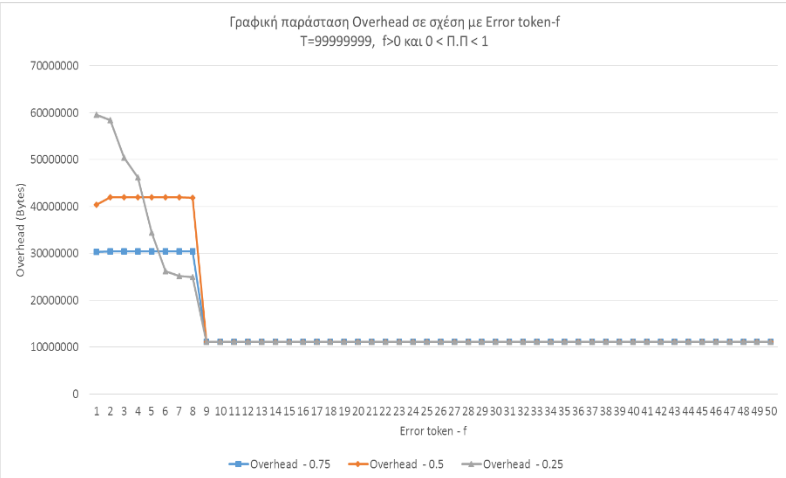
Στο Σχήματος 3.124 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Επιπλέον παρατηρούμε ότι για $f \leq 7$ το goodput μειώνεται γρηγορότερα. Αυτό οφείλεται στο γεγονός ότι μέχρι αυτό το σημείο χρησιμοποιείται κυρίως ο ADP1, ενώ στη συνέχεια



Σχήμα 3.124

μπαίνει στο παιχνίδι ο ADP2. Επίσης εύκολα διακρίνουμε ότι όταν η Π.Π είναι πιο μικρή το goodput μειώνεται με πολύ αργούς ρυθμούς. Ενώ στις άλλες δύο περιπτώσεις μειώνεται αρκετά και μετά παραμένει σχεδόν σταθερό, με μικρές αυξομειώσεις. Εδώ όμως οι αυξομειώσεις αυτές είναι πολύ μικρότερες από τις προηγούμενες περιπτώσεις. Αυτό οφείλεται και στο γεγονός ότι τα time slot που χωρίζουμε τον αλγόριθμο μας είναι πολύ μικρά. Από ένα σημείο και μετά το goodput σταθεροποιείται σχετικά, αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%.

Overhead

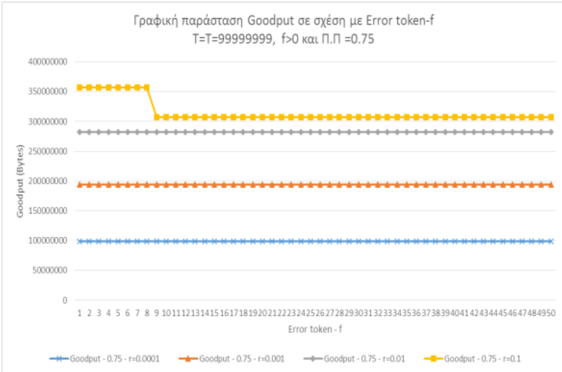


Σχήμα 6.125

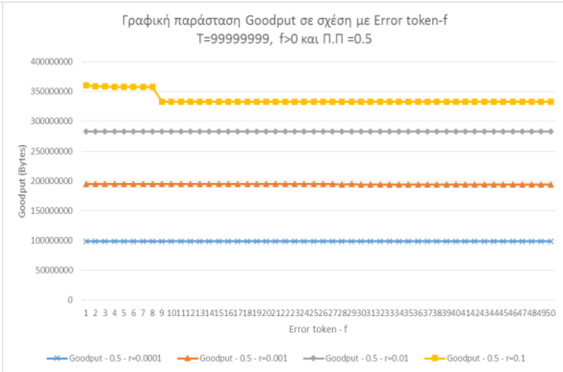
Στο Σχήματος 6.125 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Το overhead αρχικά μειώνεται γρήγορα (όταν χρησιμοποιείται ο ADP1), ενώ στην συνέχεια σταθεροποιείται. Αυτό οφείλεται στο γεγονός ότι τα time slot είναι πολύ μικρά σε συνδυασμό με το ότι ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token, που είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Επίσης φτάνουμε γρήγορα στην τερματική περίπτωση όπου το *f* είναι μεγαλύτερο από το εναπομείναντα φορτίο (για κάθε time slot) και έτσι στέλνεται ένα μεγαλύτερο πακέτο. Επίσης αυτή η συμπεριφορά συνεχίζεται για πολύ μεγάλες τιμές του *f*.

Σύγκριση όλων των r

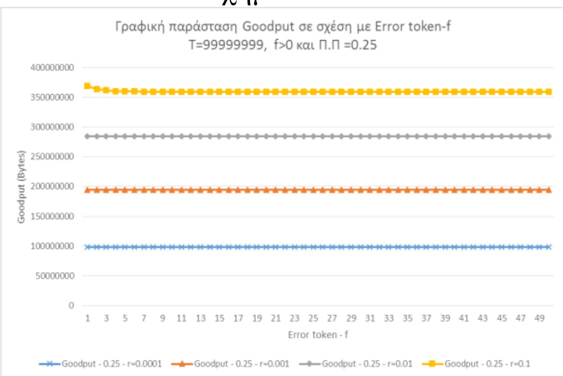
Ωφέλιμο φορτίο – Goodput



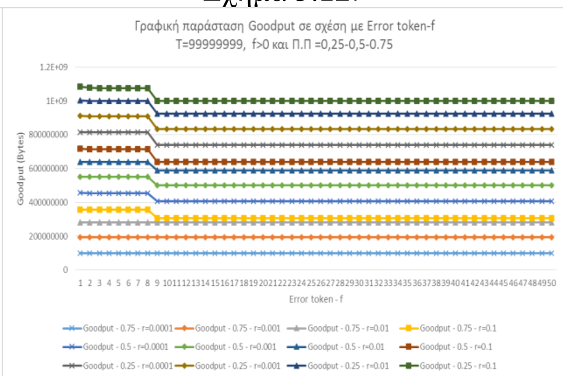
Σχήμα 3.126



Σχήμα 3.127



Σχήμα 3.128

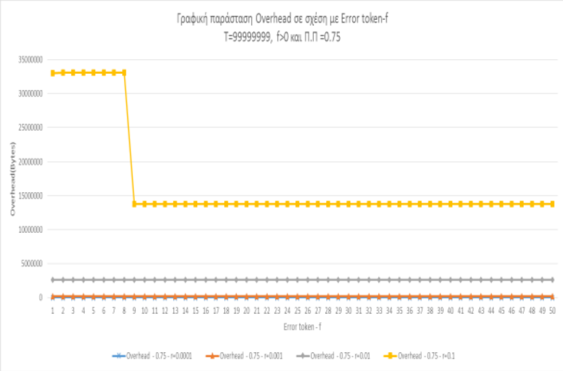


Σχήμα 3.129

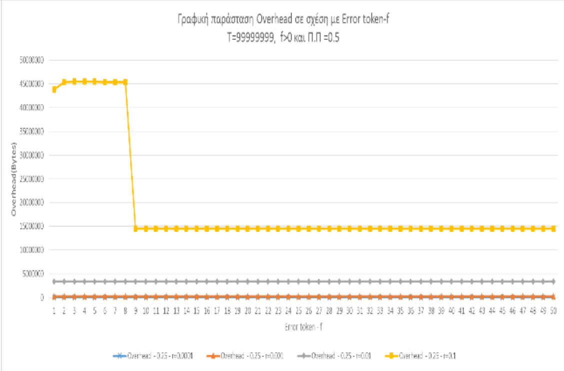
Με την βοήθεια των πιο πάνω σχημάτων (3.126, 3.127, 3.128 και 3.129) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο λιγότερο ωφέλιμο φορτίο έχουμε.

Overhead

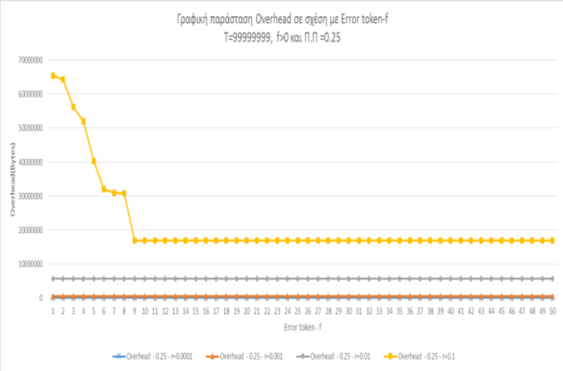
Με την βοήθεια των σχημάτων (3.130, 3.131, 3.132 και 3.133) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο μεγαλύτερο είναι το Overhead.



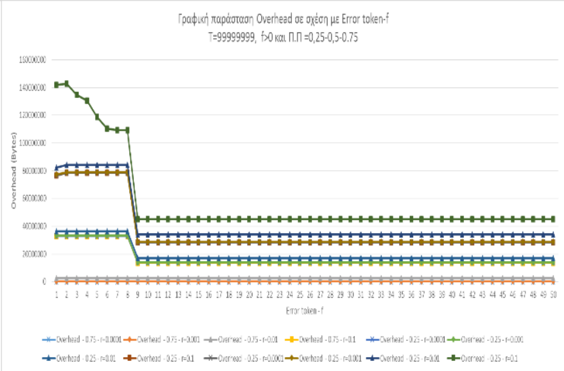
Σχήμα 3.130



Σχήμα 3.131



Σχήμα 3.132

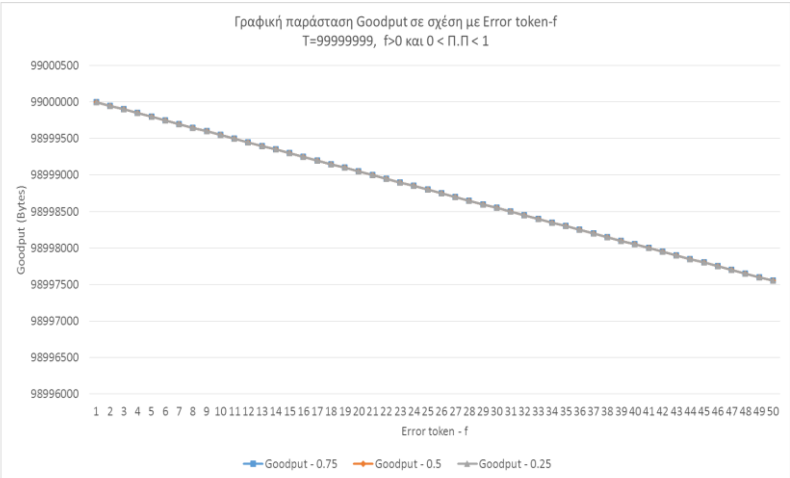


Σχήμα 3.133

Αποτελέσματα για Uniform για T = 99999999

r = 0.0001

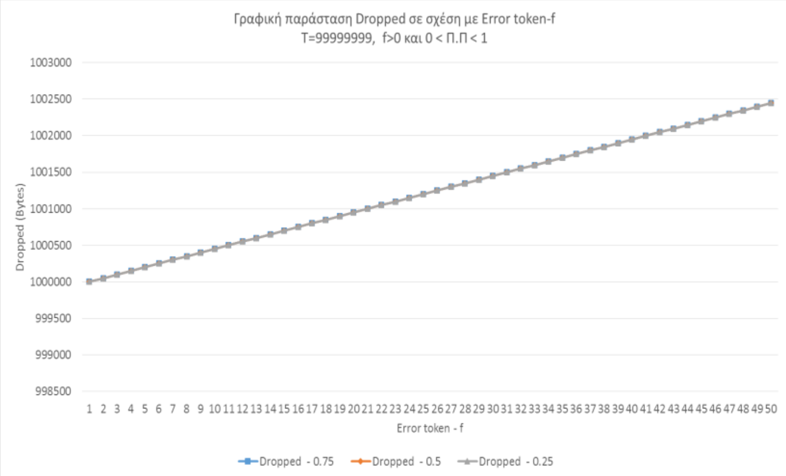
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.134

Στο Σχήματος 3.134 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το goodput. Αν όμως ελέγξουμε τους πίνακες με τις τιμές για τα ανάλογα *f*, τότε θα παρατηρήσουμε ότι για σχετικά μεγάλες τιμές του *f* έχουμε μια μικρή διαφορά για κάθε Π.Π, η οποία δεν ξεπερνά το 0,5.

<u>Overhead</u>



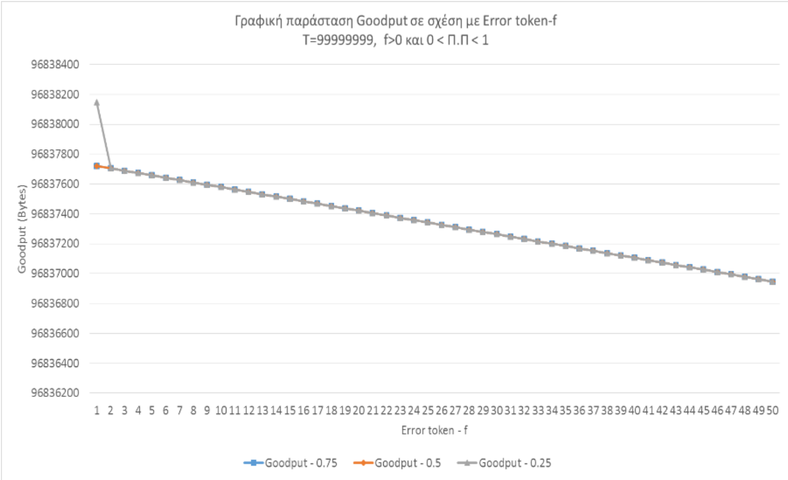
Σχήμα 3.135

Στο Σχήματος 3.135 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό αρχικά γρήγορα ενώ στην συνέχεια τείνει να γίνει γραμμική η αύξηση. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

r = 0.001

<u>Ωφέλιμο φορτίο – Goodput</u>

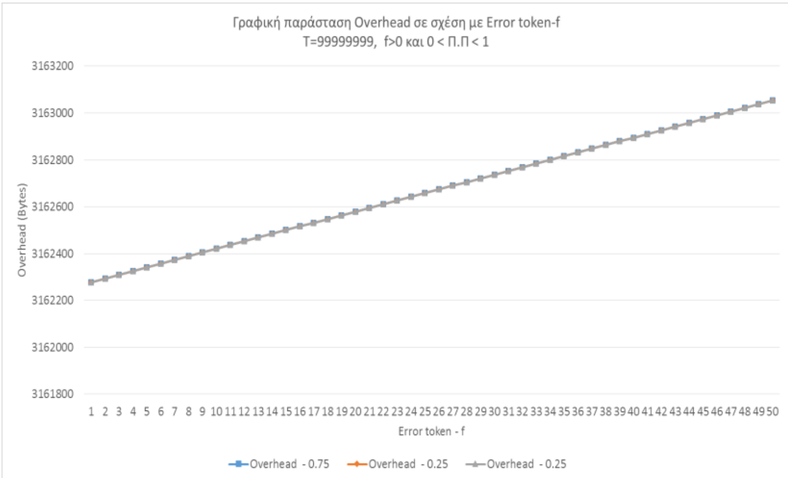
Στο Σχήματος 3.136 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται. Επίσης αντιλαμβανόμαστε ότι η μείωση του goodput αρχικά είναι σχετικά γρήγορη, όμως καθώς αυξάνουμε τα διαθέσιμα *error token*το goodput μειώνεται με πιο αργούς ρυθμούς. Επίσης εύκολα παρατηρούμε ότι για αυτό το ρυθμό επαναφόρτισης είναι λίγο



Σχήμα 3.136

εμφανείς στην αρχή η διαφορά στο goodput. Όμως στη συνέχεια η διαφορά αυτή γίνεται πολύ μικρή, για να την αντιληφθεί κάποιος πρέπει να ελέγξει τους πίνακες. Αυτό είναι λογικό αφού όταν μεγαλώσουμε το *f* στέλνονται πακέτα μικρότερου μεγέθους. Επίσης λόγω του ότι η Π.Π δεν είναι 100% => δεν χρησιμοποιούνται όλα τα *error token*όταν αυξήσουμε το *f*.

Overhead

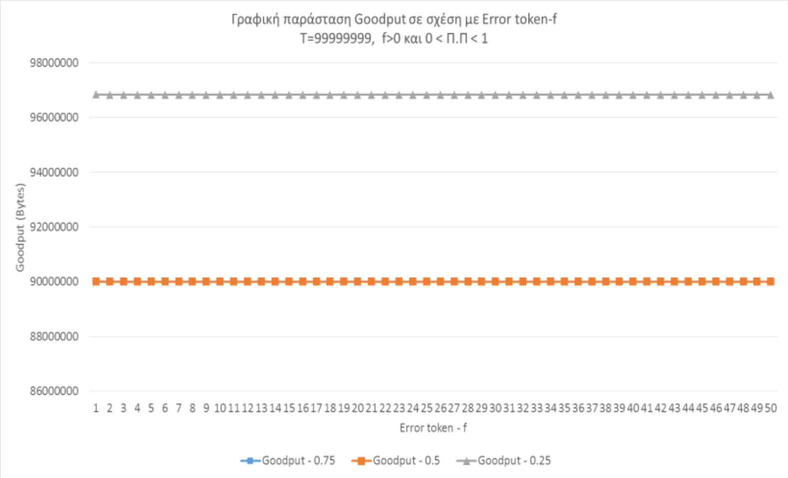


Σχήμα 3.137

Στο Σχήματος 3.137 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό αρχικά γρήγορα ενώ στην συνέχεια τείνει να γίνει γραμμική η αύξηση. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

r = 0.01

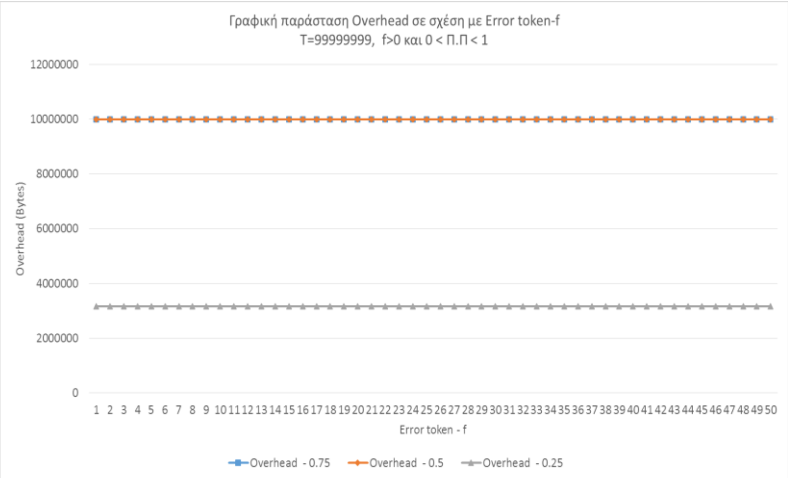
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.138

Στο Σχήματος 3.138 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με αργούς ρυθμούς (γραμμική μείωση). Επίσης εύκολα παρατηρούμε ότι για αυτό το ρυθμό επαναφόρτισης είναι λίγο εμφανείς στην αρχή η διαφορά στο goodput. Όμως στη συνέχεια η διαφορά αυτή γίνεται πολύ μικρή, για να την αντιληφθεί κάποιος πρέπει να ελέγξει τους πίνακες. Αυτό είναι λογικό αφού όταν μεγαλώσουμε το *f* στέλνονται πακέτα μικρότερου μεγέθους. Επίσης λόγω του ότι η Π.Π δεν είναι 100% => δεν χρησιμοποιούνται όλα τα *error token*όταν αυξήσουμε το *f*.

<u>Overhead</u>

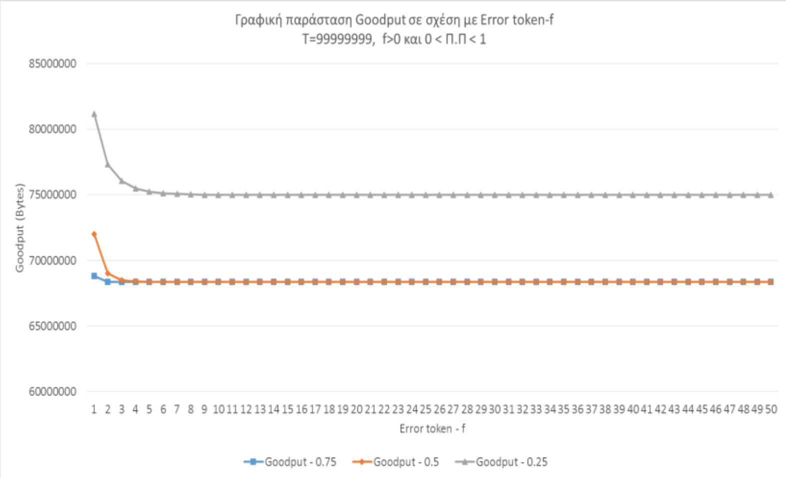


Σχήμα 3.139

Στο Σχήματος 3.139 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

r = 0.1

<u>Ωφέλιμο φορτίο – Goodput</u>

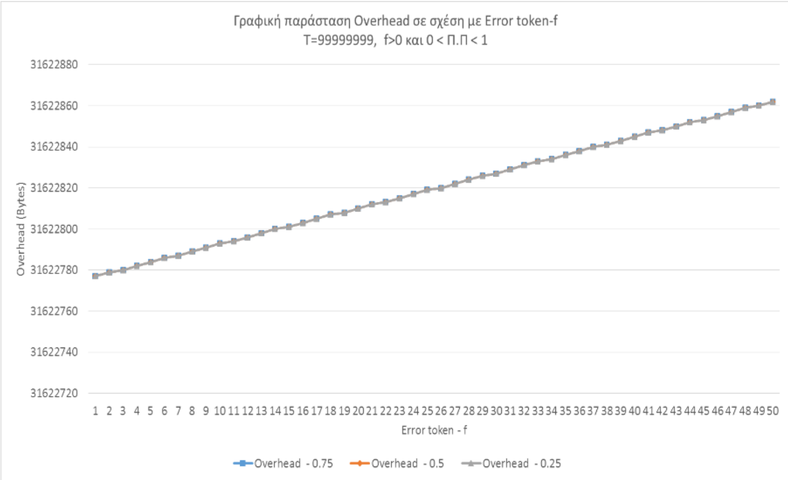


Σχήμα 3.140

Στο Σχήματος 3.140 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις/παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με αργούς ρυθμούς (γραμμική μείωση). Επίσης εύκολα παρατηρούμε ότι για αυτό το ρυθμό επαναφόρτισης είναι εμφανές ότι για μικρή Π.Π έχουμε πολύ καλύτερα αποτελέσματα. Αυτό οφείλεται στο γεγονός ότι ο ρυθμός επαναφόρτισης είναι πάρα πολύ μικρός, έχει ως φυσικό επακόλουθο το μέγεθος των πακέτων είναι αρκετά μικρό. Άρα όταν η Π.Π είναι μικρή αναμένουμε ότι θα είναι εμφανείς η διαφορά στην καταστροφή των πακέτων μιας και αυτά είναι πολύ μικρά. Επίσης λόγω του ότι η Π.Π δεν είναι 100% => δεν χρησιμοποιούνται όλα τα *error token*όταν αυξήσουμε το *f*.

<u>Overhead</u>

Στο Σχήματος 3.141 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για διαφορετικές Πιθανότητες Παρεμβολής (Π.Π). Εύκολα παρατηρούμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό γραμμικά. Αυτό είναι λογικό γιατί ό αλγόριθμος λαμβάνει υπόψη το *f* ώστε να υπολογίσει το μήκος του

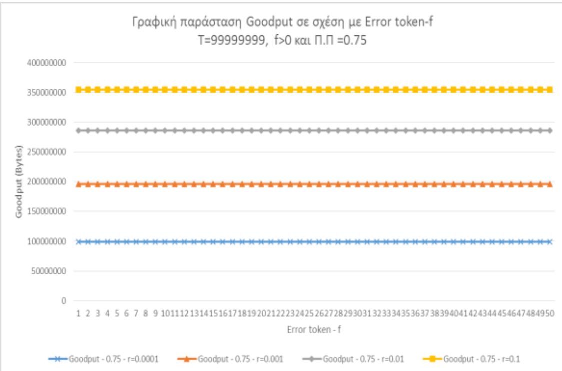


Σχήμα 3.141

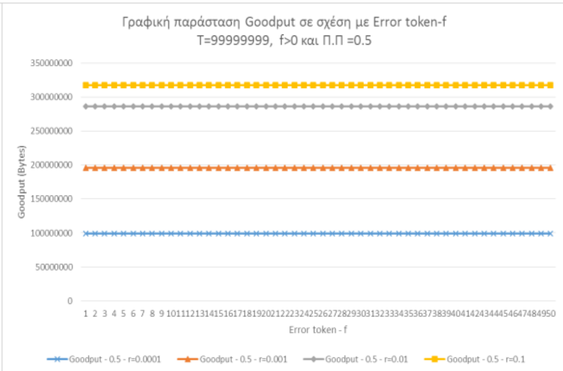
πακέτου. Μεγαλύτερο *f* => μικρότερο μήκος πακέτου => περισσότερα πακέτα για αποστολή. Επίσης εύκολα παρατηρούμε ότι η Π.Π δεν επηρεάζει το overhead. Κάτι το οποίο ήταν και αναμενόμενο αφού ο Uni*f*orm στέλνει πακέτα ιδίου μεγέθους, που δεν εξαρτώνται από τη Π.Π.

Σύγκριση όλων των r

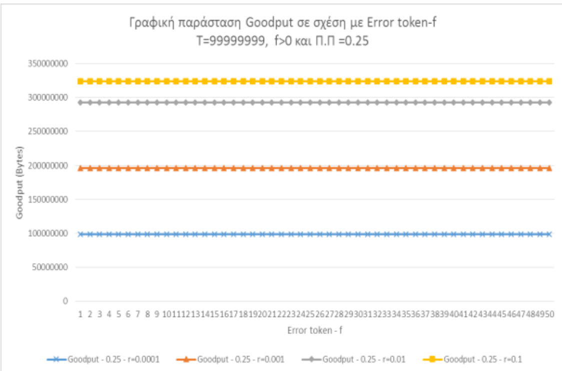
<u>Ωφέλιμο φορτίο – Goodput</u>



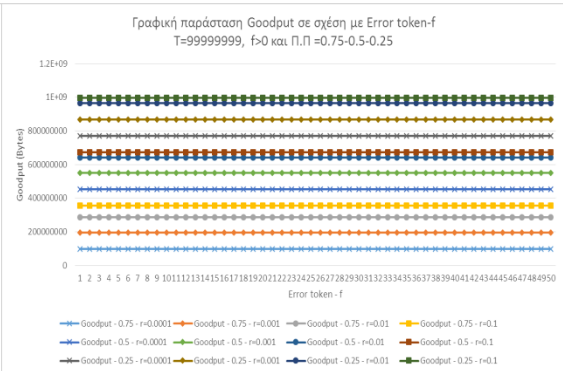
Σχήμα 3.142



Σχήμα 3.143



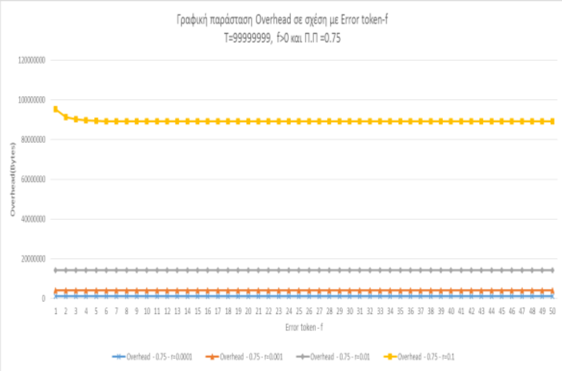
Σχήμα 3.144



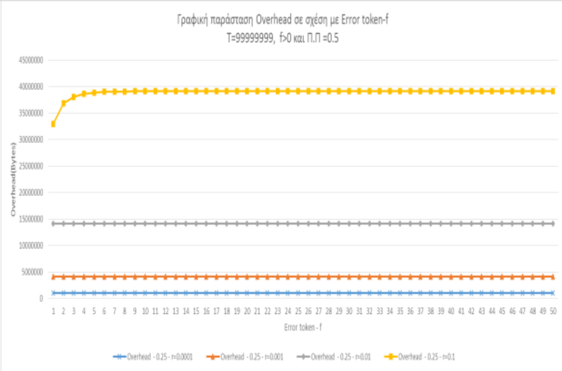
Σχήμα 3.145

Με την βοήθεια των πιο πάνω σχημάτων (3.142, 3.143, 3.144 και 3.145) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο λιγότερο ωφέλιμο φορτίο έχουμε.

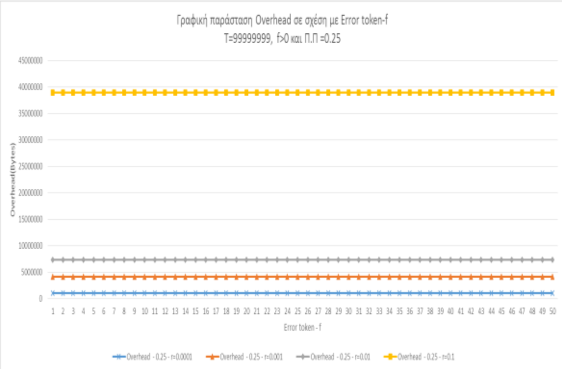
Overhead



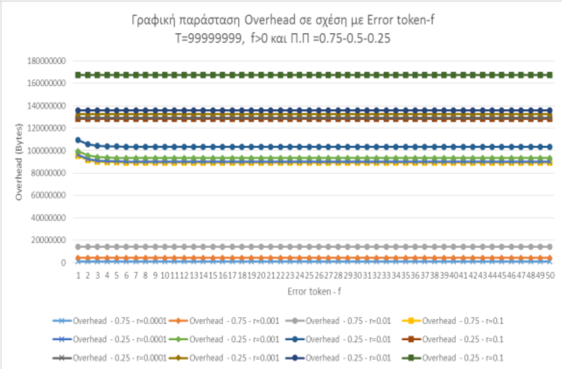
Σχήμα 3.146



Σχήμα 3.147



Σχήμα 3.148



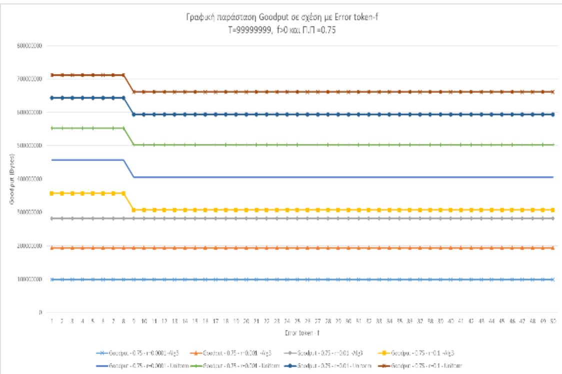
Σχήμα 3.149

Με την βοήθεια των πιο πάνω σχημάτων (3.146, 3.147, 3.148 και 3.149) ευκολά βλέπουμε όσο μεγαλύτερη είναι η Π.Π τόσο μεγαλύτερο είναι το Overhead.

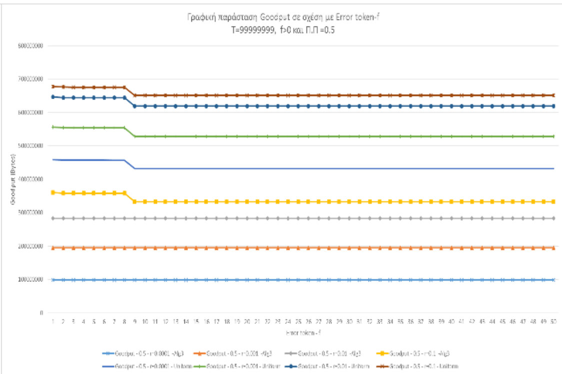
Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 VS Uniform – Γραφικές για T = 99999999

Ωφέλιμο φορτίο – Goodput

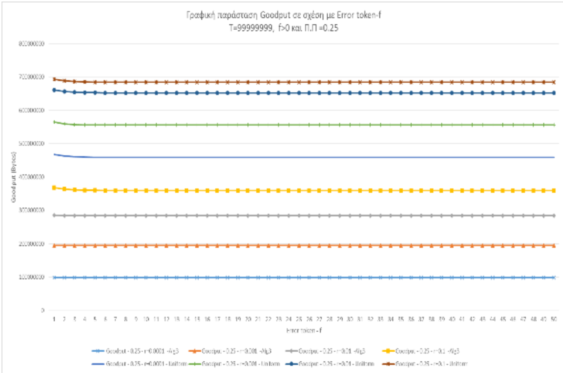
Με την βοήθεια των πιο πάνω σχημάτων (3.150, 3.151, 3.152 και 3.154) εύκολα βλέπουμε όσο μεγαλύτερη είναι η Π.Π σε συνδυασμό με το ρυθμό επαναφόρτισης τόσο λιγότερο ωφέλιμο φορτίο έχουμε.



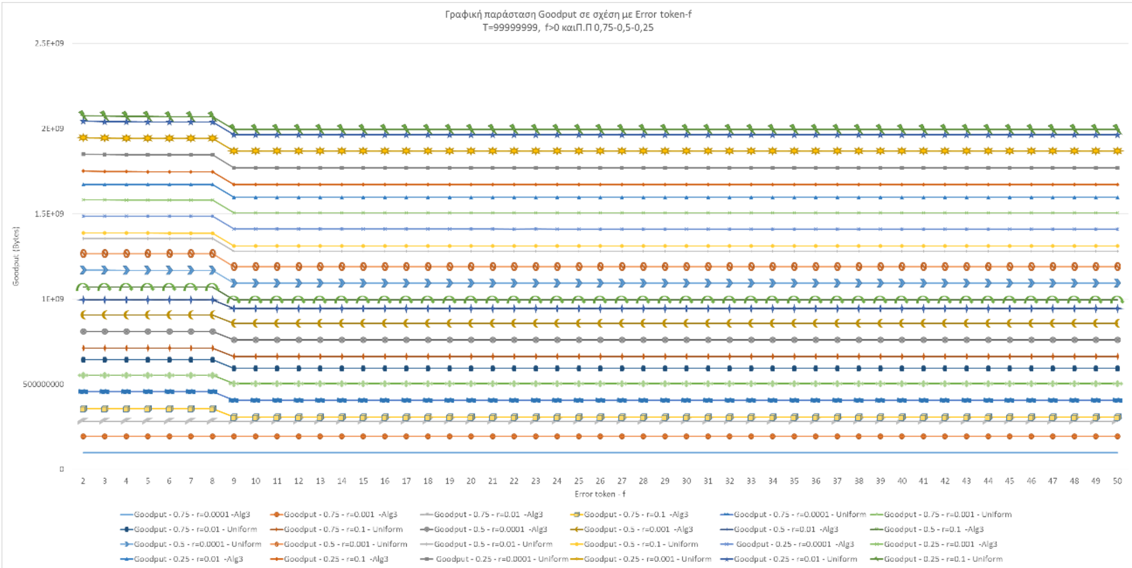
Σχήμα 3.150



Σχήμα 3.151

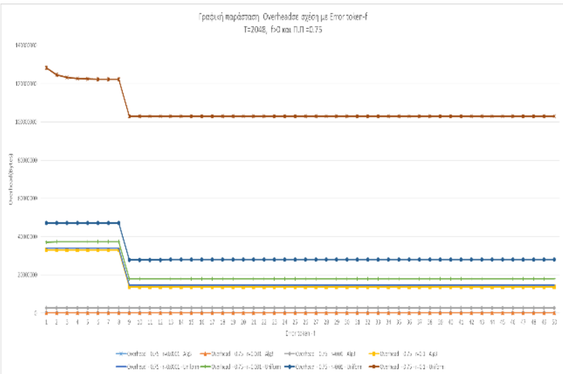


Σχήμα 3.152

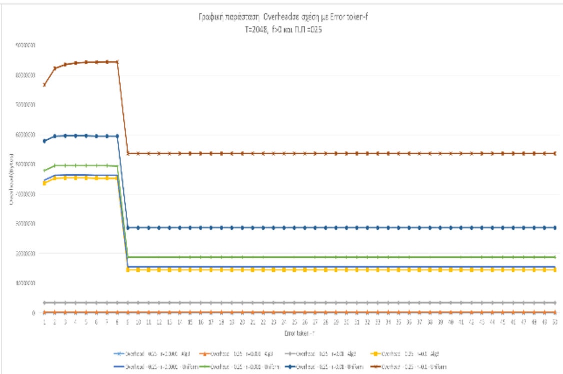


Σχήμα 3.154

Overhead

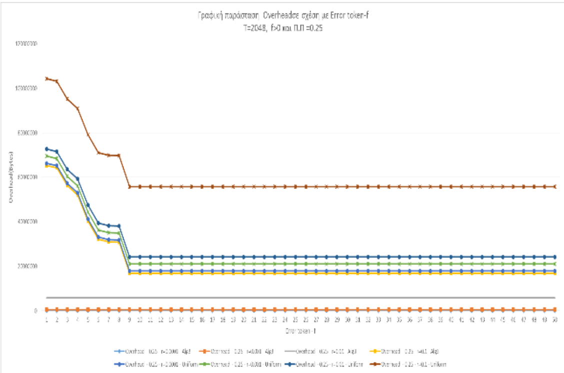


Σχήμα 3.155



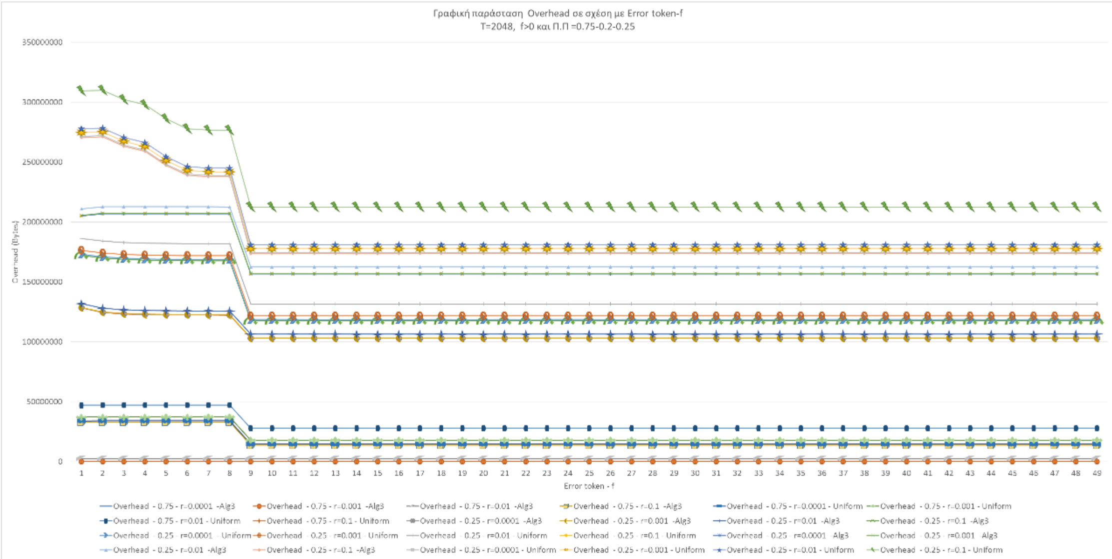
Σχήμα 3.156

Με την βοήθεια των πιο πάνω σχημάτων (3.155, 3.156, 3.157 και 3.158) εύκολα βλέπουμε όσο μεγαλύτερη είναι η Π.Π σε συνδυασμό με το ρυθμό επαναφόρτισης τόσο



Σχήμα 3.157

μεγαλύτερο overhead έχουμε. Αυτό είναι λογικό γιατί δεν υπάρχουν περιπτώσεις που δεν χρησιμοποιούμε όλα τα *error token*γρήγορα => περισσότερα πακέτα μικρότερου μήκους.

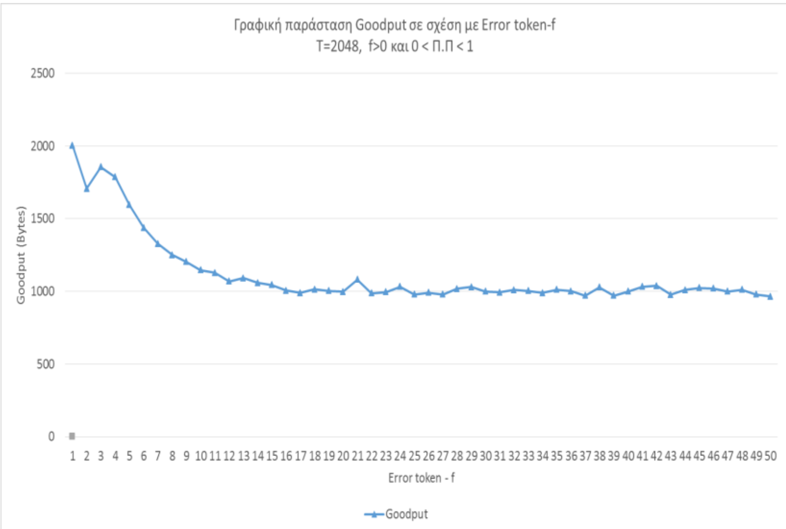


Σχήμα 3.158

Αποτελέσματα για σενάριο 4

Αποτελέσματα για αλγόριθμο ADP(*T*,*f*) με *f* >1 για T = 2048

<u>Ωφέλιμο φορτίο – Goodput</u>

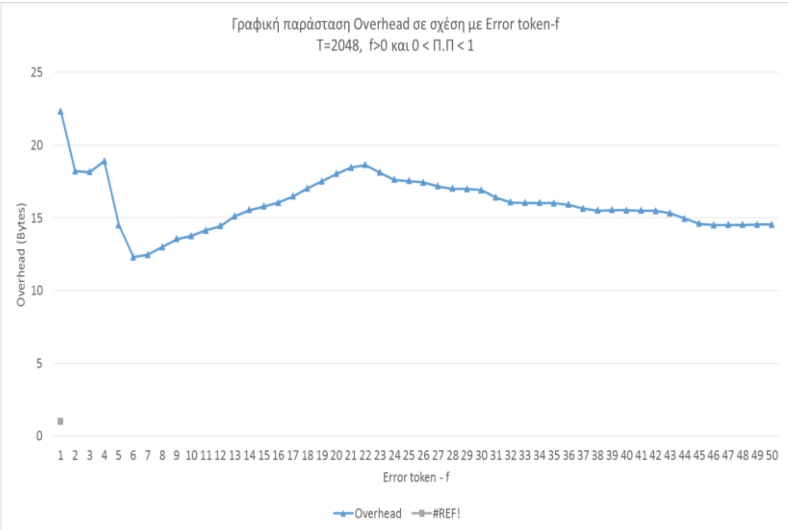


Σχήμα 3.159

Στο Σχήματος 3.159 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες

Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput παραμένει σχεδόν σταθερό, για να είμαστε πιο ακριβείς αυξομειώνετε σε πολύ κοντινές τιμές (αυτό θα μπορούμε να το δούμε ξεκάθαρα στις επόμενες γραφικής παραστάσεις, ιδίως στο σενάριο όπου T = 999999999). Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 42%.

<u>Overhead</u>



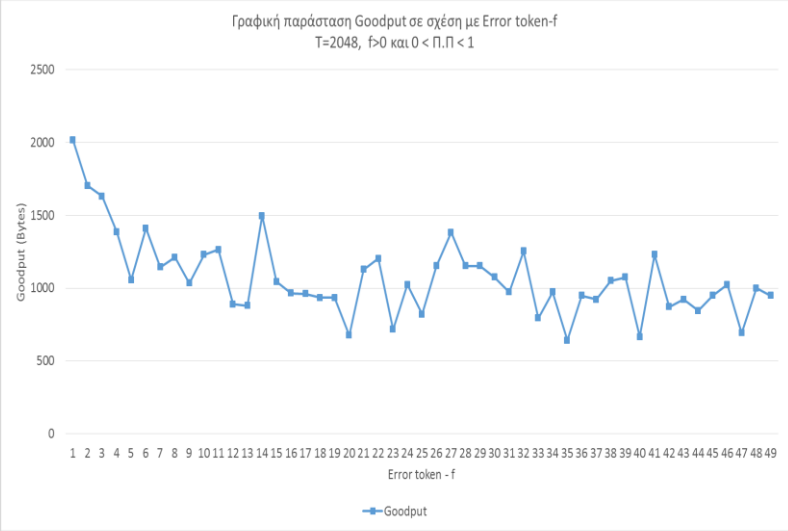
Σχήμα 3.160

Στο Σχήματος 3.160 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead αυξάνεται μέχρι ένα σημείο αφού ο αλγόριθμος καθυστερεί να φτάσει στην τερματική του περίπτωση. Όταν όμως το *f* πάρει τέτοιες τιμές ώστε τα διαθέσιμα *error token*>= εναπομείναντα φορτίο, εύκολα διακρίνουμε ότι το συνολικό overhead αρχίζει να μειώνεται. Αυτό είναι απολύτως λογικό αφού ο αλγόριθμος στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται.

Αποτελέσματα για αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 για T = 2048

r = 0.0001

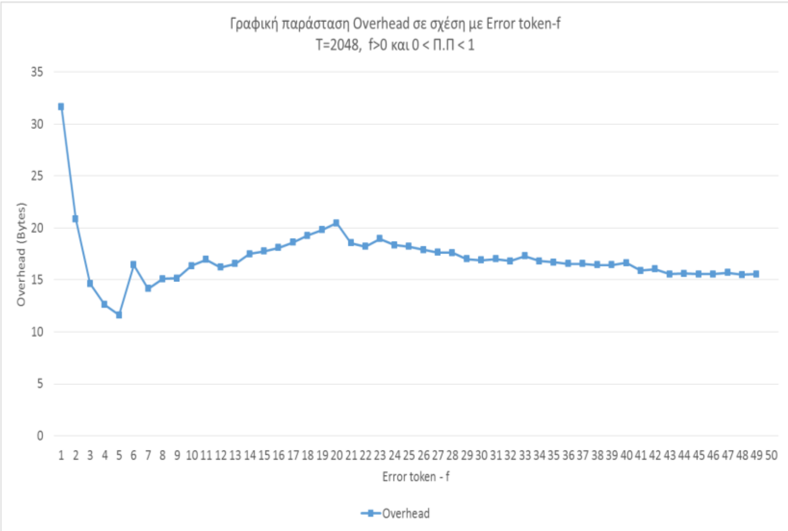
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.161

Στο Σχήματος 3.161 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,41. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput αυξομειώνεται σε κοντινές τιμές. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 41%.

Overhead

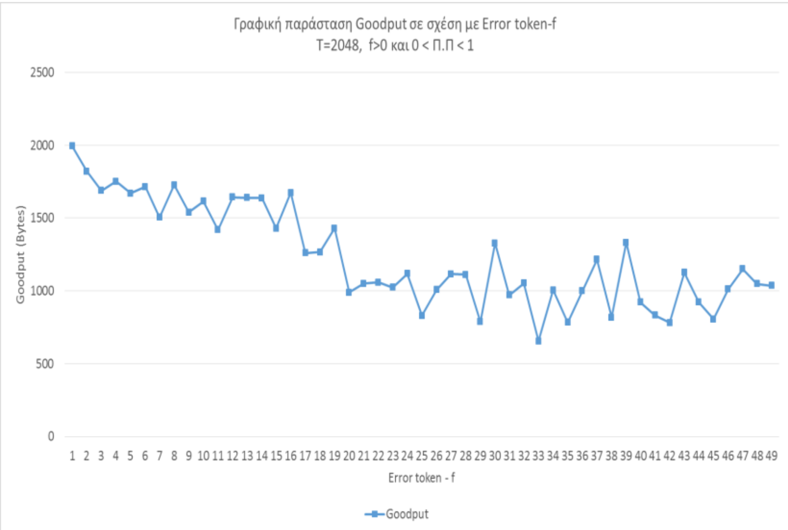


Σχήμα 3.162

Στο Σχήματος 3.162 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead αυξάνεται μέχρι ένα σημείο αφού ο αλγόριθμος καθυστερεί να φτάσει στην τερματική του περίπτωση. Όταν όμως το *f* πάρει τέτοιες τιμές ώστε τα διαθέσιμα *error token*>= εναπομείναντα φορτίο, εύκολα διακρίνουμε ότι το συνολικό overhead αρχίζει να μειώνεται. Αυτό είναι απολύτως λογικό αφού ο αλγόριθμος στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται.

r = 0.001

<u>Ωφέλιμο φορτίο – Goodput</u>

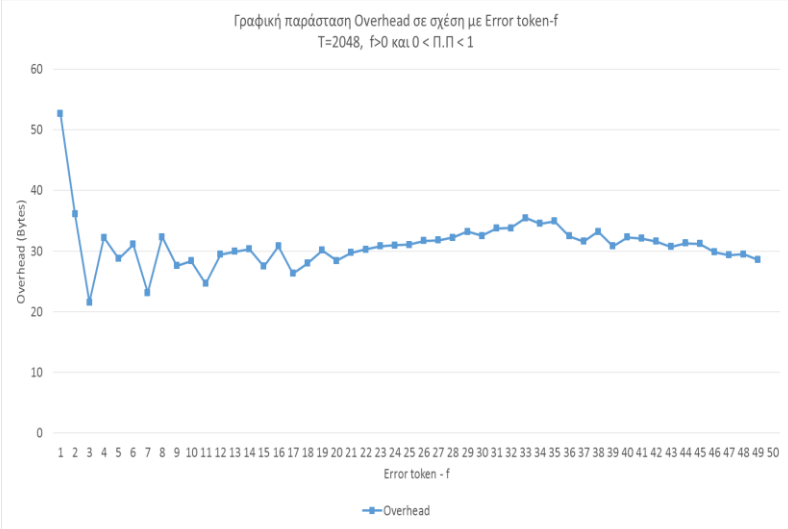


Σχήμα 3.163

Στο Σχήματος 3.163 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,41. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput αυξομειώνεται σε κοντινές τιμές. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 41%.

<u>Overhead</u>

Στο Σχήματος 3.164 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,41.

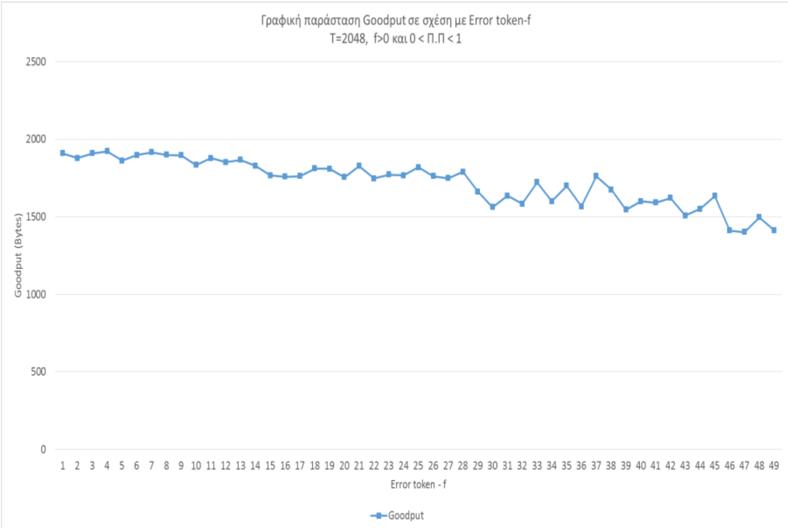


Σχήμα 3.164

Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead αυξάνεται μέχρι ένα σημείο αφού ο αλγόριθμος καθυστερεί να φτάσει στην τερματική του περίπτωση. Όταν όμως το *f* πάρει τέτοιες τιμές ώστε τα διαθέσιμα *error token*>= εναπομείναντα φορτίο, εύκολα διακρίνουμε ότι το συνολικό overhead αρχίζει να μειώνεται. Αυτό είναι απολύτως λογικό αφού ο αλγόριθμος στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται.

r = 0.01

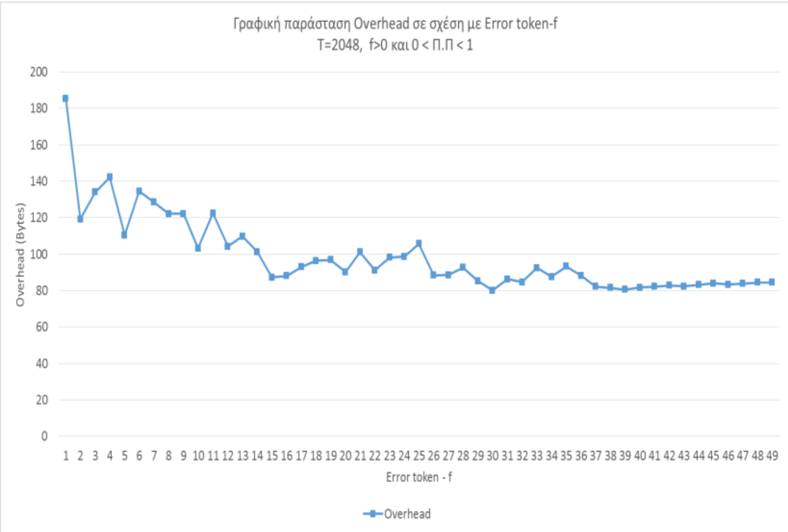
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.165

Στο Σχήματος 3.165 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με αργούς ρυθμούς λόγο του πολύ μικρού ρυθμού επαναφόρτισης. Ακολούθως όταν χρησιμοποιείται και ο ADP2 οι αυξομειώσεις αρχίζουν να γίνονται εμφανείς. Αυτό οφείλεται επίσης στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 42%.

Overhead



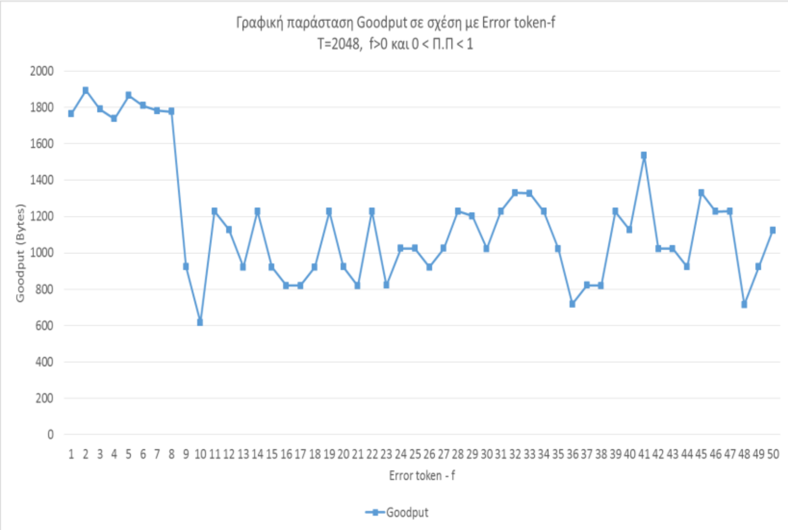
Σχήμα 3.166

Στο Σχήματος 3.166 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π є [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead αυξάνεται μέχρι ένα σημείο αφού ο αλγόριθμος καθυστερεί να φτάσει στην τερματική του περίπτωση. Όταν όμως το *f* πάρει τέτοιες τιμές ώστε τα διαθέσιμα *error token*>= εναπομείναντα φορτίο, εύκολα διακρίνουμε ότι το συνολικό overhead αρχίζει να μειώνεται. Αυτό είναι απολύτως λογικό αφού ο αλγόριθμος στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται.

r = 0.1

Ωφέλιμο φορτίο – Goodput

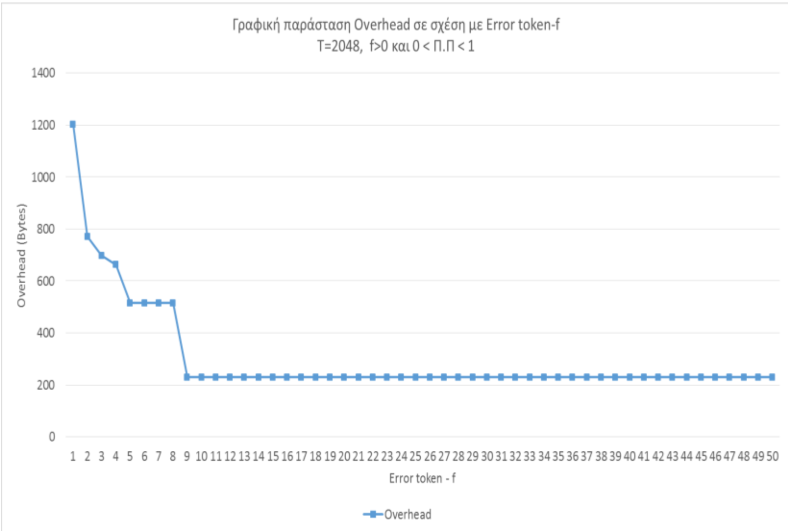
Στο Σχήματος 3.167 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π є [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,40. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με αργούς ρυθμούς λόγο του πολύ μικρού ρυθμού επαναφόρτισης. Ακολούθως όταν



Σχήμα 3.167

χρησιμοποιείται και ο ADP2 οι αυξομειώσεις αρχίζουν να γίνονται εμφανείς. Αυτό οφείλεται επίσης στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 40%.

Overhead

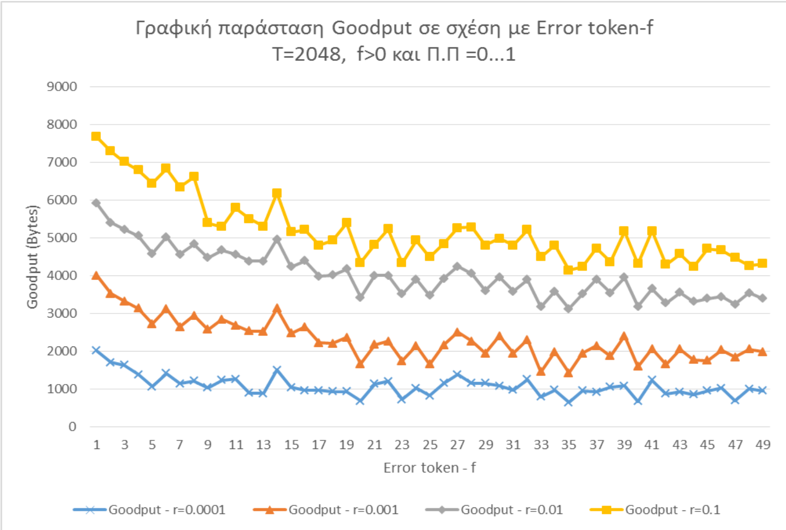


Σχήμα 3.168

Στο Σχήματος 3.168 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead αυξάνεται μέχρι ένα σημείο αφού ο αλγόριθμος να φτάσει στην τερματική του περίπτωση. Όταν όμως το *f* (εδώ *f* = 9) πάρει τέτοιες τιμές ώστε τα διαθέσιμα *error token*>= εναπομείναντα φορτίο, εύκολα διακρίνουμε ότι το συνολικό overhead μειώνεται και έπειτα σταθεροποιείται. Αυτό είναι απολύτως λογικό αφού ο αλγόριθμος στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται. Η σταθεροποίηση του overhead πραγματοποιείται γρήγορα λόγο και του πολύ μικρού ρυθμού επαναφόρτισης

Σύγκριση όλων των r

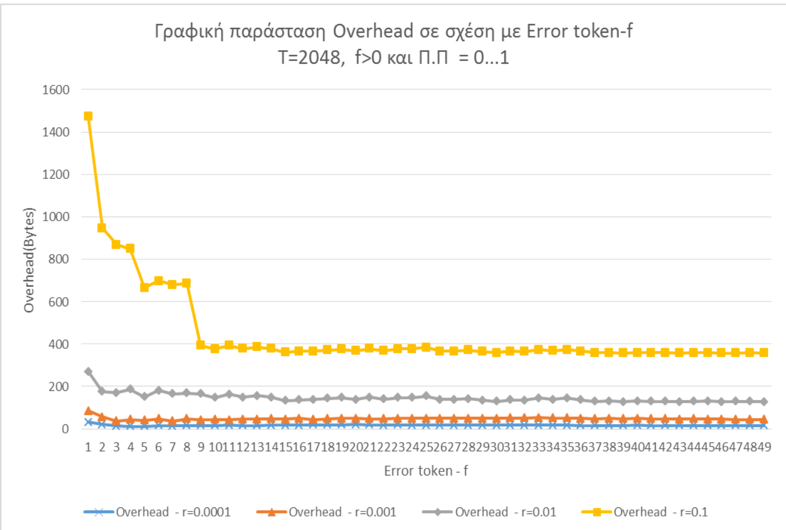
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.169

Με την βοήθεια του σχήματος 3.169 ευκολά βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια ανεξαρτήτως ρυθμού επαναφόρτισης. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο καλύτερα αποτελέσματα έχουμε στο ωφέλιμο φορτίο. Αυτό οφείλεται στο γεγονός ότι η Π.Π δεν είναι 100% και στο ότι μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου.

Overhead



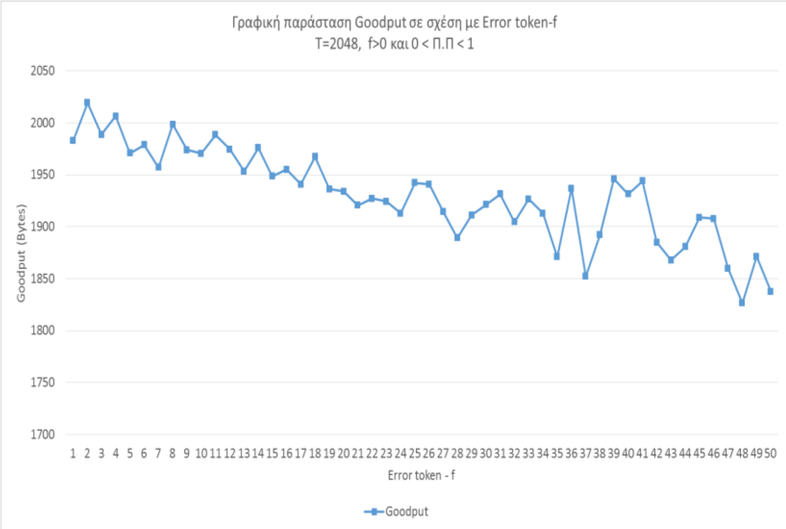
Σχήμα 3.170

Με την βοήθεια του σχήματος 3.170 ευκολά βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια ανεξαρτήτως ρυθμού επαναφόρτισης. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο μεγαλύτερο είναι το overhead. Όπως έχουμε ήδη εξηγήσει μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου => μεγαλύτερος αριθμός πακέτων για αποστολή.

Αποτελέσματα για Uniform για T = 2048

r = 0.0001

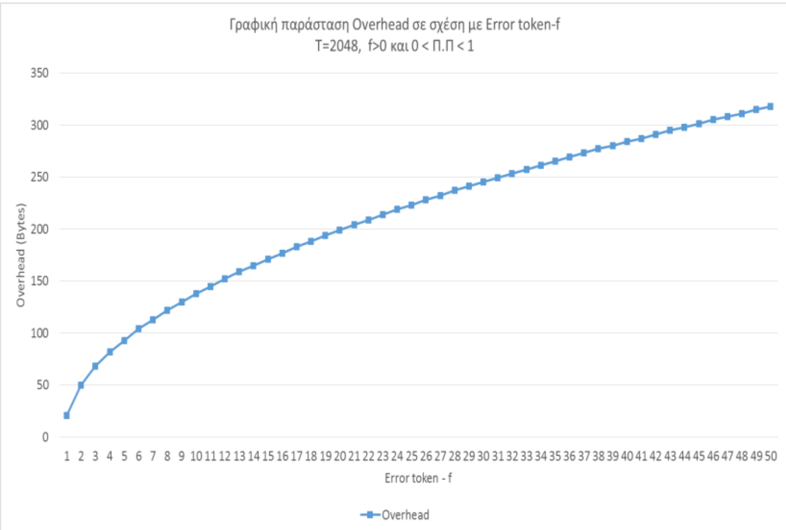
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.171

Στο Σχήματος 3.171 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,40. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται με τάση να μειώνεται με αργούς ρυθμούς. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 40%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση.

<u>Overhead</u>



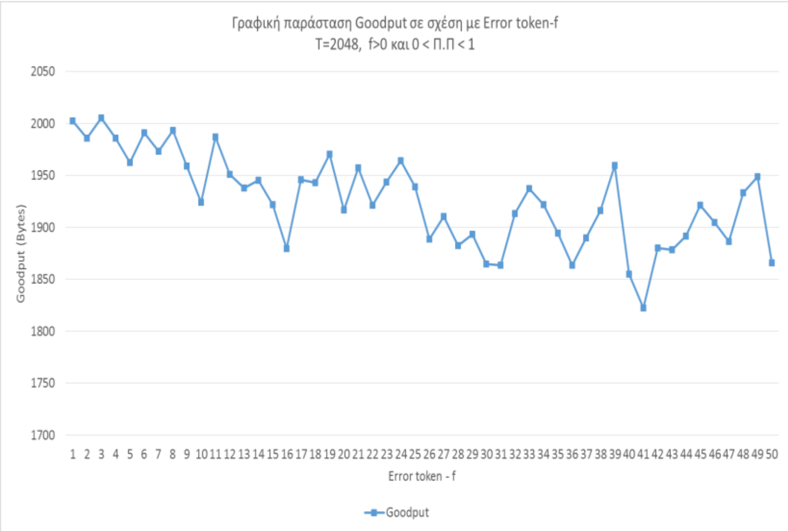
Σχήμα 3.172

Στο Σχήματος 3.172 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες

Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,40. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται γρήγορα ενώ στην συνέχεια αυξάνεται με γραμμικούς ρυθμούς. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

r = 0.001

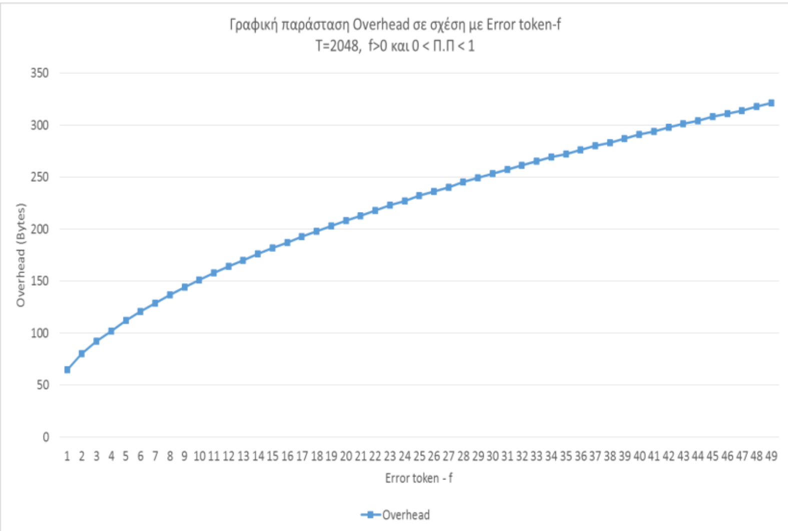
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.173

Στο Σχήματος 3.173 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται με τάση να μειώνεται με αργούς ρυθμούς. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 42%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση.

<u>Overhead</u>

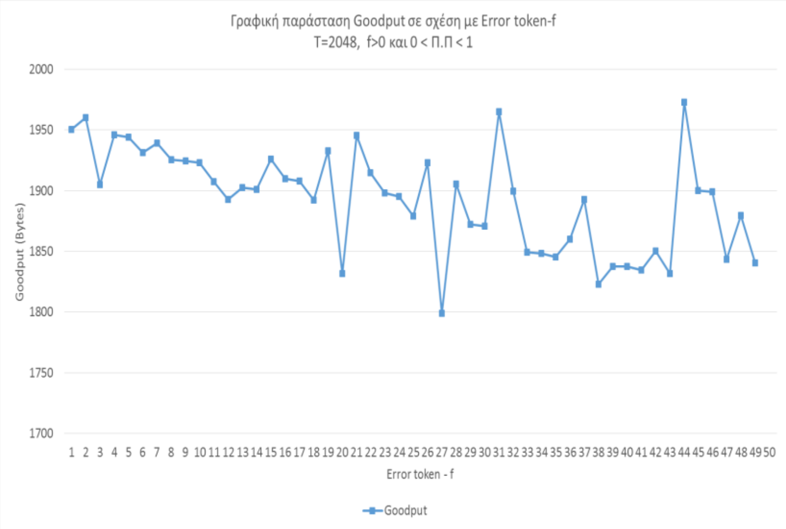


Σχήμα 3.174

Στο Σχήματος 3.174 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,40. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται γρήγορα ενώ στην συνέχεια αυξάνεται με γραμμικούς ρυθμούς. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

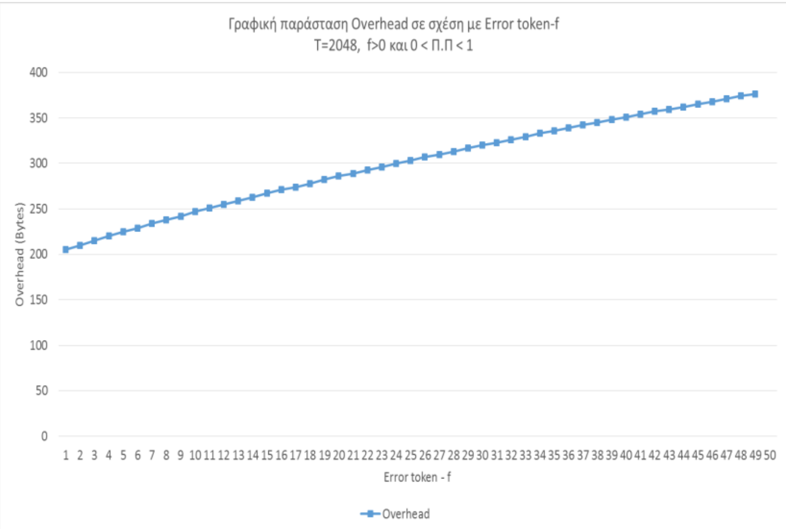
r = 0.01

Ωφέλιμο φορτίο – Goodput



Σχήμα 3.175

Στο Σχήματος 3.175 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται με τάση να μειώνεται με αργούς ρυθμούς. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 42%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση.



Σχήμα 3.176

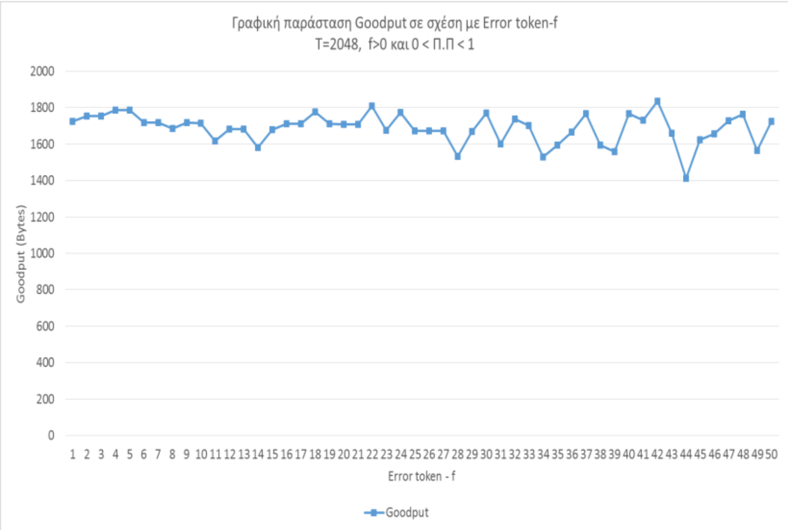
Overhead

Στο Σχήματος 3.176 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες

Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,40. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται γρήγορα ενώ στην συνέχεια αυξάνεται με γραμμικούς ρυθμούς. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

r = 0.1

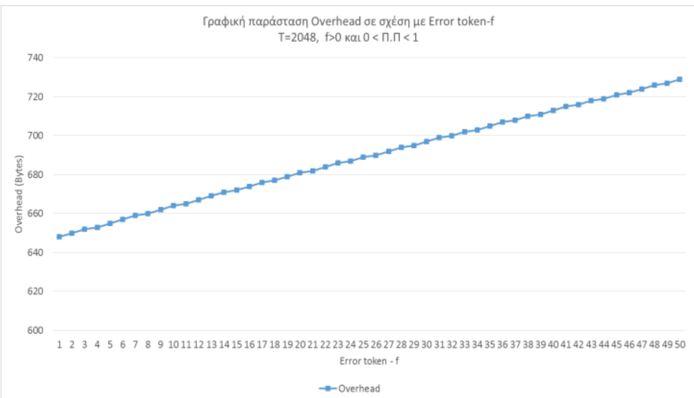
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.177

Στο Σχήματος 3.177 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,44. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται με τάση να μειώνεται με αργούς ρυθμούς. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 44%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση. Επίσης λόγο του πολύ μικρού ρυθμού επαναφόρτισης, στέλνονται πακέτα πολύ μικρού μεγέθους. Έτσι παρατηρούμε ότι το ωφέλιμο φορτίο αυξομειώνεται με αργούς ρυθμούς.

Overhead



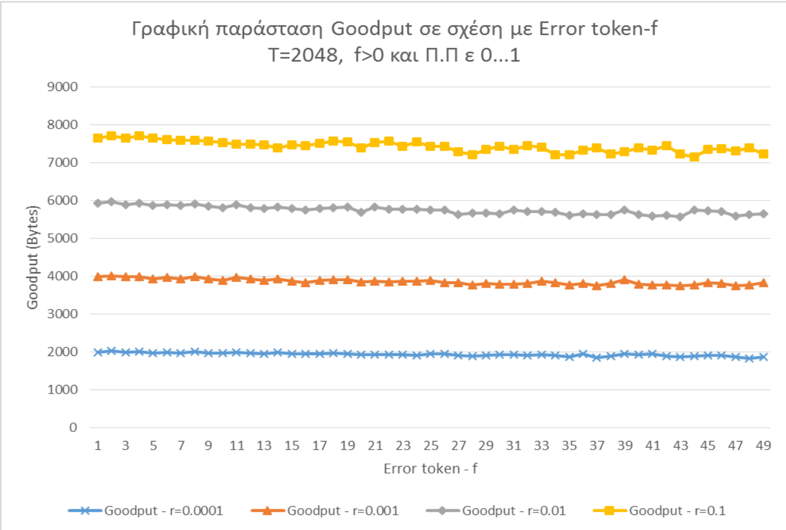
Σχήμα 3.178

Στο Σχήματος 3.178 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ

[0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,44. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται αυξάνεται και αυτό γραμμικά. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

Σύγκριση όλων των r

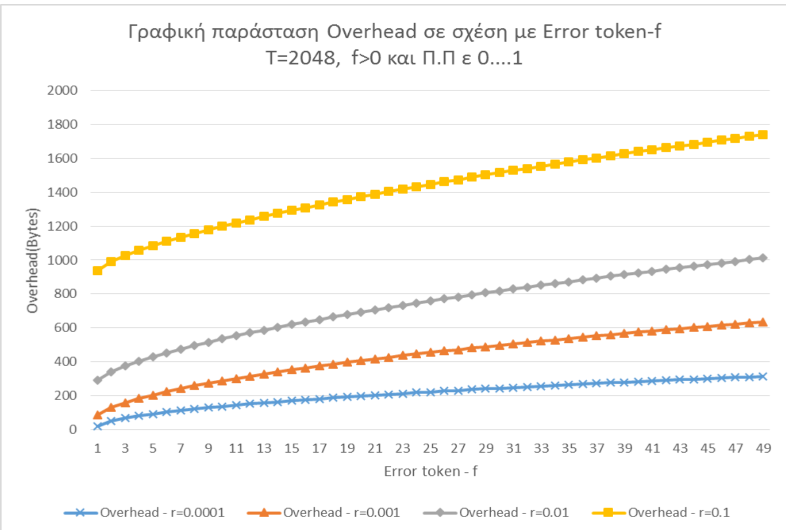
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.179

Με την βοήθεια του σχήματος 3.179 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο καλύτερα αποτελέσματα έχουμε στο ωφέλιμο φορτίο. Αυτό οφείλεται στο γεγονός ότι η Π.Π δεν είναι 100% και στο ότι μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου.

<u>Overhead</u>

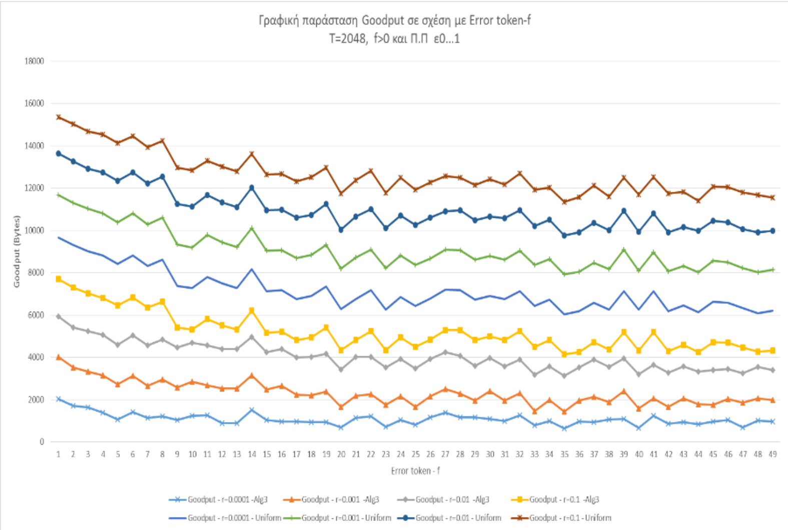


Σχήμα 3.180

Με την βοήθεια του σχήματος 3.180 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια ανεξαρτήτως ρυθμού επαναφόρτισης. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο μεγαλύτερο είναι το overhead. Όπως έχουμε ήδη εξήγηση μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου => μεγαλύτερος αριθμός πακέτων για αποστολή.

Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 VS Uniform – Γραφικές για T = 2048

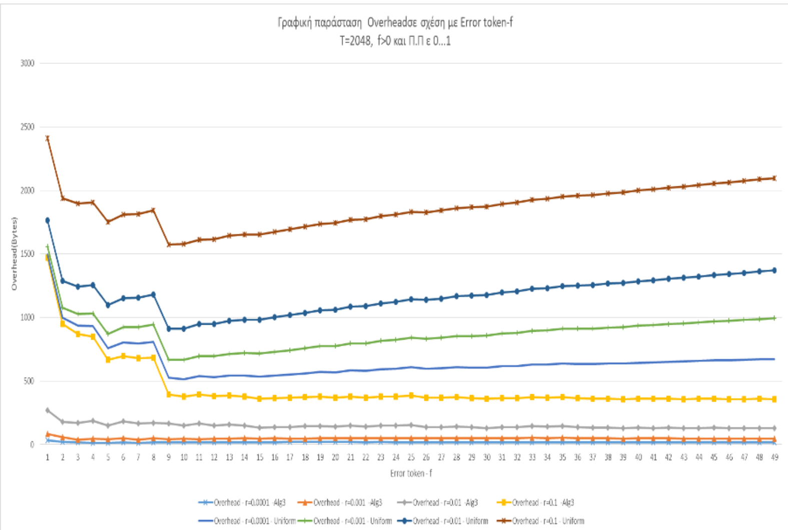
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.181

Με την βοήθεια του σχήματος 3.181 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτιση η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια. Επίσης παρατηρούμε ότι όσο μικρότερος είναι ο ρυθμός επιφόρτισης τόσο καλύτερα αποτελέσματα έχουμε για το ωφέλιμο φορτίο. Επίσης φαίνεται ξεκάθαρα ότι ο αλγόριθμος uni*f*orm αποστέλλει περισσότερο ωφέλιμο φορτίο από ότι ο αλγόριθμος ADP_CHARCHING(*T*,*f*) με f >1.

<u>Overhead</u>

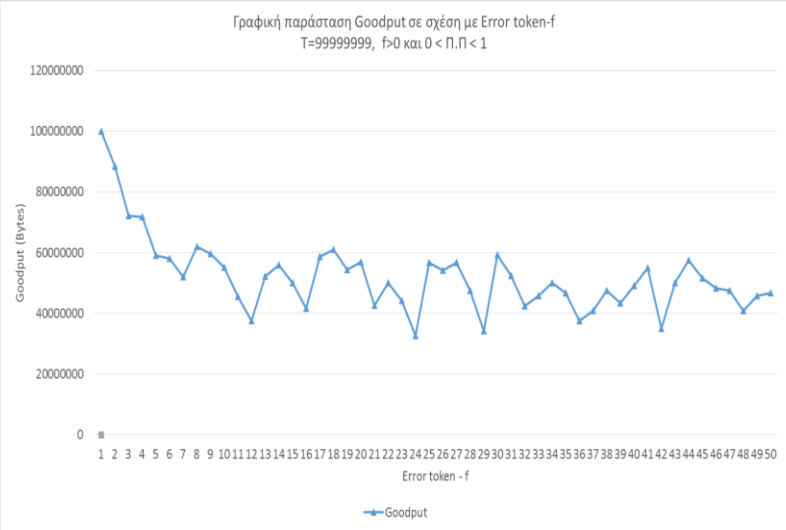


Σχήμα 3.182

Με την βοήθεια του σχήματος 3.182 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτιση η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια για κάθε αλγόριθμο. Επίσης παρατηρούμε ότι όσο μικρότερος είναι ο ρυθμός επιφόρτισης τόσο μικρότερο είναι το overhead για κάθε αλγόριθμο. Επίσης φαίνεται ξεκάθαρα ότι ο αλγόριθμος uni*f*orm σε σχέση με τον αλγόριθμο ADP_ CHARCHING (*T*,*f*) με *f* >1 δημιουργεί αρκετά μεγαλύτερο overhead.

Αποτελέσματα για αλγόριθμο ADP(*T*,*f*) με *f* >1 για T = 99999999

<u>Ωφέλιμο φορτίο – Goodput</u>

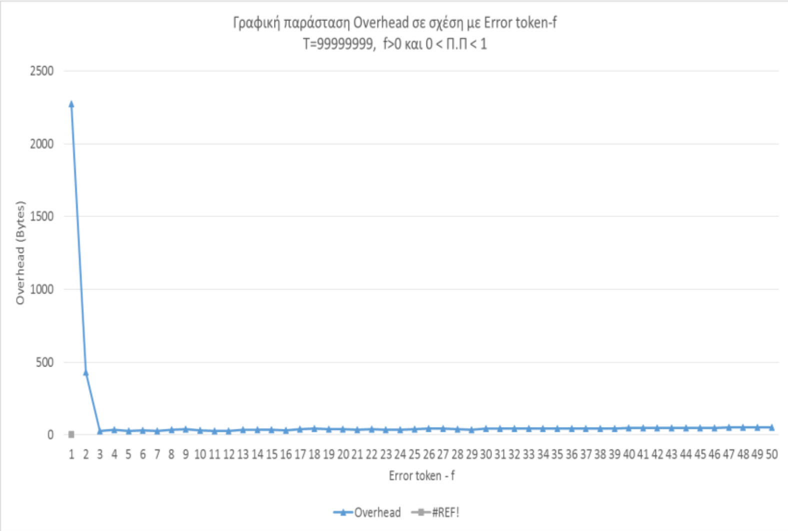


Σχήμα 3.183

Στο Σχήματος 3.183 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π є [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput μειώνεται με γρήγορους ρυθμούς αρχικά. Όμως αυτό συμβαίνει για μικρές τιμές του *f* όπου χρησιμοποιείται κατά κύριο λόγο ο αλγόριθμος ADP1. Στη συνέχεια παρατηρούμε ότι το goodput παραμένει σχεδόν σταθερό, για να είμαστε πιο ακριβείς αυξομειώνετε σε πολύ κοντινές τιμές. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 42%.

<u>Overhead</u>

Στο Σχήματος 3.184 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π є [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,42. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead αυξάνεται γραμμικά με αργούς ρυθμούς, μέχρι ένα σημείο αφού ο αλγόριθμος καθυστερεί να φτάσει στην

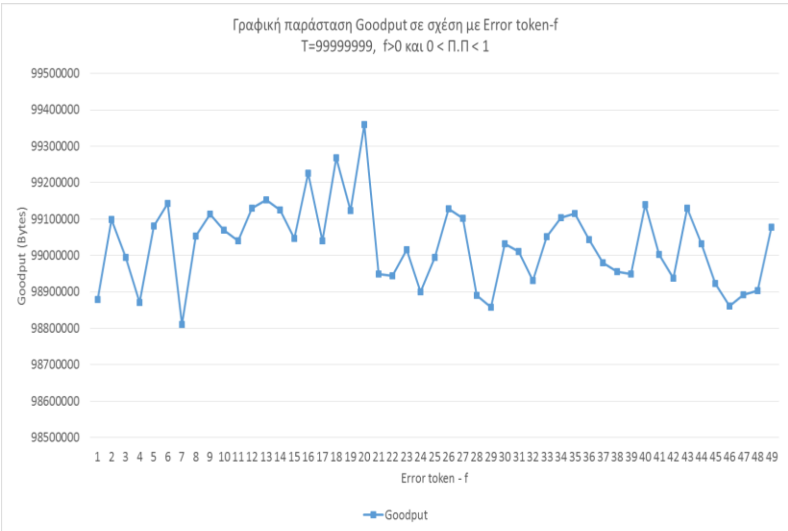


Σχήμα 3.184

τερματική του περίπτωση. Όταν όμως το *f* πάρει τέτοιες τιμές ώστε τα διαθέσιμα *error token* >= εναπομείναντα φορτίο, εύκολα διακρίνουμε ότι το συνολικό overhead αρχίζει να μειώνεται. Αυτό είναι απολύτως λογικό αφού ο αλγόριθμος στέλνει ένα μεγαλύτερο πακέτο => άρα μειώνεται ο συνολικός αριθμός πακέτων που στέλνονται.

Αποτελέσματα για αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 για T = 99999999 r = 0.0001

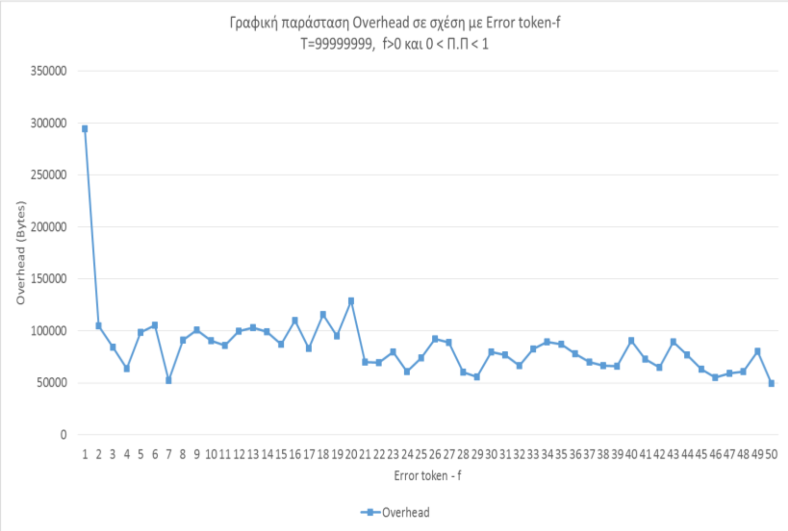
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.185

Στο Σχήματος 3.185 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,61. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται σε κοντινές τιμές. Αυτό οφείλεται στο γεγονός ότι παρόλο που αυξάνουμε το *f*, ο παρεμβολές δεν χρησιμοποιεί όλα τα διαθέσιμα error token. Αυτό είναι απολύτως λογικό, αφού η Π.Π δεν είναι 100%. Όπως έχουμε ήδη προαναφέρει εδώ ο μέσος όρος της Π.Π είναι 61%.

Overhead

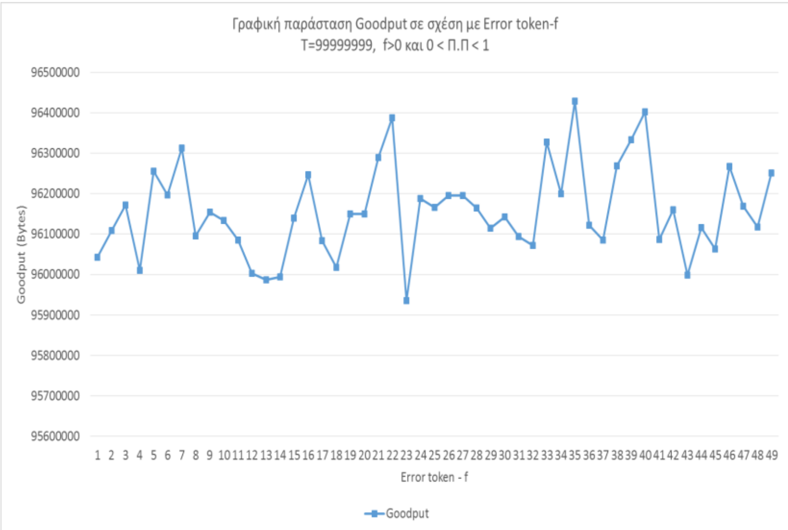


Σχήμα 3.186

Στο Σχήματος 3.186 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,61. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* το overhead αυξομειώνεται με όχι τόσο σταθερό ρυθμό. Αυτό οφείλεται στο γεγονός ότι ο μέσος όρος της Π.Π κυμαίνεται γύρο στο 61%. Σε συνδυασμό με τον ρυθμό επαναφόρτισης και το εναπομείναντα φορτίο κάθε φορά παρατηρούμε αυτή τη συμπεριφορά.

r = 0.001

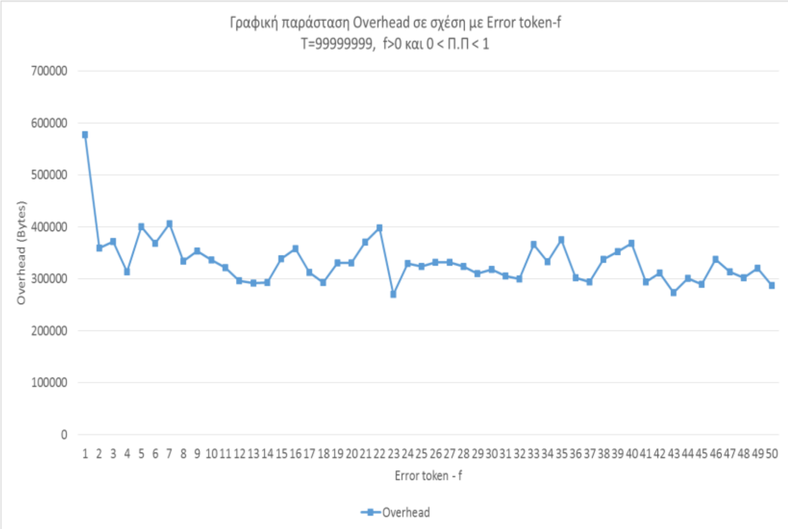
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.187

Στο Σχήματος 3.187 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π. Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,68. Παρατηρούμε μια αυξομείωση στο ωφέλιμο φορτίο. Αυτό οφείλεται στο γεγονός ότι ο μέσος όρος της Π.Π κυμαίνεται γύρο στο 68%. Σε συνδυασμό με τον ρυθμό επαναφόρτισης που είναι σχετικά μικρός και το εναπομείναντα φορτίο κάθε φορά παρατηρούμε αυτή τη συμπεριφορά.

<u>Overhead</u>



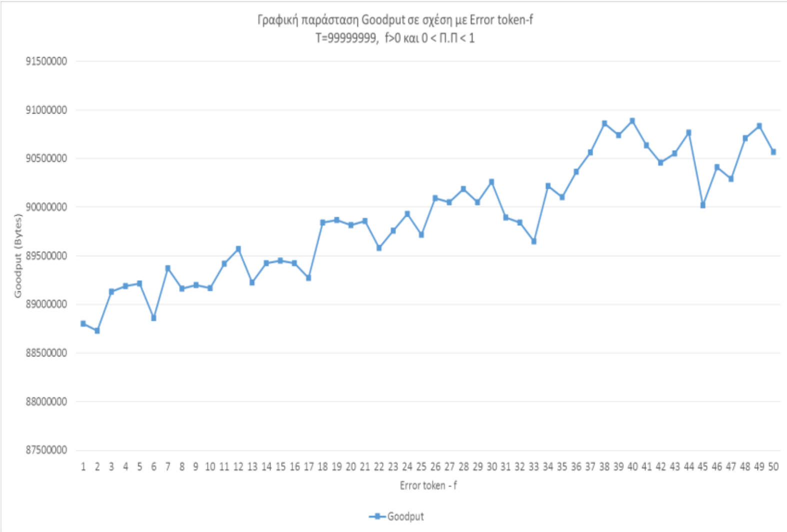
Σχήμα 3.188

Στο Σχήματος 3.188 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,68. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* το overhead αυξομειώνεται με όχι τόσο σταθερό ρυθμό. Αυτό οφείλεται στο γεγονός ότι ο μέσος όρος της Π.Π κυμαίνεται γύρο στο 68%. Σε συνδυασμό με τον ρυθμό επαναφόρτισης και το εναπομείναντα φορτίο κάθε φορά παρατηρούμε αυτή τη συμπεριφορά.

r = 0.01

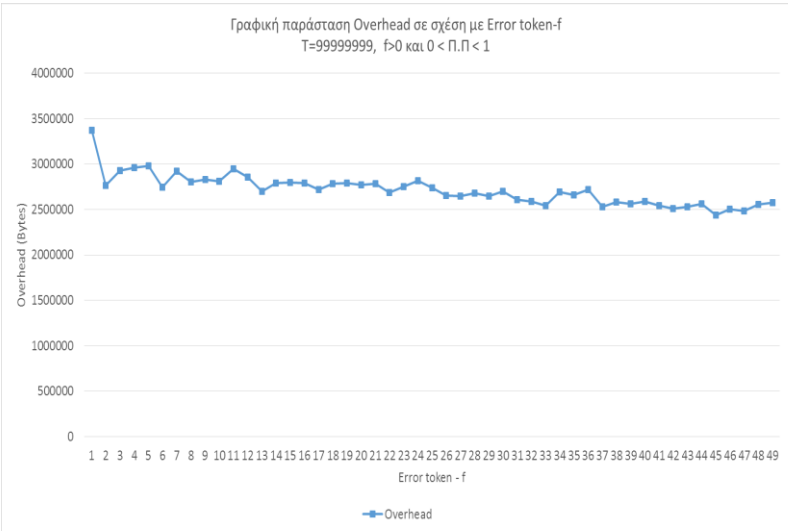
<u>Ωφέλιμο φορτίο – Goodput</u>

Στο Σχήματος 3.189 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π. Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Παρατηρούμε μια αυξομείωση στο ωφέλιμο φορτίο με τάση να αυξάνεται. Κάτι το οποίο εκ πρώτης όψεως δεν φαίνεται αρκετά λογικό. Αυτό οφείλεται στο γεγονός ότι ο μέσος όρος της Π.Π κυμαίνεται γύρο στο 68%. Σε συνδυασμό με τον ρυθμό επαναφόρτισης που είναι σχετικά μικρός και το εναπομείναντα φορτίο κάθε φορά παρατηρούμε αυτή τη συμπεριφορά. Μπορούμε να δούμε καλύτερα τη συμπεριφορά του αλγόριθμου στον επόμενο ρυθμό επαναφόρτισης.



Σχήμα 3.189

<u>Overhead</u>

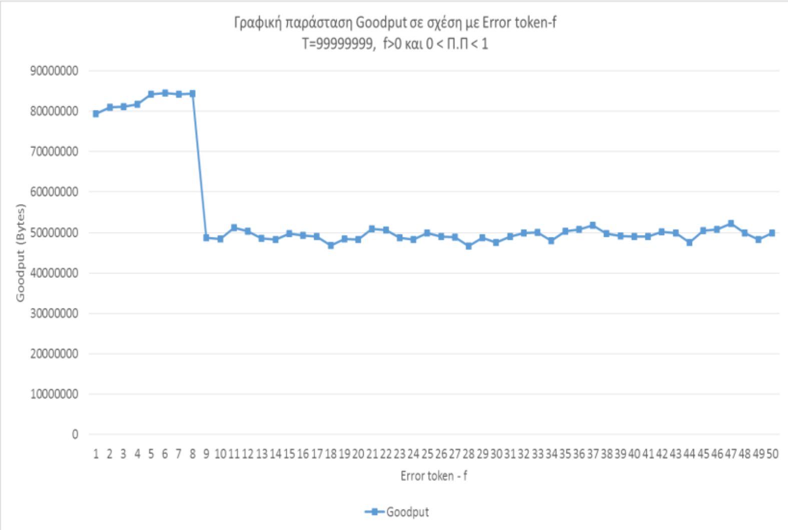


Σχήμα 3.190

Στο Σχήματος 3.190 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* το overhead αυξομειώνεται. Αυτό οφείλεται στο γεγονός ότι ο μέσος όρος της Π.Π κυμαίνεται γύρο στο 68%. Σε συνδυασμό με τον ρυθμό επαναφόρτισης και το εναπομείναντα φορτίο κάθε φορά παρατηρούμε αυτή τη συμπεριφορά.

r = 0.1

Ωφέλιμο φορτίο – Goodput

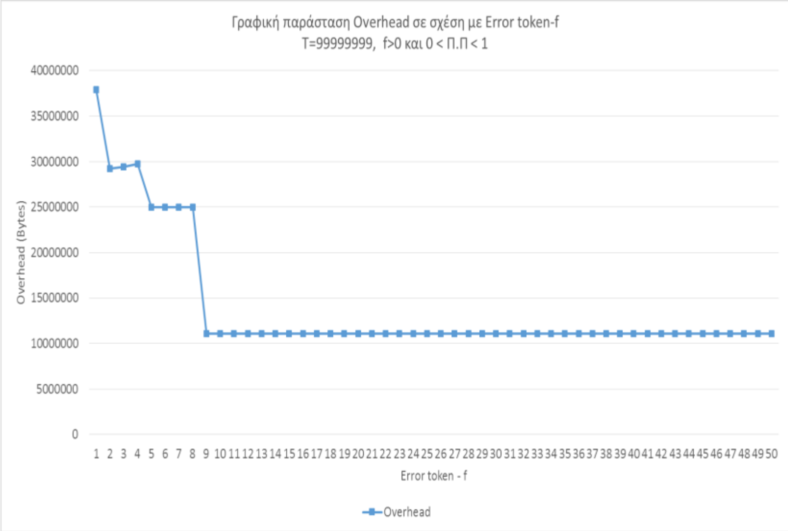


Σχήμα 3.191

Στο Σχήματος 3.191 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αρχικά αυξάνεται, ενώ στην συνέχεια μειώνεται και έπειτα αυξομειώνεται σε κοντινά επίπεδα. Αυτό οφείλεται στο γεγονός ότι καθώς αυξάνουμε το *f*, το μήκος του πακέτου μειώνεται σε συνδυασμό με την πολύ μικρή χρονική περίοδο. Έτσι σε συνδυασμό με το ότι η ο μέσος όσος Π.Π κυμαίνεται στο 67%, δεν καταστρέφονται πάντα τα πρώτα πακέτα που είναι μεγαλύτερου μήκους. Επίσης αυξάνοντας το *f* αρχίζει να χρησιμοποιείται περισσότερο και ο αλγόριθμος ADP2. Τέλος η απότομη πτώση του goodput οφείλεται στο γεγονός ότι όταν το *f* από 9 γίνεται 10, ο αλγόριθμος 3 εισέρχεται γρήγορα στην περίπτωση όπου το μήκος του πακέτου είναι μικρότερο ή ίσο από τα διαθέσιμα error token, για αυτό στέλνει ένα πακέτο με το εναπομείναντα φορτίο. Για αυτό έχουμε αυτά τα αποτελέσματα.

Overhead

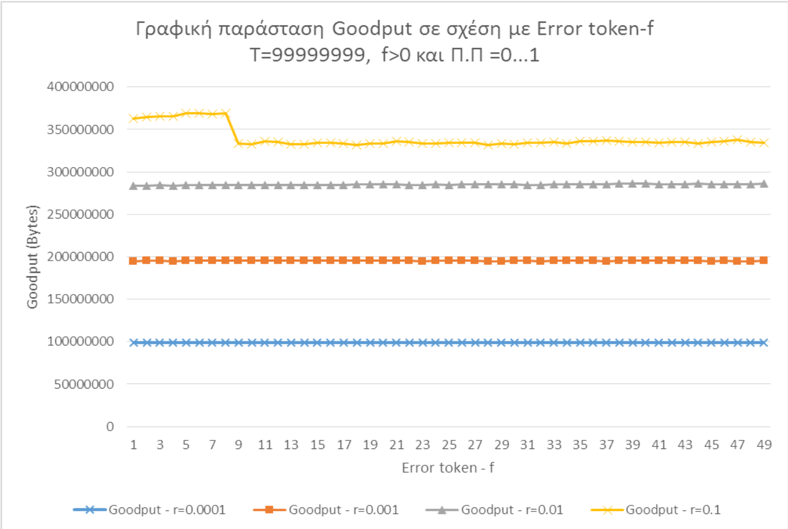
Στο Σχήματος 3.192 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά μειώνεται γρήγορα. Αυτή η απότομη μείωση οφείλεται στο γεγονός ότι αρχικά χρησιμοποιείτε ο αλγόριθμος ADP1 αφού το *f* παίρνει γρήγορα την τιμή 1 και λιγότερο ο ADP2. Ακολούθως παρατηρούμε ότι αυξάνοντας το *f* όπως αναμέναμε το overhead σταθεροποιείται. Όπως έχουμε εξηγήσει στο προηγούμενο σημείο, αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος 3 εισέρχεται γρήγορα στην περίπτωση όπου το μήκος του πακέτου είναι μικρότερο ή ίσο από τα διαθέσιμα error token, για αυτό στέλνει ένα πακέτο με το εναπομείναντα φορτίο.



Σχήμα 3.193

Σύγκριση όλων των r

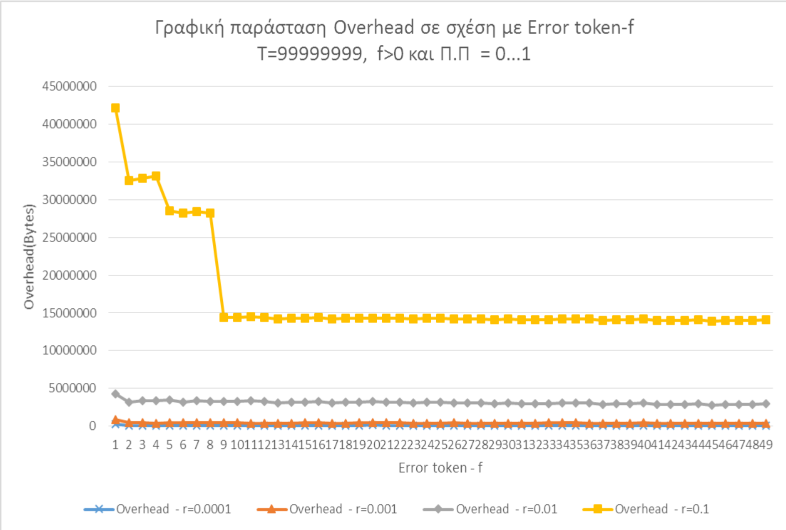
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.194

Με την βοήθεια του σχήματος 6.194 ευκολά βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια ανεξαρτήτως ρυθμού επαναφόρτισης (οι γραφικές με ρυθμός επαναφόρτισης έχουν την απότομη πτώση σε μεγαλύτερες τιμές του *f*). Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο καλύτερα αποτελέσματα έχουμε στο ωφέλιμο φορτίο. Αυτό οφείλεται στο γεγονός ότι η Π.Π δεν είναι 100% και στο ότι μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου.

Overhead



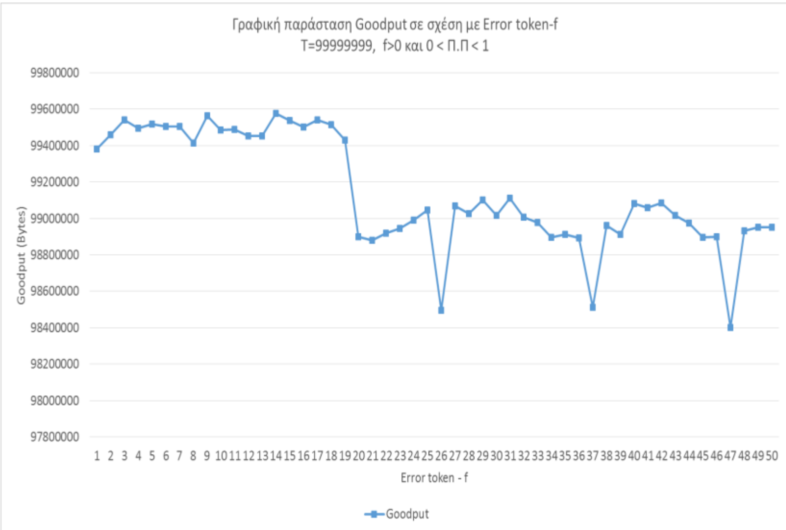
Σχήμα 3.195

Με την βοήθεια του σχήματος 3.195 ευκολά βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια ανεξαρτήτως ρυθμού επαναφόρτισης. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο μεγαλύτερο είναι το overhead. Όπως έχουμε ήδη εξηγήσει μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου => μεγαλύτερος αριθμός πακέτων για αποστολή.

Αποτελέσματα για Uniform για T = 99999999

r = 0.0001

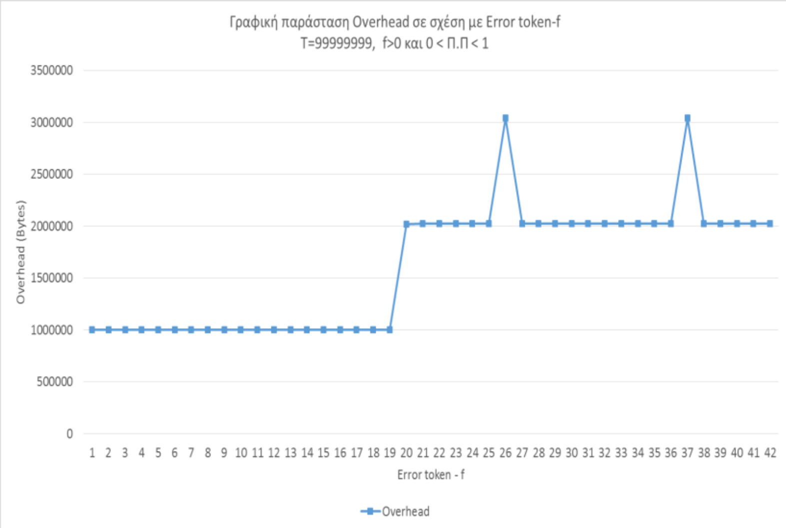
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.196

Στο Σχήματος 3.196 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται αρχικά με μικρούς ρυθμούς. Επίσης όταν το *f* από 19 γίνεται 20 παρατηρούμε μια απότομη μείωση του goodput. Σε αυτό το σημείο η νέα χρονική περίοδος ξεκινά με *f*>1, έτσι μπορεί να καταστρέψει περισσότερα πακέτα. Επίσης σε συνδυασμό με το ότι ο μέσος όρος Π.Π είναι 67%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση.

Overhead



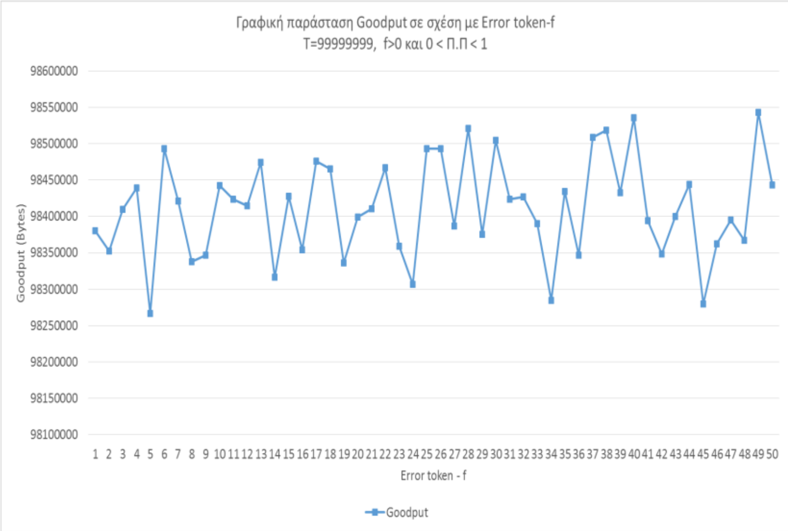
Σχήμα 3.197

Στο Σχήματος 3.197 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται γραμμικά. Στη συνέχεια παρατηρούμε μια γρήγορη άνοδο στο overhead. Ας μην ξεχνάμε ότι για τον υπολογισμό του μήκους του πακέτου λαμβάνει υπόψη το *f*. Συνεπώς αυξάνοντας το *f* κατά μία μονάδα μπορεί να μειώσει κατά πολύ το μέγεθος του πακέτου. Αυτή η συμπεριφορά συνεχίζεται για μεγάλες στιγμές του *f*.

r = 0.001

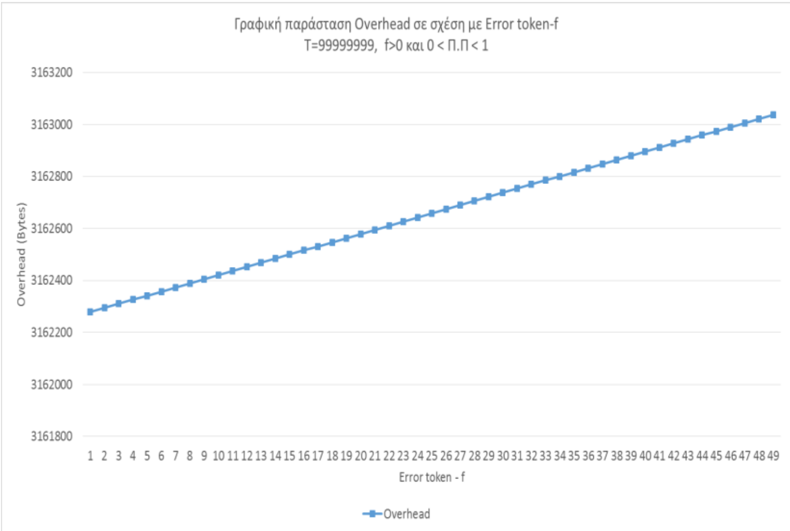
Ωφέλιμο φορτίο – Goodput

Στο Σχήματος 3.198 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 67%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση. Επίσης αυξάνοντας το *f* μειώνουμε το μήκος του πακέτου.



Σχήμα 3.198

Overhead



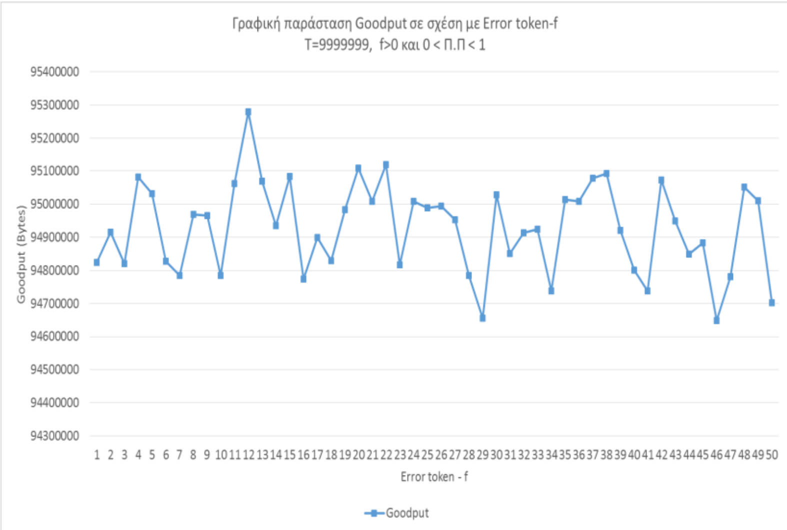
Σχήμα 3.199

Στο Σχήματος 3.199 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται με γραμμικούς ρυθμούς. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

r = 0.01

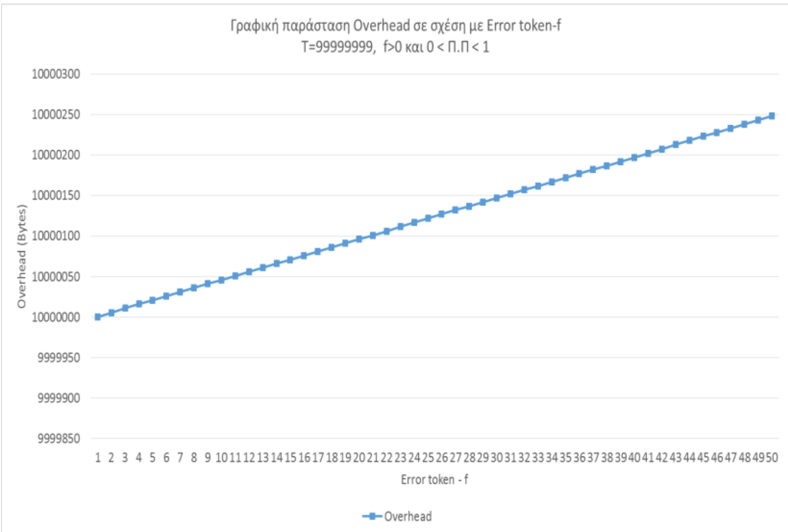
Ωφέλιμο φορτίο – Goodput

Στο Σχήματος 3.200 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται με τάση να μειώνεται με αργούς ρυθμούς. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 67%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση. Επίσης καθώς αυξάνουμε το *f* το μήκους του πακέτου μειώνεται.



Σχήμα 3.200

<u>Overhead</u>

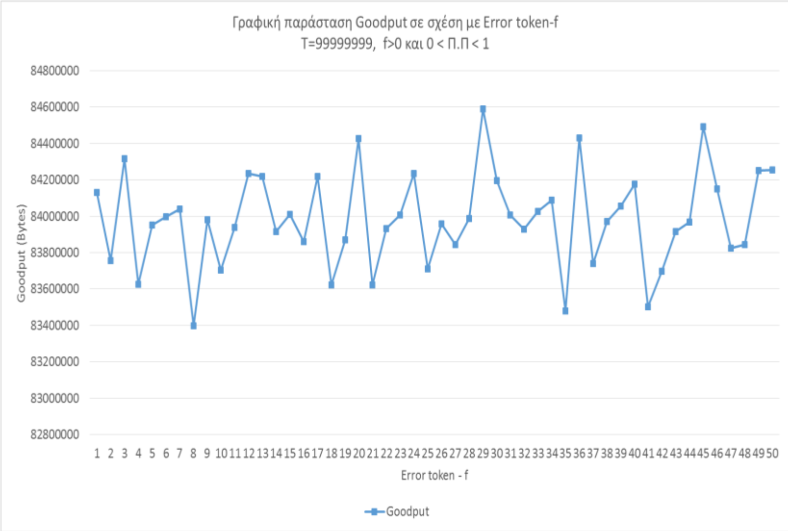


Σχήμα 3.201

Στο Σχήματος 3.201 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αυξάνεται και αυτό με γραμμικούς ρυθμούς. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

r = 0.1

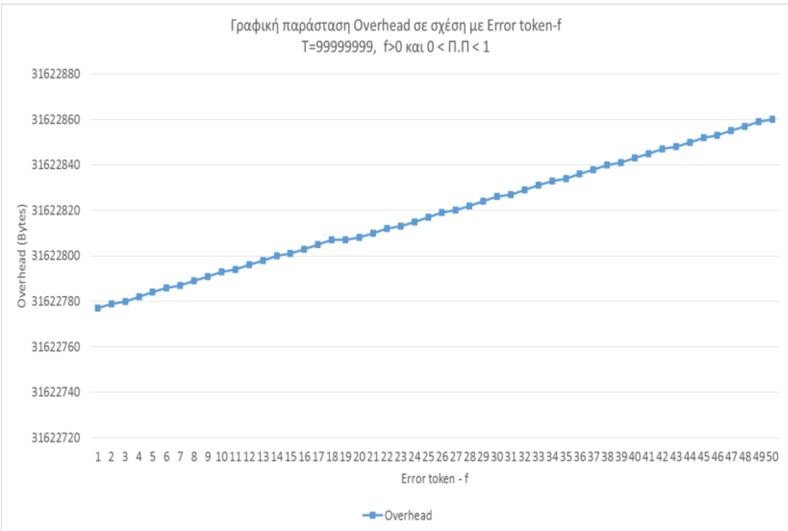
<u>Ωφέλιμο φορτίο – Goodput</u>



Σχήμα 3.202

Στο Σχήματος 3.202 παρουσιάζουμε τον όγκο των δεδομένων που παραλαμβάνονται χωρίς αλλοιώσεις / παρεμβολές από τον παραλήπτη (goodput) σε συνάρτηση με τα διαθέσιμα Error token, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Παρατηρούμε ότι καθώς αυξάνουμε το *f* το goodput αυξομειώνεται με τάση να μειώνεται με αργούς ρυθμούς. Παρόλο που στέλνουμε πακέτα ίσου μεγέθους, ο μέσος όρος Π.Π είναι 67%, συνεπώς δεν χρησιμοποιούνται όλα τα διαθέσιμα error token. Αυτός είναι ο λόγος της αυξομείωσης στην πιο πάνω γραφική παράσταση. Επίσης λόγο του πολύ μικρού ρυθμού επαναφόρτισης, στέλνονται πακέτα πολύ μικρού μεγέθους. Έτσι παρατηρούμε ότι το ωφέλιμο φορτίο αυξομειώνεται με αργούς ρυθμούς.

<u>Overhead</u>

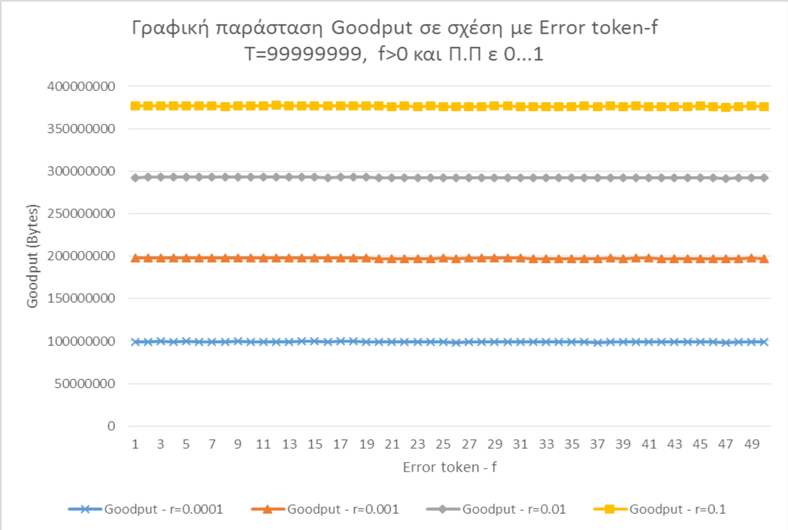


Σχήμα 3.203

Στο Σχήματος 3.203 παρουσιάζουμε τον όγκο των δεδομένων που αποτελούν το συνολικό header size (overhead) των πακέτων που στάλθηκαν κατά την προσομοίωση, για τυχαίες Πιθανότητες Παρεμβολής (Π.Π ϵ [0,1] πραγματική αριθμοί) με γενικό μέσο όρο 0,67. Εύκολα διακρίνουμε ότι καθώς αυξάνουμε γραμμικά το *f*, το overhead αρχικά αυξάνεται αυξάνεται και αυτό γραμμικά. Αυτό οφείλετε στο γεγονός ότι καθώς αυξάνουμε το *f* ο αλγόριθμος αποφασίζει να στείλει πακέτα μικρότερου μήκους.

Σύγκριση όλων των r

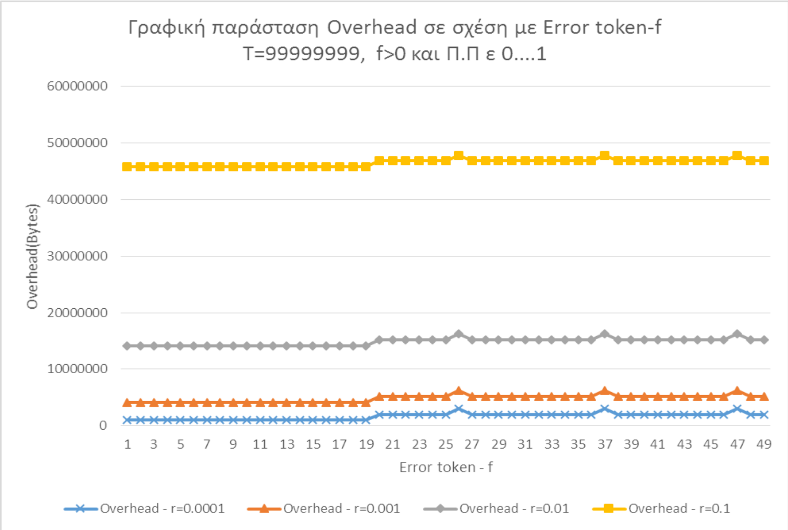
Ωφέλιμο φορτίο – Goodput



Σχήμα 3.204

Με την βοήθεια του σχήματος 3.204 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο καλύτερα αποτελέσματα έχουμε στο ωφέλιμο φορτίο. Αυτό οφείλεται στο γεγονός ότι η Π.Π δεν είναι 100% και στο ότι μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου.

Overhead

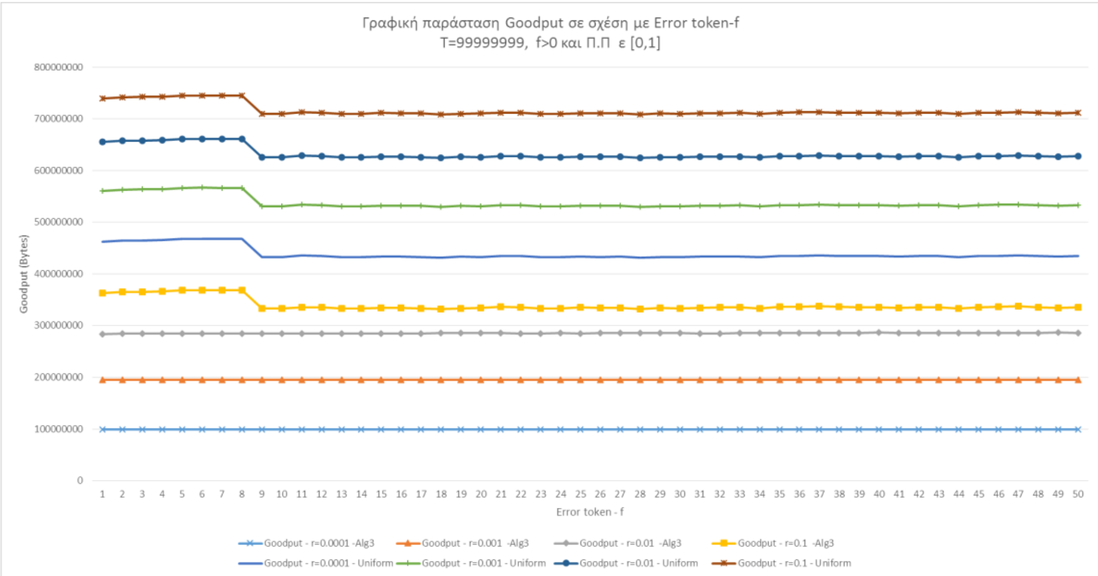


Σχήμα 3.205

Με την βοήθεια του σχήματος 3.205 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτισης η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια ανεξαρτήτως ρυθμού επαναφόρτισης. Το μόνο που διαφέρει είναι ότι όσο πιο μικρός είναι ο ρυθμός επαναφόρτισης τόσο μεγαλύτερο είναι το overhead. Όπως έχουμε ήδη εξήγηση μικρότερος ρυθμός επαναφόρτισης => πολύ μικρότερο μήκος πακέτου => μεγαλύτερος αριθμός πακέτων για αποστολή.

Αλγόριθμος ADP_CHARCHING(*T*,*f*) με *f* >1 VS Uniform – Γραφικές για T = 2048

<u>Ωφέλιμο φορτίο – Goodput</u>

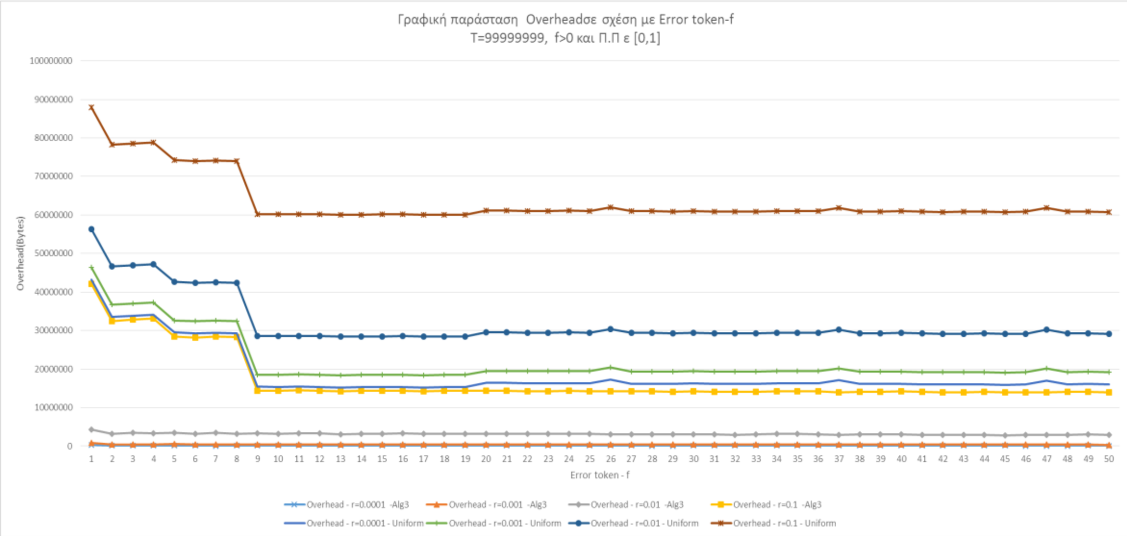


Σχήμα 3.206

Με την βοήθεια του σχήματος 3.206 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτιση η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια. Επίσης παρατηρούμε ότι όσο μικρότερος είναι ο ρυθμός επιφόρτισης τόσο καλύτερα αποτελέσματα έχουμε για το ωφέλιμο φορτίο. Επίσης φαίνεται ξεκάθαρα ότι ο αλγόριθμος uni*f*orm αποστέλλει περισσότερο ωφέλιμο φορτίο από ότι ο αλγόριθμος 3.

<u>Overhead</u>

Με την βοήθεια του σχήματος 3.207 εύκολα βλέπουμε ότι ανεξαρτήτως ρυθμού επαναφόρτιση η μορφή και συμπεριφορά των γραφικών παραστάσεων είναι η ίδια για κάθε αλγόριθμο. Επίσης παρατηρούμε ότι όσο μικρότερος είναι ο ρυθμός επιφόρτισης τόσο μικρότερο είναι το overhead για κάθε αλγόριθμο. Επίσης φαίνεται ξεκάθαρα ότι ο αλγόριθμος uni*f*orm σε σχέση με τον αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1 δημιουργεί αρκετά μεγαλύτερο overhead.


Γραφική παράσταση Overheadσε σχέση με Error token-f
T=99999999, f>0 και Π.Π ε [0,1]
Overhead(Bytes)
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Error token - f
Overhead - r=0.0001 -Alg3
Overhead - r=0.001 -Alg3
Overhead - r=0.01 -Alg3
Overhead - r=0.1 -Alg3
Overhead - r=0.0001 - Uniform
Overhead - r=0.001 - Uniform
Overhead - r=0.01 - Uniform
Overhead - r=0.1 - Uniform

Σχήμα 3.207

Κεφάλαιο 4

Συμπεράσματα και Μελλοντική Εργασία

4.1 Συμπεράσματα

Στην παρούσα διπλωματική εργασία υλοποιήθηκαν και αξιολογήθηκαν διάφοροι αλγόριθμοι χρονοπρογραμματισμού πακέτων στην παρουσία σφαλμάτων. Κάποιοι από αυτούς αποτελούσαν την βέλτιστή λύση για το πρόβλημα που αναπτύχθηκαν όπως ο αλγόριθμος ADP(*T, 1*) και ο αλγόριθμος Uniform. Οι αλγόριθμοι ADP(*T, f*) με $f > 1$ και ADP_CHARCHING(*T, f*) με $f \geq 1$, δεν αποτελούν την βέλτιστη λύση για το πρόβλημα. Παρόλα αυτά στην παρούσα εργασία είχαμε ως κύριο στόχο την σύγκριση του αλγορίθμου Uni*f*orm (που αποτελεί την βέλτιστη λύση για αποστολή πακέτων ιδίου μεγέθους) με τον αλγόριθμο ADP_CHARCHING(*T, f*) με $f \geq 1$. Η αξιολόγηση των αλγορίθμων έγινε με τη χρήση του προσομοιωτή δικτύων Network Simulation 3.

Λαμβάνοντας υπόψη τα αποτελέσματα από την πειραματική αξιολόγηση των αλγορίθμων εξαγάγαμε τα συμπεράσματα που περιγράφονται στις παραγράφους που ακολουθούν.

Με μια πρώτη ματιά φαίνεται ότι ο αλγόριθμος Uniform υπερτερεί σε σχέση με τον αλγόριθμο ADP_CHARCHING(*T, f*) με $f \geq 1$. Πιο συγκεκριμένα ο Uniform στέλνει πολύ μεγαλύτερο ωφέλιμο φορτίο από τον αλγόριθμο ADP_CHARCHING(*T, f*) με $f \geq 1$. Όπως μπορούμε να δούμε από τις γραφικές παραστάσεις του προηγούμενου κεφαλαίου, αυτό ισχύει για όλους τους ρυθμούς επαναφόρτισης και για όλα τα *f*. Έτσι μπορούμε να πούμε ότι ισχύει ανεξαρτήτως ρυθμού επαναφόρτισης ή μέγιστου αριθμού *error token*.

Πριν μιλήσουμε για το συνολικό επιπλέων φορτίο, θα ήταν καλό να θυμηθούμε ότι το μέγεθος της κεφαλίδας του πακέτου για κάθε πακέτο ορίσθηκε να είναι 1 byte. Συνεπώς μέσω των γραφικών για όλο overhead μπορούμε να δούμε και τον ακριβή αριθμό των πακέτων που αποστέλλονται κατά την προσομοίωση. Μετά από αυτό εύκολα αντιλαμβανόμαστε ότι ο αλγόριθμος Uniform στέλνει πάρα πολλά πακέτα συγκριτικά με τον αλγόριθμο ADP_CHARCHING(*T*,*f*) με *f* >1. Όμως για το χρήστη αυτή η επιπλέον πληροφορία είναι άχρηστη. Δηλαδή θεωρείται ως μη ωφέλιμο φορτίο. Παρόλα αυτά είναι αναγκαία ώστε να μπορεί να παραληφθεί και να αποκωδικοποιηθεί το πακέτο από τον παραλήπτη.

Έτσι μπορούμε να πούμε ότι το εξαιρετικά μεγάλο επιπλέων φορτίο του αλγορίθμου Uniform (για μεγάλες τιμές του *f* το overhead ξεπερνά το συνολικό φορτίο που θέλουμε να στείλουμε) έρχεται να μετριάσει το χάσμα του ωφέλιμου φορτίου ανάμεσα στους δύο αλγορίθμους.

Επίσης ας μην ξεχνάμε ότι οι πόροι των ασύρματων δικτύων είναι ακριβοί και περιορισμένοι, συνεπώς για την καλύτερη και πιο αποδοτική χρήση τους πρέπει να υπάρχει μια ισορροπία ανάμεσα στο overhead και το ωφέλιμο φορτίο.

Συνεπώς μέσω της παρούσας διπλωματικής εργασία μπορέσαμε να επιβεβαιώσουμε την θεωρητική ανάλυση των αλγορίθμων. Το πιο σημαντικό όμως και ίσως το πιο ενδιαφέρον συμπέρασμα που εξαγάγαμε από την παρούσα εργασία αφορά τον υπολογισμό του μήκους του πακέτου σε σχέση με το ρυθμό επαναφόρτισης. Πιο συγκεκριμένα παρατηρήσαμε ότι όταν ο ρυθμός επαναφόρτισης είναι σχετικά μικρός σε σχέση με το συνολικό φορτίο έχουμε καλύτερα αποτελέσματα σε σχέση με το ωφέλιμο φορτίο. Έτσι όταν γνωρίζουμε ότι ο ρυθμός επαναφόρτισης της συσκευής είναι σχετικά μεγάλος σε σχέση με το συνολικό φορτίο που θέλουμε να στείλουμε, θα ήταν καλό να μειώνουμε το μήκος του πακέτου με κάποιο τρόπο, χωρίς όμως να υπερφορτώνουμε το ασύρματο δίκτυο με πολύ επιπλέων φορτίο.

Τέλος, παρόλο που η λύση η οποία προτείναμε για τον αλγόριθμο ADP_CHARCHING(*T*, *f*) με *f* >=1, επιβεβαιώσαμε ότι ο αλγόριθμος μεγιστοποιεί σε ικανοποιητικό επίπεδο το ωφέλιμο φορτίο, χωρίς να επιβαρύνει σε μεγάλο βαθμό το δίκτυο με επιπλέων φορτίο. Έτσι αυτή η εργασία μπορεί να παροτρύνει τους ερευνητές να συνεχίσουν την εργασία τους μέχρι να βρουν την βέλτιστη λύση για την εκδοχή του προβλήματος που μελετούν [2, 3].

4.2 Μελλοντική Εργασία

Στη παρούσα διπλωματική εργασία έγινε πειραματική αξιολόγηση των αλγορίθμων που περιγράφονται κάτω από συγκεκριμένες συνθήκες. Για να γίνουμε πιο συγκεκριμένη στην παρούσα εργασία απαραίτητη προϋπόθεση είναι να γνωρίζουμε ακριβώς ή κατά προσέγγιση τον ρυθμό επαναφόρτισης της κακόβουλης συσκευής. Επίσης να γνωρίζουμε το μέγιστο αριθμό *error token* που διαθέτει η κακόβουλη συσκευή από την αρχή. Αυτό πολλές φορές δεν είναι δυνατό. Αυτό που είναι δυνατό είναι να τα μάθουμε σε προσεγγιστικό βαθμό, με βάση την ανατροφοδότηση.

Έτσι σε μελλοντικό στάδιο θα ήταν καλό να αναπτυχθούν οι αλγόριθμοι με τέτοιο τρόπο, ώστε να μπορούν να μάθουν κατά προσέγγιση το ρυθμό επαναφόρτισης και το μέγιστο αριθμό *error token* με βάση την ανατροφοδότηση. Αυτό φυσικά θα μπορεί να γίνει μέσω της ανατροφοδότηση μετά από κάποιο χρονικό διάστημα. Φυσικά ο ρυθμός επαναφόρτισης μπορεί να υπολογιστή και με τη βοήθεια αισθητήρα που να παρακολουθεί τον καιρό, σε περίπτωση που η κακόβουλη συσκευή φορτίζει μέσω της ηλιακής ενέργεια αν γνωρίζουμε τα χαρακτηριστικά του φωτοβολταϊκού πάνελ.

Φυσικά σε πρώτο στάδιο θα ήταν καλό να βρεθεί μια βέλτιστη λύση για τον υπολογισμό του μήκους του πακέτου για τον αλγόριθμο ADP_CHARCHING(T,f) με $f > 1$. Όταν επιτευχθεί αυτό, θα μπορούσε θα ήταν καλό να ελεχθεί ξανά ο αλγόριθμος πειραματικά, ώστε να μπορέσουμε να δούμε την πραγματικό όφελος τους αλγορίθμου. Ας μην ξεχνάμε ότι σε πρώτο στάδιο τα αποτελέσματα του αλγορίθμου είναι θετικά.

Φυσικά σε μελλοντική εργασία θα ήταν καλό οι αλγόριθμοι να υλοποιηθούν σε πιο χαμηλό επίπεδο, δηλαδή μέσα στα πρωτόκολλα επικοινωνίας. Με άλλα λόγια να ενσωματωθούν οι αλγόριθμοι μέσα στα παρών πρωτόκολλα, έτσι ώστε να μπορούν να βελτιστοποιήσουν το ωφέλιμο φορτίο κάτω από τις συνθήκες που αναλύσαμε στην παρούσα εργασία. Ας μην ξεχνάμε ότι όταν υλοποιήθηκαν τα πρωτόκολλα επικοινωνίας δεν είχε αναπτυχθεί αυτή η τεχνολογία (κακόβουλες συσκευές που παρεμβάλουν στο δίκτυο) ώστε να ληφθεί υπόψη.

Βιβλιογραφία

[1] Andrews Matthew, Baruch Awerbuch, Antonio Fernandez, Tom Leighton, Zhiyong Liu, and Jon Kleinberg. Universal-stability results and performance bounds for greedy contention-resolution protocols. *Journal of the ACM (JACM),* 48(1):39–69, 2001.

[2] Antonio Fernandez Anta, Chryssis Georgiou, and Elli Zavou, Adaptive Scheduling over a Wireless Channel under Constrained Jamming, *Proc. of the 9th International Conference on Combinatorial Optimization and Applications (COCOA 2015)*, pp. 261-278, Houston, TX, USA, 2015.

[3] Antonio Fernandez Anta, Chryssis Georgiou, and Elli Zavou, Packet Scheduling over a Wireless Channel: AQT-based Constrained Jamming, *Proc. of the 3rd International Conference on Networked systems (NETYS 2015)*, pp. 230-245, Agadir, Morocco, 2015.

[4] Bhagwat Pravin, Partha Bhattacharya, Arvind Krishna, and Satish K Tripathi. Enhancing throughput over wireless lans using channel state dependent packet scheduling. In *INFO-COM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 3, pages 1133–1140. IEEE, 1996.

[5] Bogdan S Chlebus, Dariusz R Kowalski, and Mariusz A Rokicki. Adversarial queuing on the multiple-access channel. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 92–101. ACM, 2006.

[6] Christophoros Christophorou – EPL476: Mobile Computer Networks [Online] http://www.cs.ucy.ac.cy/courses/EPL476/lectures.html

[7] Fu Zhenghua, Petros Zerfos, Haiyun Luo, Songwu Lu, Lixia Zhang, and Mario Gerla. The impact of multihop wireless channel on tcp throughput and loss. *In INFOCOM 2003. Twentysecond annual joint conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1744–1753. IEEE, 2003

[8] Konstantinos Pelechrinis, Marios Iliofotou, and Srikanth V Krishnamurthy. Denial of service attacks in wireless networks: The case of jammers. *Communications Surveys & Tutorials, IEEE*, 13(2):245–257, 2011.

[9] Network Simulation 3 – Manual [Online]

https://www.nsnam.org/docs/manual/html/

[10] Network Simulation 3 – Tutorial [Online] https://www.nsnam.org/docs/tutorial/html/

[11] A. Richa, C. Scheideler, S. Schmid, and Jin Zhang. Competitive and fair medium access despite reactive jamming. In *Proc. of ICDCS*, pages 507–516, 2011.

[12] Richa Andrea, Christian Scheideler, Stefan Schmid, and Jin Zhang. Towards jammingresistant and competitive medium access in the sinr model. In *Proceedings of the 3rd ACM workshop on Wireless of the students, by the students*, for the students, pages 33–36. ACM, 2011.

[13] Richa Andrea, Christian Scheideler, Stefan Schmid, and Jin Zhang. Competitive and fair throughput for co-existing networks under adversarial interference. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 291–300. ACM, 2012.

[14] Uysal-Biyikoglu Elif, Balaji Prabhakar, and Abbas El Gamal. Energy-efficient packet transmission over a wireless link. IEEE/ACM Trans. Netw., 10(4):487–499, August 2002.

[15] Vagelis Tsibonis, Leonidas Georgiadis, and Leandros Tassiulas. Exploiting wireless channel state information for throughput maximization. *Information Theory, IEEE Transactions on*, 50(11):2566–2582, 2004.

[16] http://www.defensetech.org/2012/03/05/the-democratization-of-electronic-warfare/

[17] http://www.amfastech.com/2013/03/a-project-on-packet-hiding-methods-for.html

[18] http://www.jammer4uk.com/

[19] http://www.jammer-store.com/wifi-bluetooth-jammers-blockers.html

[20] http://www.cell-jammers.com/

Παράρτημα Α

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <cstring>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/netanim-module.h"

using namespace ns3;

std::fstream filestr;

int find_i(int T);
int find_a(int i);
int find_b(int i);
int find_packet_size(int T, int a, int b);
void ADP(void);


class MyApp : public Application
{
public:

  MyApp ();

  void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize, DataRate dataRate, int T, int f,double totalTime, int
useFull, int dropped, int headerSize, int count,float P,int epa);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);
  void ADP(void);


  Ptr<Socket>     m_socket;
  Address         m_peer;
  uint32_t        m_packetSize;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  bool            m_running;
  uint32_t        m_packetsSent;
  int             m_T;
  int             m_f;
  double          m_totalTime;
  int             m_useFull;
  int             m_dropped;
  int             m_headerSize;
  int             m_totalOverhead;
  int m_count;
  float m_P;
  int m_epa;
};

//constructor
MyApp::MyApp ()
  : m_socket (0),
    m_peer (),
    m_packetSize (0),
    m_dataRate (0),
    m_sendEvent (),
    m_running (false),
    m_packetsSent (0),
```

```
    m_T (),
    m_f (),
    m_totalTime(),
    m_useFull(),
    m_dropped(),
    m_headerSize(),
    m_totalOverhead(),
    m_count(),
    m_P(),
    m_epa()
{
}

void
MyApp::Setup (Ptr<Socket> socket, Address address, uint32_t packetSize, DataRate dataRate, int T, int f,double totalTime,int useFull, int dropped, int headerSize, int count,float P,int epa)
{
  m_socket = socket;
  m_peer = address;
  m_packetSize = 0;
  m_dataRate = dataRate;
  m_T =T;
  m_f =f;
  m_totalTime = 0.0;
  m_useFull=0;
  m_dropped=0;
  m_headerSize = headerSize;
  m_totalOverhead = 0;
  m_count = count;
  m_P = P;
  m_epa = epa;

}

void
MyApp::StartApplication (void)
{
  while(m_T!=0){
    m_running = true;
    m_packetsSent = 0;
    m_socket->Bind ();
    m_socket->Connect (m_peer);
    ADP();
  }

  filestr<<m_useFull<<"\t"<<m_dropped<<"\t"<<m_totalOverhead<<"\n";

  if (m_count == m_epa)
    filestr.close();


}

void
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}


void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize);
```

```
  m_socket->Send (packet);
  if (m_T !=0)
    ScheduleTx ();
}


void
MyApp::ADP(void){
  double jammed = (rand()%100)/100.00;
  if (m_T==0)
    return;
  if (m_T>=1 && m_T<2){
    m_packetSize = m_T + m_headerSize;
    if (m_f==1 && jammed<m_P){
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
      m_T = m_T - m_packetSize + m_headerSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      SendPacket();
    }else{
      m_useFull = m_useFull + m_packetSize-m_headerSize;
      m_T = m_T-m_packetSize + m_headerSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      SendPacket();
    }
  }else{
    int i = find_i(m_T);
    int a = find_a(i);
    int b = find_b(i);
    m_packetSize = find_packet_size(m_T, a, b) + m_headerSize;
    if (m_f==1 && jammed<m_P){
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
      m_T = m_T - m_packetSize + m_headerSize ;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      SendPacket();
      m_packetSize = m_T + m_headerSize ;
      m_f=m_f-1;
    }
    m_useFull = m_useFull + m_packetSize - m_headerSize;
    m_T = m_T - m_packetSize + m_headerSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    SendPacket();
    m_packetSize=0;
  }
}

void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
      m_totalTime = m_totalTime + (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ()));
    }
}

int main (int argc, char *argv[]){

  int T= std::atoi(argv[1]);
  int f = std::atoi(argv[2]);
  int headerSize = std::atoi(argv[3]);
  float P = std::atof(argv[4]);
  int epa = std::atoi(argv[5]);
  filestr.open ("Algorithm1.txt", std::fstream::in | std::fstream::out | std::fstream::app);
  int count = 0;
  double startTime = 0.0;
  double stopTime = 10.0;
  srand ( time(NULL) );
  NodeContainer wifiStaNodes;
  wifiStaNodes.Create (2);

  YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
  YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
  phy.SetChannel (channel.Create ());
```

```
  WifiHelper wifi = WifiHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::AarfWifiManager");
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0));

  NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();

  NetDeviceContainer staDevices;
  staDevices = wifi.Install (phy, mac, wifiStaNodes);

  MobilityHelper mobility;
  mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                 "MinX", DoubleValue (0.0),
                                 "MinY", DoubleValue (0.0),
                                 "DeltaX", DoubleValue (50.0),
                                 "DeltaY", DoubleValue (20.0),
                                 "GridWidth", UintegerValue (3),
                                 "LayoutType", StringValue ("RowFirst"));

  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (wifiStaNodes);

  InternetStackHelper stack;
  stack.Install (wifiStaNodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.5.0", "255.255.255.252");
  Ipv4InterfaceContainer interfaces = address.Assign (staDevices);

  uint16_t sinkPort = 8080;
  Address sinkAddress (InetSocketAddress (interfaces.GetAddress (0), sinkPort));
  PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
  ApplicationContainer sinkApps = packetSinkHelper.Install (wifiStaNodes.Get (0));
  sinkApps.Start (Seconds (startTime+0.5));
  sinkApps.Stop (Seconds (stopTime));

  Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (wifiStaNodes.Get (1), TcpSocketFactory::GetTypeId ());

  Ptr<MyApp> app = CreateObject<MyApp> ();
   do{

  app->Setup (ns3TcpSocket, sinkAddress, 1040, DataRate ("1Bps"),T,f,0,0,0,headerSize,count,P,epa);
  wifiStaNodes.Get (0)->AddApplication (app);
  app->SetStartTime (Seconds (startTime+0.5));
  app->SetStopTime (Seconds (stopTime));

  }while(count!=epa);

  Simulator::Stop (Seconds (stopTime));
  AnimationInterface anim ("lab4.xml");
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}


int find_i(int T){
  unsigned int i;
  unsigned int low;
  unsigned int high;
  i = 1;
  do{
    i++;
    low  = (((i-1)*i)/2)+1;
    high  = (((i+1)*i)/2)+1;
    if((unsigned)T >= low && (unsigned)T<high){
      return i;
    }
  }while(true);
}

int find_a(int i){
  return (i-2);
```

```
}

int find_b(int i){
  return ((((i-1)*i)/2)-1);
}
int find_packet_size(int T, int a, int b){
  return ((T+b)/(a+2));
}
```

Παράρτημα B

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <cstring>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/netanim-module.h"

using namespace ns3;

std::fstream Results;
std::fstream Alg_work;

float find_P();
int find_i(int T);
int find_a(int i);
int find_b(int i);
int find_packet_size(int T, int a, int b);
int find_packet_size2(int T, int a, int b, int c);
void ADP1(void);
void ADP2(void);


class MyApp : public Application
{
public:
  MyApp ();

  void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize, DataRate dataRate, int T, int f,double totalTime, int useFull, int dropped, int headerSize, int count,float P,int epa);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);
  void ADP1(void);
  void ADP2(void);

  Ptr<Socket>     m_socket;
  Address         m_peer;
  uint32_t        m_packetSize;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  bool            m_running;
  uint32_t        m_packetsSent;
  int             m_T;
  int             m_f;
  double          m_totalTime;
  int             m_useFull;
  int             m_dropped;
  int             m_headerSize;
  int             m_totalOverhead;
  int             m_count;
  float           m_P;
  int             m_epa;
  int             m_countP;
  float           m_AveP;
};

//constructor
MyApp::MyApp ()
  : m_socket (0),
```

```
    m_peer (),
    m_packetSize (0),
    m_dataRate (0),
    m_sendEvent (),
    m_running (false),
    m_packetsSent (0),
    m_T (),
    m_f (),
    m_totalTime(),
    m_useFull(),
    m_dropped(),
    m_headerSize(),
    m_totalOverhead(),
    m_count(),
    m_P(),
    m_epa(),
    m_countP(),
    m_AveP()
{
}

void
MyApp::Setup (Ptr<Socket> socket, Address address, uint32_t packetSize, DataRate dataRate, int T, int f,double totalTime,int useFull, int dropped, int headerSize, int count,float P,int epa)
{
  m_socket = socket;
  m_peer = address;
  m_packetSize = 0;
  m_dataRate = dataRate;
  m_T =T;
  m_f =f;
  m_totalTime = 0.0;
  m_useFull=0;
  m_dropped=0;
  m_headerSize = headerSize;
  m_totalOverhead = 0;
  m_count = count;
  m_P = P;
  m_epa = epa;
  m_countP=0;
  m_AveP=0.0;

}

void
MyApp::StartApplication (void)
{
  while(m_T!=0){
    m_running = true;
    m_packetsSent = 0;
    m_socket->Bind ();
    m_socket->Connect (m_peer);
    if (m_f<=1){
      Alg_work<<m_f<<"\t"<<"ADP1"<<"\n";
      ADP1();
    }
    else {
      Alg_work<<m_f<<"\t"<<"ADP2"<< "\n";
      ADP2();
    }
  }
  if (m_AveP == 0)
    Results<<m_useFull<<"\t"<<m_dropped<<"\t"<<m_totalOverhead<<"\t"<<"0.0"<<"\n";
  else
    Results<<m_useFull<<"\t"<<m_dropped<<"\t"<<m_totalOverhead<<"\t"<<(float)(m_AveP/m_countP)<<"\n";

  if (m_count == m_epa){
   Results.close();
   Alg_work.close();
  }

}

void
```

```
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}


void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize);
  m_socket->Send (packet);
  if (m_T !=0)
    ScheduleTx ();
}

void
MyApp::ADP1(void){
  double jammed = (rand()%100)/100.00;
  if (m_T==0)
    return;
  if (m_T>=1 && m_T<2){
    m_packetSize = m_T + m_headerSize;
    m_P = find_P();
    if (m_f==1 && jammed<m_P){
      m_countP++;
      m_AveP = m_AveP+m_P;
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
      m_T = m_T - m_packetSize + m_headerSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      SendPacket();
    }else{
      m_useFull = m_useFull + m_packetSize-m_headerSize;
      m_T = m_T-m_packetSize + m_headerSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      SendPacket();
    }
  }else{
    int i = find_i(m_T);
    int a = find_a(i);
    int b = find_b(i);
    m_packetSize = find_packet_size(m_T, a, b) + m_headerSize;
    m_P = find_P();
    if (m_f==1 && jammed<m_P){
      m_countP++;
      m_AveP = m_AveP+m_P;
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
      m_T = m_T - m_packetSize + m_headerSize ;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      SendPacket();
      m_packetSize = m_T + m_headerSize ;
      m_f=m_f-1;
    }
    m_useFull = m_useFull + m_packetSize - m_headerSize;
    m_T = m_T - m_packetSize + m_headerSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    SendPacket();
    m_packetSize=0;
  }
}


void
MyApp::ADP2(void){
  int a = 2;
```

```
  int b = 16;
  int c = 8;
  double jammed = (rand()%100)/100.00;
  if(m_T <= (m_f+1)){
    m_packetSize = m_T;
    m_T = m_T - m_packetSize;
    m_packetSize = m_packetSize + m_headerSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    SendPacket();
    m_P = find_P();
    if (jammed<m_P && m_f>1){
      m_countP++;
      m_AveP = m_AveP+m_P;
      m_f = m_f -1;//packet jammed
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
    }else{
      m_useFull = m_useFull + m_packetSize - m_headerSize ;
    }
  }else{
    m_packetSize = find_packet_size2(m_T,a,b,c);
    m_T = m_T - m_packetSize;
    m_packetSize = m_packetSize + m_headerSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    SendPacket();
    m_P = find_P();
    if (jammed<find_P() && m_f>1){
      m_countP++;
      m_AveP = m_AveP+m_P;
      m_f = m_f -1;//packet jammed
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
    }else{
      m_useFull = m_useFull + m_packetSize - m_headerSize; ;
    }
  }
}

void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
      m_totalTime = m_totalTime + (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ()));
    }
}

int main (int argc, char *argv[]){
  int T= std::atoi(argv[1]);
  int f = std::atoi(argv[2]);
  int headerSize =  std::atoi(argv[3]);
  float P = std::atof(argv[4]);
  int epa = std::atoi(argv[5]);

  Results.open ("anwser/Alg2_Result.txt", std::fstream::in | std::fstream::out | std::fstream::app);
  Alg_work.open("anwser/Alg2_work.txt",std::fstream::in | std::fstream::out | std::fstream::app);
  Alg_work<<"_________________________________________________________________"<<"\n";

  int count = 0;
  double startTime = 0.0;
  double stopTime = 10.0;
  std::cout<<"argc= "<<argc<<"\n";
  srand ( time(NULL) );
  NodeContainer wifiStaNodes;
  wifiStaNodes.Create (2);

  YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
  YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
  phy.SetChannel (channel.Create ());

  WifiHelper wifi = WifiHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::AarfWifiManager");
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0));
```

```
  NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();

  NetDeviceContainer staDevices;
  staDevices = wifi.Install (phy, mac, wifiStaNodes);

  MobilityHelper mobility;
  mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                 "MinX", DoubleValue (0.0),
                                 "MinY", DoubleValue (0.0),
                                 "DeltaX", DoubleValue (50.0),
                                 "DeltaY", DoubleValue (20.0),
                                 "GridWidth", UintegerValue (3),
                                 "LayoutType", StringValue ("RowFirst"));

  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (wifiStaNodes);

  InternetStackHelper stack;
  stack.Install (wifiStaNodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.5.0", "255.255.255.252");
  Ipv4InterfaceContainer interfaces = address.Assign (staDevices);

  uint16_t sinkPort = 8080;
  Address sinkAddress (InetSocketAddress (interfaces.GetAddress (0), sinkPort));
  PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
  ApplicationContainer sinkApps = packetSinkHelper.Install (wifiStaNodes.Get (0));
  sinkApps.Start (Seconds (startTime));
  sinkApps.Stop (Seconds (stopTime));

  Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (wifiStaNodes.Get (1), TcpSocketFactory::GetTypeId ());

   do{
  Ptr<MyApp> app = CreateObject<MyApp> ();
  app->Setup (ns3TcpSocket, sinkAddress, 1040, DataRate ("1Bps"),T,f,0,0,0,headerSize,count,P,epa);
  wifiStaNodes.Get (0)->AddApplication (app);
  app->SetStartTime (Seconds (startTime));
  app->SetStopTime (Seconds (stopTime));
  count++;
  }while(count!=epa);



  Simulator::Stop (Seconds (stopTime));
  //AnimationInterface anim ("alg02.xml");
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}


int find_i(int T){
  int i;
  unsigned int low;
  unsigned int high;
  i = 1;
  do{
    i++;
    low  = (((i-1)*i)/2)+1;
    high  = (((i+1)*i)/2)+1;
    if((unsigned)T >= low && (unsigned)T<high){
      return i;
    }
  }while(true);
}

int find_a(int i){
  return (i-2);
}

int find_b( int i){
  return ((((unsigned)(i-1)*i)/2)-1);
```

```
}

int find_packet_size(int T, int a, int b){
  return (((unsigned)T+(unsigned)b)/((unsigned)a+2));
}

int find_packet_size2(int T, int a, int b, int c){
  unsigned int  packetSize;
  do{
    do{
      packetSize = ((a*T+ b)/c);
      a++;
      b++;
      c++;
    }while(packetSize<=0);
  }while (packetSize>(unsigned)T);

  return packetSize;
}

float find_P(){
  srand((unsigned)time(NULL));
  float par=0.0;
  // Float between 0 and 1
  par = (float)rand() / RAND_MAX;
  return par;
}
```

Παράρτημα Γ

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <cstring>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/netanim-module.h"

using namespace ns3;

std::fstream Results;
std::fstream Alg_work;

int find_i(int T);
int find_a(int i);
int find_b(int i);
int find_packet_size(int T, int a, int b);
int find_packet_size2(int T, int a, int b, int c);
void ADP1(void);
void ADP2(void);


class MyApp : public Application
{
public:
  MyApp ();

  void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize, DataRate dataRate, int T, int f,double totalTime, int useFull, int dropped, int headerSize, int count,float P,int epa, double r);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);
  void ADP1(void);
  void ADP2(void);

  Ptr<Socket>     m_socket;
  Address         m_peer;
  uint32_t        m_packetSize;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  bool            m_running;
  uint32_t        m_packetsSent;
  int             m_T;
  int             m_f;
  double          m_totalTime;
  int             m_useFull;
  int             m_dropped;
  int             m_headerSize;
  int             m_totalOverhead;
  int             m_count;
  float           m_P;
  int             m_epa;
  double                  m_nextCharging;
  double                  m_r;
  double          m_Period;
  double                  m_Remainder;
  int                             m_fMax;
  double                  m_time;
  int                             m_header;
  double                  m_old;
```

```
  int                              m_flag;

};

//constructor
MyApp::MyApp ()
  : m_socket (0),
    m_peer (),
    m_packetSize (0),
    m_dataRate (0),
    m_sendEvent (),
    m_running (false),
    m_packetsSent (0),
    m_T (),
    m_f (),
    m_totalTime(),
    m_useFull(),
    m_dropped(),
    m_headerSize(),
    m_totalOverhead(),
    m_count(),
    m_P(),
    m_epa(),
    m_nextCharging(),
    m_r(),
    m_Period(),
    m_Remainder(),
    m_fMax(),
    m_time(),
    m_header(),
    m_old(),
    m_flag()
{
}

void
MyApp::Setup (Ptr<Socket> socket, Address address, uint32_t packetSize, DataRate dataRate, int T, int f,double totalTime,int useFull, int dropped, int headerSize, int count,float P,int epa,double r)
{
  m_socket = socket;
  m_peer = address;
  m_packetSize = 0;
  m_dataRate = dataRate;
  m_T =T;
  m_f =f;
  m_totalTime = 0.0;
  m_useFull=0;
  m_dropped=0;
  m_headerSize = headerSize;
  m_totalOverhead = 0;
  m_count = count;
  m_P = P;
  m_epa = epa;
  m_nextCharging=1/r;
  m_r = r;
  m_Period = 1/r;
  m_Remainder = m_T;
  m_fMax = f;
  m_time =0.0;
  m_header=0;
  m_old=0.0;
  m_flag = 1;
}

void
MyApp::StartApplication (void)
{
    m_running = true;
    m_packetsSent = 0;
    m_socket->Bind ();
    m_socket->Connect (m_peer);
    do{
      m_old =(m_useFull+m_dropped);
      if (m_Remainder>(1/m_r))
```

```
            m_Period = 1/m_r;
      else{
            m_Period =m_Remainder;
      }

      if (m_Period <1/m_r)
            m_flag = 0;
      else
            m_flag = 1;

      m_Remainder = m_Remainder - m_Period;

      m_time = m_time + m_Period;
      while(m_Period >0){

            if (m_f<=1){
              ADP1();
            }else{
              ADP2();
            }
      }

      if (m_time>=m_nextCharging){
            m_nextCharging = m_nextCharging+1/m_r;
            if(m_fMax!=m_f){
              m_f++;
            }
      }

      if (m_old == m_useFull+m_dropped)
            break;
    }while(m_flag == 1);
    if (m_Remainder>0){
      m_packetSize = m_Remainder + m_headerSize;
      SendPacket ();
      double jammed = (rand()%100)/100.00;
      if (m_f!=0 && jammed<m_P)
            m_dropped = m_dropped+  m_packetSize- m_headerSize;
      else
            m_useFull = m_useFull +  m_packetSize - m_headerSize;
      m_totalOverhead = m_totalOverhead + m_headerSize;

    }

    Results<<m_useFull<<"\t"<<m_dropped<<"\t"<<m_totalOverhead<<"\n";

 if (m_count == m_epa){
   Results.close();
  }
}

void
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}


void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize);
  m_socket->Send (packet);
  if (m_T !=0)
```

```
    ScheduleTx ();
}

void
MyApp::ADP1(void){
  double jammed = (rand()%100)/100.00;
  if (m_Period==0)
    return;
  if (m_Period>=1 && m_Period<2){
    m_packetSize = m_Period;
    if (m_f==1 && jammed<m_P){
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
      m_Period = m_Period - m_packetSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      m_Remainder=m_Remainder+m_headerSize;
      SendPacket();
    }else{
      m_useFull = m_useFull + m_packetSize-m_headerSize;
      m_Period = m_Period-m_packetSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      m_Remainder=m_Remainder+m_headerSize;
      SendPacket();
    }
  }else{
    int i = find_i(m_Period);
    int a = find_a(i);
    int b = find_b(i);
    m_packetSize = find_packet_size(m_Period, a, b);
    if (m_f==1 && jammed<m_P){
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
      m_Period = m_Period - m_packetSize;
      m_totalOverhead =  m_totalOverhead + m_headerSize;
      m_Remainder=m_Remainder+m_headerSize;
      SendPacket();
      m_packetSize = m_Period;
      m_f=m_f-1;
    }
    m_useFull = m_useFull + m_packetSize - m_headerSize;
    m_Period = m_Period - m_packetSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    m_Remainder=m_Remainder+m_headerSize;
    SendPacket();
    m_packetSize=0;
  }
}


void
MyApp::ADP2(void){
  int a = 2;
  int b = 16;
  int c = 8;
  double jammed = (rand()%100)/100.00;
  if(m_Period <= (m_f+1)){
    m_packetSize = m_Period;
    m_Period = m_Period - m_packetSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    m_Remainder=m_Remainder+m_headerSize;

    SendPacket();
    if (jammed<m_P && m_f>1){
      m_f = m_f -1;//packet jammed
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
    }else{
      m_useFull = m_useFull + m_packetSize - m_headerSize ;
    }
  }else{
    m_packetSize = find_packet_size2(m_Period,a,b,c);
    m_Period = m_Period - m_packetSize;
    m_packetSize = m_packetSize;
    m_totalOverhead =  m_totalOverhead + m_headerSize;
    m_Remainder=m_Remainder+m_headerSize;
    SendPacket();
    if (jammed<m_P && m_f>1){
```

```
      m_f = m_f -1;//packet jammed
      m_dropped = m_dropped +  m_packetSize - m_headerSize;
    }else{
      m_useFull = m_useFull + m_packetSize - m_headerSize; ;
    }
  }
}

void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
      m_totalTime = m_totalTime + (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ()));
    }
}

int main (int argc, char *argv[]){
  int T= std::atoi(argv[1]);
  int f = std::atoi(argv[2]);
  int headerSize =  std::atoi(argv[3]);
  double r = std::atof(argv[4]);
  float P = std::atof(argv[5]);
  int epa = std::atoi(argv[6]);

  Results.open ("anwser/Alg2Char_Result.txt", std::fstream::in | std::fstream::out | std::fstream::app);
  Alg_work.open("anwser/Alg2Char_work.txt",std::fstream::in | std::fstream::out | std::fstream::app);
  Alg_work<<"____________________________________________________________________"<<"\n";

  int count = 0;
  double startTime = 0.0;
  double stopTime = 10.0;
  srand ( time(NULL) );
  NodeContainer wifiStaNodes;
  wifiStaNodes.Create (2);

  YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
  YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
  phy.SetChannel (channel.Create ());

  WifiHelper wifi = WifiHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::AarfWifiManager");
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0));

  NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();

  NetDeviceContainer staDevices;
  staDevices = wifi.Install (phy, mac, wifiStaNodes);

  MobilityHelper mobility;
  mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                 "MinX", DoubleValue (0.0),
                                 "MinY", DoubleValue (0.0),
                                 "DeltaX", DoubleValue (50.0),
                                 "DeltaY", DoubleValue (20.0),
                                 "GridWidth", UintegerValue (3),
                                 "LayoutType", StringValue ("RowFirst"));

  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (wifiStaNodes);

  InternetStackHelper stack;
  stack.Install (wifiStaNodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.5.0", "255.255.255.252");
  Ipv4InterfaceContainer interfaces = address.Assign (staDevices);

  uint16_t sinkPort = 8080;
  Address sinkAddress (InetSocketAddress (interfaces.GetAddress (0), sinkPort));
  PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
  ApplicationContainer sinkApps = packetSinkHelper.Install (wifiStaNodes.Get (0));
```

```
  sinkApps.Start (Seconds (startTime));
  sinkApps.Stop (Seconds (stopTime));

  Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (wifiStaNodes.Get (1), TcpSocketFactory::GetTypeId ());

  do{
    std::cout<<count<<" - "<<f<<"/200\n";
    Ptr<MyApp> app = CreateObject<MyApp> ();
    app->Setup (ns3TcpSocket, sinkAddress, 1040, DataRate ("1Bps"),T,f,0,0,0,headerSize,count,P,epa,r);
    wifiStaNodes.Get (0)->AddApplication (app);
    app->SetStartTime (Seconds (startTime));
    app->SetStopTime (Seconds (stopTime));
    count++;
  }while(count!=epa);



  Simulator::Stop (Seconds (stopTime));
  //AnimationInterface anim ("alg02.xml");
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}


int find_i(int T){
  int i;
  unsigned int low;
  unsigned int high;
  i = 1;
  do{
    i++;
    low  = (((i-1)*i)/2)+1;
    high  = (((i+1)*i)/2)+1;
    if((unsigned)T >= low && (unsigned)T<high){
      return i;
    }
  }while(true);
}

int find_a(int i){
  return (i-2);
}

int find_b( int i){
  return ((((unsigned)(i-1)*i)/2)-1);
}

int find_packet_size(int T, int a, int b){
  return (((unsigned)T+(unsigned)b)/((unsigned)a+2));
}

int find_packet_size2(int T, int a, int b, int c){
  unsigned int  packetSize;
  do{
    do{
      packetSize = ((a*T+ b)/c);
      a++;
      b++;
      c++;
    }while(packetSize<=0);
  }while (packetSize>(unsigned)T);

  return packetSize;
}
```

Παράρτημα Δ

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <math.h>
#include <iomanip>
#include <sstream>
#include <cstring>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/netanim-module.h"

using namespace ns3;
float find_P();

double packet_size (double P,double r, int s);

std::fstream Results;


class MyApp : public Application
{
public:
  MyApp ();

  void Setup (Ptr<Socket> socket, Address address, DataRate dataRate,int maxUnit, int headerSize, double r, double P,int
count,double Pit,int epa);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);
  void uniform(void);

  Ptr<Socket>     m_socket;
  Address         m_peer;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  uint32_t        m_packetsSent;
  bool            m_running;
  double          m_packetSize;
  double          m_useFull;
  double          m_dropped;
  int             m_overhead;
  int             m_maxUnit;
  int             m_headerSize;
  int             m_s;
  double          m_r;
  double          m_P;
  double          m_nextCharching;
  double          m_totalTime;
  int m_count;
  double m_Pit;
  int m_epa;
  double test;
  int             m_countP;
  float           m_AveP;


};

//constructor
MyApp::MyApp ()
```

```
  : m_socket (0),
    m_peer (),
    m_dataRate (0),
    m_sendEvent (),
    m_running (false),
    m_packetSize(),
    m_useFull(),
    m_dropped(),
    m_overhead(),
    m_maxUnit(),
    m_headerSize(),
    m_s(),
    m_r(),
    m_P(),
    m_nextCharching(),
    m_totalTime(),
    m_count(),
    m_Pit(),
    m_epa(),
    test(),
    m_countP(),
    m_AveP()


{
}

void
MyApp::Setup (Ptr<Socket> socket, Address address, DataRate dataRate, int maxUnit,int headerSize, double r,double P,int
count,double Pit,int epa)
{
  m_socket = socket;
  m_peer = address;
  m_dataRate = dataRate;
  m_packetSize=0.0;
  m_useFull=0.0;
  m_dropped = 0.0;
  m_overhead =0;
  m_maxUnit = maxUnit;
  m_headerSize = headerSize;
  m_s = maxUnit;
  m_r = r;
  m_P = P;
  m_packetSize=packet_size(m_P,m_r,m_s);
  m_nextCharching = 1/m_r;
  m_totalTime = 0.0;
  m_count = count;
  m_Pit = Pit;
  m_epa = epa;
  test = 0.0;
  m_countP=0;
  m_AveP=0.0;

}

void
MyApp::StartApplication (void)
{
  m_running = true;
  m_packetsSent = 0;
  m_socket->Bind ();
  m_socket->Connect (m_peer);

  do{
    uniform();
  }while(m_P>0);
  Results << std::fixed;
  Results << std::setprecision(2);

    if (m_AveP == 0)
      Results<<m_useFull<<"\t"<<m_dropped<<"\t"<<m_overhead<<"\t"<<"0.0"<<"\n";
    else
      Results<<m_useFull<<"\t"<<m_dropped<<"\t"<<m_overhead<<"\t"<<(float)(m_AveP/m_countP)<<"\n";
```

```
 if (m_count == m_epa){
   Results.close();
  }

 }



void
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}


void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize+m_headerSize);
  m_socket->Send (packet);
  if (m_P!=0)
    ScheduleTx ();
}

void
MyApp::uniform(void){
  if (m_totalTime>=m_nextCharching){
    m_nextCharching = m_nextCharching + (1/m_r);
    if (m_s < m_maxUnit)
      m_s++;
  }

  double jammed = (rand()%100)/100.00;
  if (m_packetSize>=m_P){
    m_packetSize = m_P;
  }

  SendPacket();
  m_Pit = find_P();
  if (jammed<m_Pit && m_s!=0){
    m_countP++;
    m_AveP = m_AveP+m_Pit;
    m_s--;
    m_dropped = m_dropped + m_packetSize;
  }else{
    m_useFull = m_useFull + m_packetSize;
  }
  m_overhead = m_overhead + m_headerSize;
  m_P  = m_P - m_packetSize;
}


void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
      m_totalTime = m_totalTime + (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ()));
    }
}
```

```
int main (int argc, char *argv[]){
  double P= std::atof(argv[1]);
  std::cout << std::fixed;
  std::cout << std::setprecision(2);
  double r = atof(argv[2]);
  int s = atoi(argv[3]);
  int headerSize = atoi(argv[4]);
  double Pit = std::atof(argv[5]);
  int epa = std::atoi(argv[6]);

  Results.open ("anwser1/uniform_Result.txt", std::fstream::in | std::fstream::out | std::fstream::app);

  int count = 0;

  double startTime = 0.0;
  double stopTime = 10.0;
  srand ( time(NULL) );
  NodeContainer wifiStaNodes;
  wifiStaNodes.Create (2);

  YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
  YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
  phy.SetChannel (channel.Create ());

  WifiHelper wifi = WifiHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::AarfWifiManager");
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0));

  NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();

  NetDeviceContainer staDevices;
  staDevices = wifi.Install (phy, mac, wifiStaNodes);

  MobilityHelper mobility;
  mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                 "MinX", DoubleValue (0.0),
                                 "MinY", DoubleValue (0.0),
                                 "DeltaX", DoubleValue (50.0),
                                 "DeltaY", DoubleValue (20.0),
                                 "GridWidth", UintegerValue (3),
                                 "LayoutType", StringValue ("RowFirst"));

  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (wifiStaNodes);

  InternetStackHelper stack;
  stack.Install (wifiStaNodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.5.0", "255.255.255.252");
  Ipv4InterfaceContainer interfaces = address.Assign (staDevices);

  uint16_t sinkPort = 8080;
  Address sinkAddress (InetSocketAddress (interfaces.GetAddress (0), sinkPort));
  PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
  ApplicationContainer sinkApps = packetSinkHelper.Install (wifiStaNodes.Get (0));
  sinkApps.Start (Seconds (startTime));
  sinkApps.Stop (Seconds (stopTime));

  Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (wifiStaNodes.Get (1), TcpSocketFactory::GetTypeId ());
  do{
    Ptr<MyApp> app = CreateObject<MyApp> ();
    app->Setup (ns3TcpSocket, sinkAddress, DataRate ("1Bps"),s,headerSize,r,P,count,Pit,epa);
    wifiStaNodes.Get (0)->AddApplication (app);
    app->SetStartTime (Seconds (startTime));
    app->SetStopTime (Seconds (stopTime));
    count++;
  }while(count!=epa);

  Simulator::Stop (Seconds (stopTime));
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
```

```
}


double packet_size (double P,double r, int s){
  double l =  (sqrt (P*(P*r+(s-1)*(1-r))-(P*r)))/((P*r)+s-1);
  return l;
}

float find_P(){
  srand((unsigned)time(NULL));
  float par=0.0;
  par = (float)rand() / RAND_MAX;
  return par;
}
```